
MASCOTS 2020

Nice, France
17–18 November 2020
# Table of Contents

Foreword ........................................................................................................................................ viii
Organizing Committee ................................................................................................................... x
Program Committee ..................................................................................................................... xi
Additional Reviewers .................................................................................................................... xii
Sponsors .......................................................................................................................................... xiii

## Traffic and Network Modelling

- Adversarial attacks in a deep reinforcement learning based cluster scheduler .................. 1  
  *Shaojun Zhang, Chen Wang and Albert Zomaya*
- Incentive mechanism for collective coordination in an urban intelligent transportation system using G-networks ................................................................. 9  
  *Huibo Bi, Erol Gelenbe and Yanyan Chen*
- A Priority-based Dynamic Link Scheduling Algorithm Using Multi-criteria Decision Making in Wireless Body Area Networks ........................................... 17  
  *Beom-Su Kim and Ki-Il Kim*

## Hardware Optimization

- Symbolic Execution for Network Functions with Time-Driven Logic .............................. 25  
  *Harsha Sharma, Wenfei Wu and Bangwen Deng*
- Reliable Reverse Engineering of Intel DRAM Addressing Using Performance Counters .... 33  
  *Christian Helm, Soramichi Akiyama and Kenjiro Taura*
- A NUMA-aware NVM File System Design for Manycore Server Applications ............ 41  
  *June-Hyung Kim, Youngjae Kim, Safdar Jamil and Sangyong Park*
Performance Optimization

New Centrality Measures in Networks and their Applications to the International Trade and Migration Networks .......................................................... 46
  Fuad Aleskerov, Anna Rezyapova, Alina Roman and Vyacheslav Yakuba

Optimum Checkpoints for Time and Energy ........................................... 54
  Erol Gelenbe, Pawel Boryszko, Miltiadis Siavvas and Joanna Domanska

TrimTuner: Efficient Optimization of Machine Learning Jobs in the Cloud via Sub-Sampling ................................................................. 62
  Pedro Mendes, Maria Casimiro, Paolo Romano and David Garlan

Mobile Network Traffic Forecasting Using Artificial Neural Networks ........... 70
  Anil Kirmaz, Diomidis S. Michalopoulos, Irina Balan and Wolfgang Gerstacker

Session Data Driven Systems

Towards a common environment for learning scheduling algorithms ............... 71
  Renato Cunha and Luiz Chaimowicz

Fabsim-X: A simulation Framework for the Analysis of Large-Scale Topologies and Congestion Control Protocols in Data Center Networks ......................... 79
  Malek Musleh, Roberto Penaranda, Allister Alemania, Pedro Yebenes Segura, Gene Wu, Jan Zielinski, Krzysztof Raszkowski, Nan Ni, Scott Diesing, Anupama Kurpad, Ram Huggahalli, Curt Bruns, Steven Miller and Sujoy Sen

Effective Elastic Scaling of Deep Learning Workloads ................................ 87
  Vaibhav Saxena, K. R. Jayaram, Saurav Basu, Ashish Verma and Yogish Sabharwal

Merkle Hash Grids Instead of Merkle Trees .......................................... 95
  Jehan-Francois Paris and Thomas Schwarz

Hardware and Storage

µCache: a mutable cache for SMR translation layer .................................. 103
  Mohammad Hossein Hajkazemi, Mania Abdi and Peter Desnoyers

Symbiotic HW Cache and SW DTLB Prefetching for DRAM/NVM Hybrid Memory ..... 111
  Onkar Patil, Frank Mueller, Latchesar Ionkov, Jason Lee and Michael Lang

Improving NAND flash performance with read heat separation ..................... 119
  Roman Pletka, Nikolaos Papandreou, Radu Stoica, Haris Pozidis, Nikolas Ioannou, Tim Fisher, Aaron Fry, Kip Ingram and Andrew Walls

A Smart Background Scheduler for Storage Systems .................................. 127
  Maher Kachmar and David Kaeli
Performance Evaluation

Investigating Genome Analysis Pipeline Performance on GATK with Cloud Object Storage .................................................. 135
Tatsuhiro Chiba and Takeshi Yoshimura
Self-adaptive Threshold-based Policy for Microservices Elasticity ........................................ 143
Fabiana Rossi, Valeria Cardellini and Francesco Lo Presti
COCOA: Cold Start Aware Capacity Planning for Function-as-a-Service Platforms ....... 151
Alim Ul Gias and Giuliano Casale
Evaluating the Performance of a State-of-the-Art Group-oriented Encryption Scheme for Dynamic Groups in an IoT Scenario ................................................................. 159
Thomas Prantl, Peter Ten, Lukas Iffländer, Alexandra Dmitrenko, Samuel Kounev and Christian Krupitzer
Performance Prediction for Data-driven Workflows on Apache Spark ........................................ 167
Andrea Gulino, Arif Canakoglu, Stefano Ceri and Danilo Ardagna

Cloud and Fog Computing

Voilà: Tail-Latency-Aware Fog Application Replicas Autoscaler ........................................ 168
Ali Fahs, Guillaume Pierre and Erik Elmroth
Baloo: Measuring and Modeling the Performance Configurations of Distributed DBMS ... 176
Johannes Grohmann, Daniel Seybold, Simon Eismann, Mark Leznik, Samuel Kounev and Jörg Domaschka
Instability in Geo-Distributed Kubernetes Federation: Causes and Mitigation........ 184
Mulugeta Ayalew Tamiru, Guillaume Pierre, Johan Tordsson and Erik Elmroth
Security-Performance Trade-offs of Kubernetes Container Runtimes .................. 192
William Viktorsson, Cristian Klein and Johan Tordsson
Infrastructure-Aware TensorFlow for Heterogeneous Datacenters ...................... 196
Moiz Arif, M. Mustafa Rafique, Seung-Hwan Lim and Zaki Malik

Performance Modelling

Model-Aided Learning for URLLC Transmission in Unlicensed Spectrum ............... 204
Ayat Zaki-Hindi, Salah-Eddine Elayoubi and Tijani Chahed
Statistical Learning of Markov Chains of Programs ........................................... 212
Emilio Incerto, Annalisa Napolitano and Mirco Tribastone
Age of Information in an Overtake-Free Network of Quasi-Reversible Queues ........ 220
Ioannis Koukoutsidis
Energy Packet Networks with general service time distribution .................. 226
Youssef Ait El Mahjoub, Hind Castel and Jean-Michel Fourneau
Concept Drift and Avoiding its Negative Effects in Predictive Modeling in Power Plants . . 234
Marek Moleda, Alina Momot and Dariusz Mrozek

Non-Asymptotic Performance Analysis of Size-Based Routing Policies ................. 241
Eitan Bachmat and Josu Doncel
MASCOTS 2020
Foreword

It is a great pleasure and honour for us to introduce the IEEE Computer Society Proceedings of the 28th IEEE International Symposium on the Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, affectionately known as MASCOTS.

This year’s IEEE MASCOTS 2020 was a 3-day conference focusing on the performance modeling and analysis of computer systems and networks, supported by the Institute of Theoretical and Applied Informatics of the Polish Academy of Sciences, co-sponsored by the IEEE Computer Society and the journal Sensors. The technical program for the IEEE MASCOTS 2020 conference included two keynote talks, full papers, a few short papers, and a workshop whose papers will appear separately with Springer LNCS. The contributed papers appearing in these proceedings represent approximately 27% of 124 submitted papers.

We are especially grateful to all those who submitted papers, from all five continents and from more than 27 countries. For the accepted papers, we tried to set up our timetable in a manner that could satisfy the time zone constraints of our speakers, and we are very grateful to our authors who kindly accepted the inconvenience of a conference that finally had to be held only online due to the Covid-19 pandemic, while we had hoped for a meeting in the lovely city of Nice, in a hotel a few tens of meters from the Mediterranean Sea.

To recall the beginnings of this conference series, below we give details of the proceedings of the first five Mascots conferences starting with the one in 1993, and fondly recall the names of some of the other pioneers who were involved in starting these symposia, such as Kallol Vijay Bagchi who energetically convinced several of us, Doog de Groot, Patrick Dowd, Vijay Madisetti, Herb Schwetman, Kishor Trivedi and Jean Walrand.

With regard to this year’s symposium, we are most grateful to the Program Committee who volunteered their high level of competence and valuable time to selecting the accepted papers and promoting the best submitted work. Their work is fundamental to the success of the conference and to the content of these proceedings. We also thank our keynote speakers to whom we are deeply grateful:

- Professor Gabriele Kotsis, of Johannes Kepler University Linz, who generously accepted the invitation to speak, in the middle of her already heavy academic duties and her role as current President of the ACM, and
- Professor Ernesto Damiani of Khalifa University and University of Milano, who kindly accepted while he has the additional constraints of juggling an international career.

Erol Gelenbe, General Chair
Maria Carla Calzarossa, Program Chair
Krysztof Grochla and Konrad Polys, Publicity Chairs
Ricardo Lent, Publications Chair
Tadeusz Czachórski, Finance Chair
References

5th MASCOTS 1997: Haifa, Israel

4th MASCOTS 1996: San Jose, California, USA

3rd MASCOTS 1995: Durham, North Carolina, USA

2nd MASCOTS 1994: Durham, North Carolina, USA

1st MASCOTS 1993: La Jolla, San Diego, CA, USA
Organizing Committee
MASCOTS 2020

General Chair
Erol Gelenbe, IITIS and University of Côte d’Azur

Program Chair
Maria Carla Calzarossa, University of Pavia

Publicity Chairs
Krysztof Grochla, IITIS
Konrad Polys, IITIS

Publications Chair
Ricardo Lent, University of Houston

Finance Chair
Tadeusz Czachórski, IITIS
Program Committee
MASCOTS 2020

Jussara Almeida, Federal University of Minas Gerais, Brazil
Jonatha Anselmi, INRIA, France
Louiza Bouallouche-Medjkourne, University of Bejaia, Algeria
Gilles Bernot, Université Côte d’Azur, France
Andre-Luc Beylot, Institut National Polytechnique de Toulouse, France
Huibo Bi, Beijing University of Technology, China
Jalil Boukholba, IRT, Lab-STICC, Univ. Bretagne Occidentale, Brest, France
Ivona Brandic, Technical University of Vienna, Austria
Maria Carla Calzarossa, University of Pavia, Italy
Valeria Cardellini, University of Roma Tor Vergata, Italy
Emiliano Casalicchio, University of Rome Sapienza, Italy
Suryadip Chakraborty, University of Cincinnati, USA
Claudio Cicconetti, IIT-CNR, Pisa, Italy
Tadeusz Czachórski, IITIS, Polish Academy of Sciences, Poland
Sofiene Djahel, Manchester Metropolitan University, UK
Joanna Domuńska, IITIS, Polish Academy of Sciences, Poland
Josu Doncel, University of the Basque Country, Spain
Jean-Michel Fourneau, University of Versailles, France
Anshul Gandhi, Stony Brook University, USA
Erol Gelenbe, IITIS, Polish Academy of Sciences, Poland and Univ. Côte d’Azur, France
Krysztof Grochla, IITIS, Polish Academy of Sciences, Poland
Yassine Hadjadj-Aoul, IRISA, France
Adelkrim Haqiq, University of Hassan I, Tunisia
Khaled Hussain, Assiut University, Egypt
Alexandru Iosup, Free University of Amsterdam, Netherlands
Alain Jean-Marie, INRIA Sophia-Antipolis, France
Samuel Kounev, University of Wuerzburg, Germany
Ricardo Lent, University of Houston, USA
Aniket Mahanti, University of Auckland, New Zealand
Raymond Marie, University of Rennes, France
Luisa Massari, University of Pavia, Italy
Daniel Menasce, George Mason University, USA
Jehan-Francois Paris, University of Houston, USA
Nihal Pekergin, University of Paris-Est-Creteil, France
Riccardo Pincirol, College of William and Mary, USA
Paolo Romano, Lisbon University, Portugal
Dan Rubenstein, Columbia University, USA
Gerardo Rubino, INRIA, France
Daniele Tessera, Catholic University of Sacred Heart, Italy
Benny van Houdt, University of Antwerp, Belgium
Marco Vieira, University of Coimbra, Portugal
Umberto Villano, University of Sannio, Italy
Carey Williamson, University of Calgary, Canada
Additional Reviewers
MASCOTS 2020

Abdelhamed, Abdelrahman
Afifi, Mahmoud
Ahmed, Ali Hussein
Amaro, Pedro
Barbierato, Enrico
Beierlieb, Lukas
Benhamida, Nadjet
Bononi, Luciano
Boumouni, Mahdi
Bujari, Armir
Campos, João R.
Catillo, Marta
D’Angelo, Gabriele
Dartois, Jean-Emile
Di Felice, Marco
D’abruzzo Pereira, José
Eismann, Simon
Ferretti, Stefano
Gharieb, Reda
Gorawski, Michal
Herrnleben, Stefan
Ifländer, Lukas
Ivaki, Naghmeh
Kim, Hyunbum
Lahlah, Souaad
Lemarchand, Laurent
Leppich, Robert
Lourenço, Nuno
Mahdy, Mohamed
Marfia, Gustavo
Marzolla, Moreno
Nebbione, Giuseppe
Ouarnoughi, Hamza
Pecchia, Antonio
Polys, Konrad
Prantl, Thomas
Semchedine, Fouzi
Slabicki, Mariusz
Slimani, Camélia
Taj-Eddin, Islam
Tan, Sovanna
Züfle, Marwin
Adversarial Attacks in a Deep Reinforcement Learning based Cluster Scheduler

Shaojun Zhang\textsuperscript{1}, Chen Wang\textsuperscript{2}, and Albert Y. Zomaya\textsuperscript{1}

\textsuperscript{1}School of Computer Science, The University of Sydney, Australia
\textsuperscript{2}CSIRO Data61, Sydney, Australia

szha6955@uni.sydney.edu.au, chen.wang@csiro.au, albert.zomaya@sydney.edu.au

Abstract—A scheduler is essential for resource management in a shared computer cluster, particularly scheduling algorithms play an important role in meeting service level objectives of user applications in large scale clusters that underlie cloud computing. Traditional cluster schedulers are often based on empirical observations of patterns of jobs running on them. It is unclear how effective they are for capturing the patterns of a variety of jobs in clouds. Recent advances in Deep Reinforcement Learning (DRL) promise a new optimization framework for a scheduler to systematically address the problem. A DRL-based scheduler can extract detailed patterns from job features and the dynamics of cloud resource utilization for better scheduling decisions. However, the deep neural network models used by the scheduler might be vulnerable to adversarial attacks. There is limited research investigating the vulnerability in DRL-based schedulers. In this paper, we give a white-box attack method to show that malicious users can exploit the scheduling vulnerability to benefit certain jobs. The proposed attack method only requires minor perturbations in job features to significantly change the scheduling priority of these jobs. We implement both greedy and critical path based algorithms to facilitate the attacks to a state-of-the-art DRL based scheduler called Decima. Our extensive experiments on TPC-H workloads show a 62\% and 66\% success rate of attacks with the two algorithms. Successful attacks achieve a 18.6\% and 17.5\% completion time reduction.

Index Terms—scheduling, deep reinforcement learning, robustness, adversarial attack, directed acyclic graph

I. INTRODUCTION

Normally, there is a large number of jobs from a variety of users running in the cloud. Database queries and data analytic pipelines are typical jobs that run in such clusters in the cloud. These jobs often involve dependency-aware workflows. The workflows are normally represented by Directed Acyclic Graphs (DAGs), with tasks of jobs dispatched to available computing resources to run by cluster schedulers. While a lot of work [1]–[7] has been done on heuristic scheduling policies, there still lacks of a systematic way to design optimal scheduling policies.

Recently, Deep Reinforcement Learning (DRL) [8] has been applied to many real-world problems in the large scale clusters [9]–[11]. Its ability to make decisions through interaction with the system inspired a new paradigm for resource provisioning and job scheduling. A DRL model is trained either from scratch or from exiting schedules. System performance metrics, such as throughput or make-span, are incorporated in reward functions to train an optimized task assignment policy. Many work [12]–[14] have demonstrated the superiority of policies learnt by DRL-based schedulers over the traditional heuristic policies.

However, the limitations of DRL-based scheduling algorithms are not well studied so far. In cloud computing, users have certain Service Level Objectives (SLOs) for their jobs submitted to run. It is not straightforward to include various SLOs into a reward function for optimization in deep reinforcement learning [15], [16]. More importantly, the Deep Neural Networks (DNNs) models are known to be vulnerable to adversarial attacks. While adversaries on DNNs have been investigated for years and there are works on addressing the problems in DRL-based controlling systems [17]–[19], the vulnerability and its impact on DRL-based scheduling algorithms have not been examined thoroughly.

In a multi-tenant cloud platform, we intend to answer the question about how a user can take advantage of the vulnerability in DRL models to make the scheduler prioritize her own jobs.

Similar to DNN models, on which adversaries induce misclassifications by perturbations on input data samples, the scheduler can also be manipulated to take a scheduling action that benefits the attacker who deliberately perturbs certain features of her own jobs through exploiting the knowledge on DRL models used by the scheduler. However, there are two main challenges that make attacking DRL-based scheduling more complicate than attacking a DNN model: firstly DRL-based scheduling involves a sequence of task assignment actions and each action inevitably changes the state of the input of the scheduler, therefore when to apply perturbation is a problem; secondly, the jobs of a cluster are often represented as DAGs, how to attack a DRL model that extracts patterns from graph represented jobs is a challenge. To our knowledge, we are not aware of existing work investigating these two aspects: (1) how the scheduling decision is manipulated and shifted from an optimal policy, and (2) how attacks are crafted on the input job DAGs.

In this paper, we give a detailed analysis for the white-box attacks in job scheduling in a multi-tenant cluster and propose a method to do so efficiently (Section III). Jobs, represented as DAGs, have dependencies between their nodes, each of which represents a collection of tasks. The DRL-based scheduler maps tasks from different jobs to computing resources by taking into account of node features and their dependency patterns. In this process, however, an attacker might obtain the parameters and structures of the neural network models, and impose perturbations on a handful of features or edges of her job such that certain tasks are given higher priority to be assigned to resources. We show that the perturbation can be
crafted through calculating the gradients of the DRL model’s outputs with respect to job DAGs (Section IV). We propose both a greedy and a critical path based algorithm to achieve this goal, and demonstrate the effectiveness of these algorithms through a prototype system called PertSchedule (Section V). The simulation on Decima [13], the state-of-the-art DRL-based scheduler, have shown promising results (Section VI). In total, 62% and 66% perturbed jobs have an early completion (18.6% and 17.5% less completion time on average) in the system with the greedy and critical path algorithm respectively. The success rate of early completion increases to 76% with both algorithms if perturbation budget increases.

This paper is organized as follows: Section II introduces the background and related work; Section III describes the threat model; Section IV gives the two algorithms; Section V provides the details of implementation; And Section VI presents the evaluation results followed by conclusion.

II. BACKGROUND AND RELATED WORK

A. Dependency-aware job scheduling

Scalable data analytics engines, such as Spark [20], Storm [21], Hadoop [22], and Tez [23], often represent their jobs as Directed Acyclic Graphs (DAGs). In the DAG, the node encapsulates a collection of homogeneous tasks, which always run the same code, and edges determines dependencies between tasks. Jobs from different users are managed by the scheduler and employ parallel execution whenever possible. This dependency-aware scheduling problem is often resolved in the algorithmic manner by heuristic schedulers. For example, the Critical Path First (CPF) scheduler [24] assigns tasks for execution by estimating the run time along each job’s critical path. Graphene [25] packs tasks regarding the available resource-time space for all jobs, and overlaps the long-running tasks to guarantee low latency, high throughput and comparative fairness.

B. Deep Reinforcement Learning-based scheduler

Deep Reinforcement Learning (DRL) reformulates job scheduling from a series of decision-making actions to a Markov decision process. The scheduler is trained during episodic interaction with the system [26]. Specifically, it learns the overall pattern of jobs and maps their tasks to the computing resources by encompassing system status and job features in the high dimensional input space. Training the scheduler will also requires setting a high-level objective, e.g. minimal Job Completing Time (JCT), to its rewards function. By maximizing the rewards, the scheduler can even generalize the learned scheduling policy to job patterns it has not seen before.

Current DRL based schedulers are able to outperform their heuristics counterparts on job DAG scheduling. For example, the state-of-the-art scheduler, Decima [13], adopts a policy gradient [27] model, which achieves 21% less average JCT than the fine-tuned fair scheduler. Another example, Spear [12], implements an improved Monte Carlo Tree Search (MCTS) model, which reduces job makespan by 20% compared to Graphene [25]. Moreover, GoDAG [28] utilizes an actor-critic model [27] to realize a better packing strategy on multi-resources than naive CPF [24].

C. Adversarial Attacks to neural networks

Most neural networks are susceptible to malicious attacks. Attacks are often classified into two categories, white-box attacks and black-box attacks. The former needs knowledge about the model structure and parameters [17] while the latter assumes no prior knowledge about the model structure and parameters. For white-box attacks, Fast Gradient Sign Method (FGSM) [17] and Jacobian-based Saliency Map Approach (JSMA) [29] are commonly used in attacking models for image classification. They introduce minimal perturbations on image pixels based on the model gradients that lead to prediction change [30]. Recently, the work in [31] applies a variation of gradient sign method called integrated gradients-guided attack on graph data which misleads node classification in graph convolutional networks (GCNs).

Even though attacks on DRL model share similarity with attacks on other deep neural networks, there is no existing work on understanding the risk of such attacks on a DRL-based job scheduler in clusters.

III. THREAT MODEL

In this section, we define the threat model to a DRL based cloud scheduler.

A. Job scheduling in the Cloud

A computer cluster in a cloud computing platform serves a variety of users. These users submit jobs to the system to run. Each job contains multiple tasks with dependency to each other, and is maintained in the queue of the cloud platform. A cluster scheduler maps tasks of multiple jobs in the queue to available computing resources called executors. The scheduling decision is triggered when an executor becomes available, which represents a schedule stage. A DRL-based scheduler continuously learns a policy to map tasks efficiently to executors based on job features and resource utilization status. Normally, it predicts a probability that a task is assigned to resources in the next scheduling stage, and selects the task with the highest probability to run.

B. Attacking scenario

We consider white-box attack in this paper, which means the model is accessible by the attacker. In practice, an attacker may obtain the model and parameters through techniques such as model stealing [32]. When submitting a job, the attacker can make minor change to certain job features without noticing by the scheduler. The attacker can also modify features of submitted jobs. The feature change of the job is done by perturbing features according to the gradient direction of the DRL model used by the scheduler on the job. An attack is marked ‘successful’ only if the probability of the attacker’s task is higher than those in the queue. Note that, the feature
change does not necessarily happen at every scheduling stage because a single perturbation is likely to achieve the completion time objective for the whole job. Also the feature change is constrained by the original features. A perturbation of features may fail if there is no room for feature change in a job.

Attackers can only change their own job DAGs, either node features or the edges but not other users’ jobs. Two kinds of feasible perturbations are shown in Figure 1. The node in a job contains multiple tasks, of which the total number and duration of each are feasible to change. The edge connecting two nodes indicates the dependency between two collections of tasks, and can be fabricated between nodes without dependency. Take database system as an example, the DAG for a query plan can establish edges with the operation ‘join’ between two tables [33], which might then be utilized for perturbation. It is worth noting that a fake edge does not affect the execution of tasks other than incurring trivial data transfer overhead between those nodes. Perturbations like these have little impact on the functionality of original jobs but are likely to significantly reduce their waiting time in the cluster scheduling system.

IV. METHODOLOGY

In DRL-based scheduler, job DAGs from different users form a large graph at each scheduling stage. In a job DAG, homogeneous tasks form a single task node, i.e., a task node may contain instances running the same code on different executors in parallel. The scheduler allocates resources based on task nodes. It first selects the node with the highest dispatching probability in the graph, and then assigns its task to run on executors. In this section, we introduce the gradient-based perturbations to attack the vulnerability of a DRL-based scheduler.

A. Formalization

The end-to-end model \( F \) in the DRL-based scheduler uses graph neural networks (GNNs) to represent the job features and task dependency, and a DNN for prediction of the schedule probability. Given the overall job feature space \( X \) and edges (task dependency) \( E \), the model output \( F(X, E) \) produces a probability distribution for all nodes (of which the total number is \( N \)), indicating how likely selecting them to run next can minimize the overall job completion time. Normally, a task from the node producing \( max_i(F_i(X, E)) \) is scheduled for execution, where \( i \) is in the range of \( (1, N) \). When an attacker wants to preempt job \( J^* \) to run, she needs to calculate the features to change in \( J^* \) that are most likely to lead to an early completion of the job while keeping the feature perturbation minimal.

There are two types of feature perturbations in a graph, changing the node attributes and changing the node connections, which we describe below.

Node feature perturbation. First, we compute the gradient \( G_N \) of \( F(X, E) \) w.r.t nodes features of the target job \( J^* \) as follows:

\[
G_N(F, J^*) = \frac{\partial F}{\partial J^*} = \left[ \frac{\partial F_t}{\partial x_j,k} \right]_{N,K} \tag{1}
\]

where \( F_t \) indicates the target output component we wish to improve, which refers to the node to prioritize in job \( J^* \). And \( x_j,k \) is the \( k^{th} \) feature of the \( j^{th} \) node in the target job. \( N \) is the number of task nodes in the job, and \( K \) is the number of features of a node.

Basically, absolute gradients on the input features indicate the relative importance these features on the targeted output. Perturbation on important features is likely to flip the prediction. Our objective is to maximize the change in completion time to benefit the objective with minimal perturbation in the input features. The task node \( m \) and feature \( n \) to perturb are therefore selected by the following equation:

\[
m, n = \arg \max_{j,k} (\text{abs}(\frac{\partial F_t}{\partial x_j,k})) \tag{2}
\]

The features are perturbed progressively along the gradient direction to ensure a minimal change to trigger the change of scheduling decisions. Each perturbation step in the following incurs a small fraction of feature change denoted by \( \delta \):

\[
x_{m,n} = x_{m,n} + \delta \cdot \text{sign}(\frac{\partial F_t}{\partial x_{m,n}}) \tag{3}
\]

Edge perturbation. In job DAGs, edges are often represented by binary values of 0 or 1 in the adjacent matrix. Edge perturbation is restricted to change corresponding value from 0 to 1. In other words, perturbation by adding edges is allowed, but removing edges is not feasible because it breaks the functionality of the job.

On the other hand, common perturbation methods, such as FGSM and JSMA, do not give sufficient information about the direction for edge perturbation when the edge is represented by binary values. We therefore use an integrated-gradient based method to infer the perturbation direction following [31]. Integrated-gradient based method [34] approximates the edge perturbation by interpolating intermediate values \( x' \) between an all-one or all-zero adjacent matrix and the target graph \( x \). It then calculates the gradients for each intermediate graph and sums them up as a gradient score. The process is defined as below:

\[
G(F, x) = (x - x') \times \int_{\alpha=0}^{1} \frac{\partial F(x' + \alpha x(x - x'))}{\partial x} d\alpha \tag{4}
\]

Specifically for edge addition perturbations, our method starts from a fully connected adjacent matrix and iteratively
Algorithm 1 Perturbation algorithm $F$ is the DRL $\epsilon$

$X$ is the current feature space, $E$ is the edges, $J^*$ is the job DAG, $\delta$ is the perturbation step, and $\gamma$ is the max ratio for the perturbation allowed on an feature.

Require: $F$, $X$, $J^*$, $\delta$, $\gamma$

1: $X^* \leftarrow X$
2: $\Omega^*$: frontier nodes from $J^*$
3: $\Omega^*_c$: nodes in the critical path of $J^*$
4: for Node $N^*$ in $\Omega^*$ (or $\Omega^*_c$) do
5:   $F_i$: the component that refers to $N^*$
6:   $G^*_e(F_i,J^*) = [G(F_i,e_{j,k})]_{N,N}$
7:   $m,n = \text{argmax}_{j,k}([G(F_i,e_{j,k})]_{N,N})$
8:   $e_{m,n} = 1$  \(\triangleright\) edge perturb
9:   $G^*_N(F_i,J^*) = \left[ \frac{\partial G^*_e}{\partial e_{j,k}} \right]_{N,K}$
10: $m,n = \text{argmax}_{j,k}(G^*_N(F_i,J^*))$
11: $\text{sum} = 0$  \(\triangleright\) total node perturbation
12: while $\text{sum} \leq \gamma \cdot x_{m,n}$ and $F(X^*) \neq N^*$ do
13:   $x_{m,n} = x_{m,n} + \delta$  \(\triangleright\) node perturbation
14:   $\text{sum} = \text{sum} + \delta$
15: end while
16: if $F(X^*,E^*) = N^*$ then
17:   break  \(\triangleright\) perturbation done
18: else
19:   $X^* = X$  \(\triangleright\) perturbation discarded
20: end if
21: return $J^*$

removes edges according to the gradient directions. The integrated gradients are applied to the adjacency matrix $E^*$ of the whole graph as follows:

$$G^*_e(F_i,J^*) = [G(F_i,e_{j,k})]_{N,N}$$

in which $e_{j,k}$ belongs to $E^*$, $N$ is the total nodes, and $G(F_i,e_{j,k})$ is the discrete approximation of (4):

$$G(F_i,e_{j,k}) = \frac{1 - e_{j,k}}{m} \sum_{l=1}^{m} \frac{\partial F_i(X,e_{j,k} + \frac{1}{m}(1-e_{j,k}))}{\partial e_{j,k}}$$

Afterwards, node $m$ and $n$ are selected if they produce a maximum integrated gradient score:

$$m,n = \text{argmax}_{j,k}([G(F_i,e_{j,k})]_{N,N})$$

An edge is added between $m$ and $n$ if they belong to the job the attacker intends to manipulate.

B. Perturbation Algorithm

The scheduling decision in a cluster is triggered when a scheduled task finishes execution or a new job is submitted to the system. The DAGs of jobs in the system are dynamic. We give two algorithms to determine when to perturb a job.

Greedy algorithm. The attacker’s job $J^*$ has a collection of frontier task nodes, denoted by $\Omega^*$, whose dependencies have already been satisfied. In the greedy algorithm, gradient-based perturbation attempts to prioritize one frontier node belonging to the attacker’s job to run first. Given the white-box assumption, the attacker obtains the model $F$, original node features $X$ and adjacent matrix $E$. As shown in Algorithm 1, edge perturbation is attempted first as edge addition does not affect a task’s resource requirement. An iterative process is performed to perturb node features. The iteration process intends to minimize the difference between a perturbed job and the the origin job. We call the difference perturbation ratio, denoted by $\gamma$. When a frontier task node of $J^*$ achieves the highest probability from the model output, the perturbation is considered successful. If none of the frontier task nodes of $J^*$ has the highest probability after perturbation, the attack in this scheduling stage fails and all perturbations are discarded. A new perturbation will be attempted in the next scheduling stage when the graph changes. Note that, the perturbation is only done at most once to a job.

Critical-path based algorithm. A greedy algorithm tries to advance each frontier task node of job $J^*$ to preempt its execution. Sometimes prioritization of tasks deep in the DAG of $J^*$ is more effective if these tasks are in the critical path of the job. We give a critical path aware perturbation algorithm to exploit the potential. A critical path [35] is defined as the sequence of task nodes that add up to the longest duration for the job. We denote nodes along the path as $\Omega^*_c$. Completion time of the last task in $\Omega^*_c$ is the final job completion time, or JCT. Hence, Algorithm 1 prioritizes nodes in $\Omega^*_c$ for perturbation. By doing so, the collection of nodes to be prioritized is much smaller at each scheduling stage. The critical path aware perturbation has a high likelihood to reduce the completion time of job $J^*$.

V. IMPLEMENTATION

We implement a prototype system, PertSchedule, to demonstrate the effectiveness of attacks on DRL-based job schedulers. We integrate PertSchedule with the state-of-the-art DRL-based scheduling system Decima [13]. Decima uses TPC-H [36] queries for performance evaluation on Spark. Each query is compiled as a job DAG, of which the node contains homogeneous tasks.

As visualized in Figure 2, Decima utilizes the DRL model to predict nodes for schedule, and PertSchedule uses the same model for job perturbation. Specifically, the DRL model contains two cascaded neural networks: (i) a GCN, which perceives and aggregates per-node information along the connections with 3-layer fully-connection network (the number of neurons at each layer is 16, 8, and 8); (ii) a DNN, which
acquires the embedding of GCN and produces probability for each node with 4-layer fully-connected network (the number of neurons at each layer is 32, 16, 8, and 1) and a softmax layer. The node features, as part of normal scheduler input, are composed of the number of tasks within it, and the estimated duration of each task. Additionally, Decima incorporates some key system status, such as available executors in the system and locality of the executors for each job, as supplementary input. PertSchedule only exploits the first group of input.

VI. EVALUATION

Experimental setup. The DRL model trained by Decima aims to reduce the average JCT of jobs in the system. Decima outperforms common heuristic schedulers. In order to evaluate PertSchedule, we generate 50 TPC-H jobs using the simulator included in Decima. These jobs contain around 435 nodes and more than 30000 tasks in total. There are 10 executors in our setting. The task duration on each executor is estimated based on running real tasks in a small cluster. To keep perturbation minimal, the amount of node feature perturbation per iteration is set to 20ms for task duration and 5 for the number of tasks. The perturbation budget, or the maximum ratio of feature values allowed for perturbation, $\gamma$ is set to 10%.

A. Effectiveness of the greedy algorithm

Overall performance. We evaluate PertSchedule by attacking the 50 jobs one by one. As shown in Figure 3, the similarity between node features for the attacked job and the original job is high (red dashes). The the job DAG difference with edge perturbation is measured by the Hamming distance. The distance is low. The ratio of benefits, i.e., the decrease of JCT and waiting time, after applying the greedy algorithm is shown in Figure 4. In total, PertSchedule successfully makes 31 out of 50 jobs finish earlier after perturbation, accounting for a 62% success rate. The average decrease of JCT for these jobs is 18.6%. The waiting time (the time before the job’s first task being scheduled) of 25 jobs perturbed decreases by 41.5%. Meanwhile, the attack fails on 12 jobs, which causes slight delay for these jobs (1.89% increase of JCT in average). Meanwhile, because of the constraint on the maximum perturbation ratio, 7 jobs cannot be perturbed. We also note that our greedy algorithm, while being successfully in reducing the completion time of most jobs, may delay a targeted job for preempting a task node not on a critical path to run first.

Case study. In order to understand how perturbation is crafted for an attack, we conducted two case studies respectively on Job 10 and Job 14. As shown by their DAG structures in Figure 5, Job 10 (upper) contains 11 nodes and Job 14 (lower) contains 4 nodes. Each node is composed of different number of tasks with estimated per-task execution time (in seconds), as shown by digits inside the node.

Attack on Job 10 is crafted by node perturbation at schedule stage 4. The frontier nodes, which have no dependencies or whose dependencies have already finished, of Job 10 are Node \{0, 1, 3\}. The attack is successfully crafted to prioritize one of them (namely Node 3). As indicated by the gradients from Eq (2), the feature to be perturbed is the number of tasks in Node 3. Specifically, 1 out of 18 tasks of Node 3 is concealed deliberately, as shown by the job (upper right) after the attack in Figure 5. Although the difference is quite minor between two job DAGs, it is enough to manipulate the DRL model to

![Fig. 3. Similarity between the original job and job after attack by PertSchedule in the Greedy manner. Deshlines show the Cosine similarity for node features, and bars show the Hamming distances for the edges (which are represented as matrix). Blank bars indicate zero Hamming distance.](image)

![Fig. 4. The cumulative distribution of jobs in terms of decrease of job completing time (JCT) (left) and decrease of waiting time (WT) (right). When greedy attack is applied, jobs achieve 18.6% decrease on JCT and 41.5% decrease of WT on average. To compare, when attacked based on critical path, 17.5% and 47.9% decrease on JCT and WT are achieved on average.](image)
The node completion time of Job 11 is illustrated in Figure 10. Basically, execution is finished based on node prioritization of Job 10. Delayed by an average rate of 10.9% due to prioritization of the node 1 results in early adjustment on number of tasks in Node 5 from 117 to 115, as shown in the upper sub-figure of Figure 6. Actually, the execution flow follows the same pattern, while node completion time after attack is reduced by 78.9% on average. The prioritization of Job 10 draws deep effects on the system in the upper-side Figure 7, the original schedule (blue dash line) is biased due to the attack. Actually, 19 jobs are enabled early execution at following schedule stages without attack is compared in the upper sub-figure of Figure 6. Meantime, the per-node completion time for Job 10 with and without attack is illustrated in Figure 10. The node completion time after attack is reduced by 78.9% on average. The prioritization of Job 10 draws deep effects on other jobs. As shown by the decrease of remaining job in the hierarchy within the DAG structure. Specifically, its frontier nodes of Job 14 are Node 5, 6, 8, 10, 11, 12, 13. The greedy algorithm is carried out at the schedule stage 368 to prioritize Node 7, one of the frontier nodes. Specifically, perturbations are crafted as follows: per-task duration is adjusted from 1.86s to 1.68s for Node 0, 0.07s to 0.063s for Node 4, and 1.97s to 1.78s for Node 3, while the number of tasks in Node 7 is adjusted from 27 to 25, as shown by the shaded nodes and features in the middle Figure 9. On the contrary, the critical path algorithm succeeds at the scheduling stage 398 to assign the highest priority to Node 5, which is on the job’s critical path. The perturbation is crafted in a more efficient manner with adjustment on number of tasks in Node 5 from 117 to 115, as shown in the right-hand side of Figure 9.

Overall analysis. The effectiveness of the critical path based attacks are evaluated with the same 50 jobs. The reduction of JCT and waiting time is shown in Figure 4. In total, 33 jobs obtain shorter completion time (17.5% decrease on average), accounting for a perturbation success rate of 66%, and 23 jobs achieve less waiting time (time before a job’s first task being scheduled) (47.9% on average). It outperforms the greedy algorithm. Note that, changes on jobs are not noticeable, as shown by the similarity measure in Figure 8. Overall, it shows the effectiveness of prioritizing nodes along critical paths.

Comparison of greedy and critical-path based algorithm. We conducted a case study on Job 11 to illustrate the subtle difference between the effects of two attack algorithms. Specifically, the greedy algorithm on Job 11 causes a slight delay from 709.5s to 718.3s, while the critical-path based algorithm leads to an early completion of 669.4s. As shown in Figure 9, Job 11 has 14 nodes and a complex hierarchy within the DAG structure. Specifically, its frontier nodes include Node {0, 1, 3, 5, 7, 9}, while the nodes on the critical path (red arrows in right-hand side of Figure 9) are such as Node {5, 6, 8, 10, 11, 12, 13}. The greedy algorithm is carried out at the schedule stage 368 to prioritize Node 7, one of the frontier nodes. Specifically, perturbations are crafted as follows: per-task duration is adjusted from 1.86s to 1.68s for Node 0, 0.07s to 0.063s for Node 4, and 1.97s to 1.78s for Node 3, while the number of tasks in Node 7 is adjusted from 27 to 25, as shown by the shaded nodes and features in the middle Figure 9. On the contrary, the critical path algorithm succeeds at the scheduling stage 398 to assign the highest priority to Node 5, which is on the job’s critical path. The perturbation is crafted in a more efficient manner with adjustment on number of tasks in Node 5 from 117 to 115, as shown in the right-hand side of Figure 9.
dependencies: the frontier nodes go first and then the others. However, completion time of parallel node only has a weak correlation, and even differs hundreds of seconds (see Node 7 and Node 8). Actually, benefit of less completion time can only transmit along the critical path, and inappropriate node prioritization may incur contention between parallel nodes. Hence, even though Node 7 finishes earlier with the greedy attack, the overall JCT increases (shown by red line in Figure 10). To compare, critical path aware attack prioritizes Node 5, and thereby decreases the overall JCT significantly (shown by green dots in Figure 10).

C. Effects of the maximum perturbation ratio

To conduct adversarial attacks on job DAGs in practice, we need to configure a perturbation budget, or maximum perturbation ratio on the values of node features ($\gamma$ in Algorithm 1). Normally, a higher budget often leads to a higher perturbation success rate. As shown in Figure 11, the number of jobs, which benefit from perturbations and thereby have shorter JCT (green bars), increases as $\gamma$ grows. The success rate hits the peak around 76% for both the greedy algorithm and the critical path based algorithm. In the meantime, the number of jobs that are delayed (blue bars) or not affected by perturbations (orange bars) decreases as $\gamma$ increases. However, a higher perturbation ratio also leads to a higher chance of the perturbed job being detected by a job monitoring system in a cluster.

VII. CONCLUSION

In this paper, we studied the robustness of a deep reinforcement learning based scheduler in multi-tenant clusters. We showed how to effectively attack the scheduler through job feature perturbations. We gave a gradient-based method to craft perturbations and proposed a greedy algorithm and a critical path-based algorithms to preempt certain jobs through job feature perturbation. We demonstrated the effectiveness of these algorithms in a prototype system. Experiments showed the state-of-the-art DRL-based scheduler is vulnerable to such adversarial attacks.
VIII. DISCUSSION AND FUTURE WORK

Perturbation on multiple jobs. Many cloud platforms allow a single user to submit multiple jobs for execution. In such scenarios, the attacker can perturb these jobs simultaneously to maximize the benefit. As a result, the perturbation algorithms should consider the overall benefit of jobs and avoid contention in perturbed jobs.

White-box Vs. black-box attacks. Gradients-based attack follows the white-box assumption that the neural network model is accessible by attackers. However, under black-box settings, a normal approach is to re-train a separate model for the same input and output space [37]. The approximate model is then utilized to compute the gradients and craft perturbations accordingly. Hence, two settings do not have many differences other than training the approximate model.

Defence from adversarial attacks. Basically, study of attacks aims to strengthen the system. However, defence from job attacks is still an open question for the scheduler. To certain degree, our attacking methods also provide a direction for defense. For example, an effective defense is to infer the maximum perturbation ratio and to monitor the task execution to gather the actual execution time for comparison.

REFERENCES

Incentive mechanism for collective coordination in an urban intelligent transportation system using G-networks

Huibo Bi, Erol Gelenbe, Life Fellow, IEEE, and Yanyan Chen

Abstract—Although the abilities of human beings as participants in urban traffic, when they take decisions and interact with the transportation infrastructure and other vehicles, have been greatly amplified by powerful portable devices and efficient human-machine interfaces, the intelligence of vehicle drivers and pedestrians and their possible pro-social behaviour such as helpfulness and sense of duty, have been excluded in previous studies of Intelligent Transportation Systems (ITS). Thus the robustness of an ITS has not been evaluated as a function of the likelihood that participants follow instructions. Moreover, much effort has been dedicated to the use of Artificial Intelligence, while in fact many tasks can be easily accomplished by road users in the system who use ordinary human intelligence. Hence, in this paper, we propose a reward mechanism to integrate the intelligence of human road users into a large-scale transportation system to improve the effectiveness and robustness of the system by introducing a transportation-related task publishing system which is assisted by a queueing network model. The experimental results show that the use of a reward mechanism can significantly improve the performance of the transportation system in terms of average travel time of vehicles and the average response time to various tasks.

Index Terms—Intelligent Transportation Systems, Reward Mechanism, Traffic Management, Intelligence Augmentation, G-networks.

I. INTRODUCTION

The advancement of artificial intelligence in the recent decades has revolutionised the methodology and technique in urban traffic control, and has unveiled the emergence of various intelligent transportation systems. However, as a typical category of cyber-physical systems, the intelligence of participants has been, to some extent, excluded in the previous intelligent transportation systems, and the positive intrinsic and extrinsic characteristics of drivers, such as helpfulness and sense of duty, which have the potential to generate efficient cooperative collective behaviours, have not been motivated and considered as influence factors. Hence, in this paper, we propose a reward mechanism to encourage human intelligence augmented cooperative behaviours via rewarding credits to stimulate road users to carry out various types of tasks. The credits, which are expected to be fully exchangeable with currency, can also be used in exchange for better quality of services in the system or other authorised benefits. The transportation system is modelled with a queueing network based model to capture the dynamics of traffic flow under the effects of the reward mechanism. A gradient descent optimisation algorithm is employed to increase the social benefits and improve the traffic efficiency by solving the desired credit values via optimising the probabilistic choices of linked road segments at each task-publishing intersection with respect to historical observations on routine traffic. The remainder of the paper is organised as follows: we first review the related work in Section II, followed by the system model in Section III, which is composed of problem formulation in III-A, the system approximation model in III-B and the cost-benefit goal function in III-D. The simulation assumptions and results are introduced in Section IV. Finally, we draw conclusions in Section V.

II. RELATED WORK

A. Human-augmented Computing In Transportation

In the research field of transportation, human-augmented computing has mostly been used in collectively solving transportation related problems in a “crowdsourcing” manner, such as “crowd delivery” and “crowd sensing”. For instance, owing to the inaccurate or less detailed near-destination navigation provided by the popular digital map services, the research in [1] presents a “social navigation” algorithm to plan the last mile routes for drivers by connecting the top-ranked landmarks scored by experienced local drivers; in addition, to fight against the vulnerability of crowdsourcing systems to malicious attacks, a trajectory estimation approach is employed for location authentication by inferring the future possible locations of users, the users whose GPS points are frequently mismatched with the predictions will be considered as attackers and discarded. Similarly, the study in [2] developed a human augmented route recommendation system to provide routes for mobile phone users with the aid of experienced users; to facilitate crowd to use the system, a user-friendly human machine interface is designed to show candidate routes on a digital map where users can select by a single click; moreover, users are given various reward points in terms of their recommendation frequency and quality, the reward points can be used in exchange for a new recommendation request. On the other hand, the research in [3] has a special focus on the malicious bidding and free-riding behaviours (users are given rewards but pay no effect in executing sensing tasks) in crowd sensing; a three-step incentive mechanism is proposed to ensure the fairness of rewards and high quality of the sensed data during a crowd sensing process via service provider selection, service provider payment determination and sensory data quality evaluation; the incentive mechanism is mimicked...
and verified with a combination of a reverse auction and a Vickrey auction model.

However, to the best of our knowledge, although several research has integrated the use of human intelligence into ITS by taking advantage of the advanced visual perception and decision-making ability of human being. No related research has presented a comprehensive incentive based system framework to motivate participants to take part in the operations and decision-making of a transportation system, and analyse the influence of the introduction of the incentive mechanism.

B. Reward or Incentive Mechanisms In Transportation

Nowadays, although various point-based or badge-based reward systems [4] have been implemented to motivate participants or evoke perceptions of enjoyment in crowdsourcing systems such as crowdsourcing games [5] or file caching in delay tolerant networks [6], and the performance of these systems is proven to exceed the non-reward counterparts, only a few studies have been dedicated to the field of ITS. The use of reward mechanisms in the existing research of ITS is mostly reported in the crowdsourcing based transportation systems as a approach to motivate the share of experience and courses of cooperative action among users such as route-recommending and accident-monitoring. For example, the research in [7] presents a Blockchain based announcement network, namely “CreditCoin”, to motivate users to forward transportation information in a vehicular announcement network; in this system, each user is assigned with a credits account which contains reputation points; users can gain reputation points by relaying announcements or making an announcement such as reporting an accident. The work in [8] designs a traffic incident report system to identify various road situations by leveraging the reports from passing-by drivers; a monetary reward mechanism is introduced to improve the reliability and willingness of drivers to provide truthful reports; the experimental results show increased inference accuracy rates with the aid of a modified weighted majority voting and Bayesian inference approaches. Although several studies [9], [10] have proposed the use of credits to optimise the travel mode and pattern of users, most of them are investigated from an analytical point of view to evaluate the effects of the credit schemes rather than an online system implementation aspect. For instance, the research in [11] discusses possible tradable credit distribution and charging schemes for homogeneous users to reduce traffic demand; the credits are presumed to be issued and distributed to eligible users by the government and can be traded among users; users are charged with a specific amount of credits in terms of the used road segment; the research shows that an user equilibrium can always be reached under either fixed or elastic traffic demand.

In the research field of transportation, a popular trending that is similar to the use of incentive mechanisms is employing various rationing and pricing strategies to regulate the behaviours of participants. For instance, the study in [12] discusses the robustness of two pricing strategies used to direct traffic away from congested areas; the two pricing strategies are fixed tolling which simply charges users of each link with a fixed price and marginal-cost tolling which assigns each link with a flow-varying toll; the two strategies are evaluated under various scenarios including network structure changing, traffic rate changing and traffic demand changing; the simulation results show that the marginal-cost tolling strategy is much more robust than fixed tolling, but still lack robustness to heterogeneous users with different price sensitivity. The research in [13] analyses and compares the effectiveness of existing or potential traffic demand management policies, including rationing policies and road pricing policies; rationing policies such as vehicle ownership rationing and vehicle usage rationing are jointly analysed with the impact of road pricing policies; the numerical results show that (1) the vehicle usage rationing strategies always causes user welfare losses from a long-term perspective; (2) the road pricing strategy always generates more social welfare than rationing policies. In [14] a dynamic pricing system is presented to optimise the “Last-Mile” travel service by offering different prices for different types of passengers such as aged people, children or students; the last-mile service process is modelled as a batch arrival, batch service, multi-server queueing model and the optimal prices for passengers are determined by a constrained nonlinear optimization model. Similarly, the work in [15] proposes a dynamic pricing policy to mitigate congestion and offer reliable travel time for road users on managed lanes with multiple entrances and exits; a deterministic Markov decision model is employed to solve the optimal toll rate with the value function approximation approach; a lane choice model is proposed to choose routes for road users at each diverge point with the consideration of all possible routes; the experimental results show that the proposed model and related algorithm is efficient at discovering the optimal tolls.

However, although various rationing or pricing strategies have been proposed to optimise the traffic demands or traffic flows, most research has limited to pure traffic management rather than system management. In addition, the fairness of rationing and pricing strategies could cause disputes when charging heterogeneous users with different tolls.

III. SYSTEM MODEL

To introduce and motivate human intelligence and positive collective behaviours into current intelligent transportation systems, we design a novel reward mechanism that can improve social benefits and traffic efficiency by establishing a traffic task publishing system. In this section, we first describe and formulate the problem, then propose a comprehensive urban traffic network model to describe and capture the traffic dynamics under the influence of the reward mechanism. Finally, we use a optimisation model to calculate the desired credit for each task.

A. Problem Formulation

As a type of cyber-physical systems, state-of-the-art ITS commonly rely on heuristic algorithms to provide assistance to
participants, yet to a great extent, ignore the positive and negative influences of participants inside the system. For example, participants may benefit the performance and efficiency of the system by conducting coordinated collective behaviours such as monitoring and reporting unattended accidents, avoiding certain types of vehicles such as bicycles or coaches, and hitchhiking. On the other hand, participants may not follow or obey the instructions of the ITS, and as a result, may significantly reduce the performance of the ITS.

In this paper, we propose a reward mechanism to encourage the participants to conduct cooperative tasks by introducing a task publishing system. The cooperative tasks are categorised into 3 types: low level sensing tasks, high level cooperative tasks and simple route diversion tasks. For low level sensing tasks, since it is expensive and impossible to deploy sufficient sensors to cover the whole area of a large-scale urban traffic system, it would be helpful if road users are driven to monitor and report the unattended accidents or other dangerous behaviours. For high level cooperative tasks, it is obvious that the performance of a ITS will be increased if road users are prone to coordinate with others, such as avoiding specific vehicles or choosing a less congested route. In addition, road users are also encouraged to conduct prosocial behaviours to further optimise the performance of the system by allowing hitchhiking, pulling over at peak hours, etc. For the simple route diversion tasks, they are generated by the task-publishing system when there are not sufficient tasks to achieve the optimal route flow ratio. In the proposed mechanism, we reward different tasks with dynamic reward points to regulate collective behaviours and optimise the performance of the system with respect to diverse traffic situations and demands. Participants can sell their credits in exchange of money or to gain other authorised benefits.

B. G-network based System Approximation Model

Since our reward distribution framework requires certain traffic information to compute desired credit values for users, we assume an ITS system with sensing and computing facilities is pre-deployed in the targeted traffic network. The traffic network of the designated area is simplified as a directed graph consisting of nodes and edges. Nodes are intersections where vehicles can queue up and wait for the traffic signal progression, as well as receive credit scoring tasks from the reward distribution framework. Edges are road segments that link the intersections. Each intersection is considered as a queuing system of one single server with Poisson arrival users and exponentially distributed service time, and therefore the whole network can be modelled as a queuing network. The schematic diagram of a simplified road network, depicting the relations among joint intersections and the task publishing system, is shown in Figure 1. When a user approaches a task publishing intersection, rather than choosing the next edge based on each civilian’s own interests or the GPS instruction, certain credit scoring tasks can be sent to the users and incentivise users to choose a route to conduct cooperative tasks. To this end, we employ a G-network model to formulate the system dynamics and increase the likelihood for civilians to conduct cooperative tasks by providing appropriate credits for probabilistic choices of all available cooperative tasks to maximise social benefits and traffic efficiency.

G-networks [16] are a class of queueing network models that can describe real-world processes and practical scenarios with basic entities — positive customers, and additional control factors including negative customers [17], removals [18], triggers [19] and resets [20]. G-networks have been used in a wide range of applications, including describing the workload in computer systems [21], [22], realising energy efficiency in packet networks [23], as well as modelling energy systems [24], [25], populations of biological agents [26], gene regulatory networks [27] and emergency management [28], [29]. One useful property of the G-networks is the existence of a product form solution (PFS) [19], where the joint equilibrium distribution of the number of positive customers in the network can be derived.

The exact model we use to capture the dynamics of a routine traffic process is based on [19], and has only positive customers and triggers representing vehicles and re-routing decisions affected by credit scoring tasks, respectively. The positive customers (vehicles) that have just started to move from the stationary state will enter the network by joining their nearest intersections, and this “external” arrival of vehicles to $n_i$ occurs at an average rate of $\Lambda n_i$. The average service rate of vehicles at intersection $n_i$ is denoted by $\rho_{n_i}$ which depends on the physical characteristics of the node including the size and number of intersecting roads, as well as potential credit scoring tasks that may cost certain time to accept and handle. A vehicle which is leaving intersection $n_i$ will either head towards another connected intersection $n_j$ with probability $P_{n_i,n_j}$ or leave the network (reach the destination or pull over at the side of the road) with probability $d_{n_i}$. If we assume there are $N$ intersections in the network, then the routing choices of vehicles yield:

$$d_{n_i} + \sum_{j=1}^{N} P_{n_i,n_j} = 1 \quad (1)$$
Meanwhile, the road users may respond to reward gaining tasks, which are periodically published by the remote control center, to conduct cooperative tasks. The tasks are generated based on the requests of road users or the transportation system. The tasks reach users at an intersection $n_i$ at average rate $\lambda_n^+\eta_i$, instructing the surrounding vehicles to conduct certain cooperative tasks by moving to intersection $n_j$ with probability $Q_{n_i,n_j}$, where $\sum_{j=1}^N Q_{n_i,n_j} = 1$. This probability, which is associated with and affected by the potential credit, is a key parameter to be optimised in our system, as previous research [23] has indicated that it can significantly affect the performance.

With the aforementioned assumptions, the steady-state probability that an intersection $n_i$ has one or more vehicles is given by [19]:

$$q_{n_i} = \frac{\lambda^+_{n_i}}{r_{n_i} + \lambda^+_{n_i}}$$

(2)

where $\lambda^+_{n_i}$ is the total average arrival rate of vehicles to intersection $n_i$, including vehicles that were previously parked or at other intersections:

$$\lambda^+_{n_i} = \Lambda_{n_i} + \sum_{j=1}^N q_{n_j}r_{n_j,n_i} + \lambda_{n_i}Q_{n_j,n_i}$$

(3)

Notice that the quantities $q_{n_i}$ are coupled, and therefore (2) is a nonlinear equation that can be solved numerically. Term $r_{n_i}$ is the average service rate at intersection $n_i$ and it is the inverse of the average service time. The service time is defined as the time cost for a vehicle to traverse from one intersection to another; it is the summation of the time cost at the intersection and the time cost on the link.

\[
\Pr(K_{n_i} = k, i = 1, 2, \ldots, N) = \prod_{t=1}^N q_n^+(1 - q_{n_i})
\]

(4)

With this product form, we can readily analyise the current state of the system, especially the on-going analysis of vehicles within the system, by calculating $q_{n_i}$ at each intersection.

C. Point-based Reward Value Determination

The credits, which are the reward points used to incentivise the road users to conduct cooperative tasks during the transportation process, are modelled as the offsets between the “original status” of civilians to conduct cooperative tasks and the “optimised status” of civilians to conduct cooperative tasks from the system performance point of view. The “original status” of civilians is defined as the number of “willing-to-help” and probabilistic choices of civilians for various cooperative tasks without introducing the reward mechanism. On the other hand, the “optimised status” is considered as the potential number of “willing-to-help” civilians under the influence of the proposed reward mechanism and optimised probabilistic choices of civilians who would like to offer help in the optimal case with respect to the overall system performance such as average traffic delay or fuel utilisation.

To formulate this point-based reward mechanism clearly, firstly, we focus on a single intersection to investigate the relations among factors of interest, which can be extended and applied to a whole urban network; secondly, we build an optimisation model for the point-based reward value determination problem and solve it with a nonlinear programming approach.

Consider an intersection $n_i$ that is directly linked with other $m$ intersections $\{n_1, n_2, \ldots, n_m\}$, where $n_j$ refers to the intersection with identifier $j$. The set of credit scoring tasks that are published at each intersection is denoted by $B = \{b_1, b_2, \ldots, b_k\}$, where $b_k$ refers to the task type with identifier $k$, term $K$ represents the number of task types. Let term $N_{n_i}$ denotes the total number of road users who arrive at intersection $n_i$. Term $T^k_{n_i}$ represents the transfer rate between the total number of civilians who arrive at the intersection and the total number of civilians who are willing to conduct credit scoring tasks of type $b_k$ at the intersection. In other words, it represents the percentage of road users that conducts tasks of type $b_k$. Term $P^k_{n_i,n_j}$ denotes the percentage of tasks of type $b_k$ conducted by moving towards intersection $n_j$. In our treatment, $P^k_{n_i,n_j}$ is set to be directly proportional to the number of published tasks that can be conducted by moving towards intersection $n_j$. Hence, $Q^k_{n_i,n_j}$, which is aforementioned in subsection III-B as the probability for a vehicle to conduct certain credit scoring tasks by moving towards intersection $n_j$, can be expressed as:

Fig. 2. The G-network diagram of the intersection 1 shown in Figure 1.
\[ Q_{n_i,n_j} = \frac{\sum_{k=1}^{K} N_{n_i} T_{n_i}^k P_{n_i,n_j}^k}{N_{n_i}} \]

(5)

Hence, we build the relation between variables \( Q_{n_i,n_j} \) and \( T_{n_i}^k \). Therefore, \( T_{n_i}^k \) can be solved since \( Q_{n_i,n_j} \) can be determined by gradient descent optimisation. Term \( P_{n_i,n_j}^k \) can be obtained by on-site observation or set as empirical values based on the following equation:

\[ P_{n_i,n_j}^k = \frac{I_{n_i,n_j}^k}{\sum_{c=1}^{m} I_{n_i,n_c}^k} \]

(6)

where term \( I_{n_i,n_j}^k \) represents the number of published tasks that can be conducted by moving towards intersection \( n_j \). Term \( m \) denotes the number of intersections that are directly connected with intersection \( n_j \). Term \( m \) stands for the number of intersections that links to intersection \( n_j \).

The next step after determining \( T_{n_i}^k \) is to resolve \( \hat{R}_{n_i}^k \), which represents the point-based reward value for task type \( b_k \) at intersection \( n_i \). During the operation of the reward-scoring mechanism, certain reward points \( \hat{R}_{n_i} \), which are directly proportional to transfer rate \( T_{n_i}^k \) at intersection \( n_i \), are assigned to each intersection \( n_i \) to encourage civilians to offer help. This is reasonable as it implies that with the increase of the reward points, people are more and more likely to take part in. Form the task-publishing system point of view, we aim to minimise the cost of reward points to achieve the traffic management goal. The formulation of the problem is shown as follows:

\[
\min \sum_{k=1}^{K} \hat{N}_{n_i}^k \hat{R}_{n_i}^k \\
\text{subject to:} \\
\sum_{k=1}^{K} T_{n_i}^k P_{n_i,n_j}^k = Q_{n_i,n_j} \quad \forall Q_{n_i,n_j} \\
\hat{N}_{n_i}^k = T_{n_i}^k N_{n_i} \\
\hat{R}_{n_i}^k = \alpha_{b_k} T_{n_i}^k \\
\hat{N}_{n_i} \leq \sum_{c=1}^{m} I_{n_i,n_c}^k \\
\hat{R}_{n_i}^k \geq 0
\]

(7)

where term \( \hat{N}_{n_i}^k \) denotes the number of accepted tasks of type \( b_k \), and term \( \alpha_{b_k} \) denotes the relation between reward points \( \hat{R}_{n_i}^k \) and task transfer rate \( T_{n_i}^k \). The relation between \( \hat{R}_{n_i}^k \) and \( T_{n_i}^k \) is obtained by a questionnaire survey. In our treatment, for the sensing tasks \( b_1 \), cooperative tasks \( b_2 \), and simple route diversion tasks \( b_3 \), term \( \alpha_{b_1} \), \( \alpha_{b_2} \) and \( \alpha_{b_3} \) are set to 200, 166.7 and 71.4, respectively. It is also worthy to note that, in order to fulfill the above restrictions, the task-publishing system will have to release some simple route diversion task if there are not sufficient tasks to achieve the optimal route flow ratio \( Q_{n_i,n_j} \). From the implementation aspect, solutions can be obtained by using “CVXPY”, which is a Python-based modeling package for convex optimization problems.

### D. Cost-benefit Estimation Model

In this section, we mainly introduce the cost-benefit estimation model, or in other words, the goal function (objective function), to calculate the optimal route flow ratio for users to conduct credit scoring tasks at each intersection. The performance we aim to optimise are twofold: traffic efficiency and social benefits. The traffic efficiency is defined as the total delay for a vehicle to experience in the network. The social benefits are simplified as the cost for a task request to receive respond inside the network.

For a vehicle, the total delay in the urban network includes delays at both intersections and road segments, and depends on the congestion level. Since we have modelled the road network as a queueing network, the average number of vehicles at a queue can be derived directly from (4) yielding:

\[ N_{n_i} = \frac{q_{n_i}}{1 - q_{n_i}} \]

(8)

Using Little’s formula, the average traversal times are given by:

\[ D_{n_i} = \frac{N_{n_i}}{\lambda_{n_i}} \]

(9)

while the total average delay experienced by a vehicle in the network is:

\[ D_t = \frac{\sum_{i=1}^{N} N_{n_i}}{\sum_{i=1}^{N} \lambda_{n_i}} \]

(10)

where the numerator is the total average number of vehicles in the network, and denominator is the total rate at which vehicles join the network.

On the other hand, the cost for a task to be accepted is affected by its occurrence position, type of the task request, arrival rate of this type of task request, number of vehicles at the intersection, percentage of vehicles that is willing to conduct this type of task:

\[ P_k = \sum_{i=1}^{N} \sum_{k=1}^{K} \frac{P_{n_i}^k q_{n_i} T_{n_i}^k}{1 - q_{n_i}} \]

(11)

where \( P_{n_i}^k \) is the probability for a task request of type \( b_k \) to appear at the intersection \( n_i \), which are defined in (12).

\[ P_{n_i}^k = \frac{\lambda_{n_i}^k}{\sum_{i=1}^{N} \sum_{k=1}^{K} \lambda_{n_i}^k} \]

(12)

where term \( \lambda_{n_i}^k \) is the arrival rate for a request of type \( b_k \) to reach the network from node \( n_i \), which can be calculated by statistical approaches.
To achieve social benefits with an acceptable network latency, we combine the two metrics above and the goal function can be expressed as:

$$G_c = D_t + \frac{\epsilon}{P_s}$$  \hspace{1cm} (13)

where $\epsilon$ is a constant that coordinates the relative importance between the traffic delay and social benefits. In our case, $\epsilon$ is set to 20. The goal function can be solved by using gradient descent optimisation.

IV. SIMULATIONS AND RESULTS

We employ a Python based simulation tool, namely the smart environment simulator (SES), to evaluate the performance of our proposed reward mechanism. The SES works as a client to dynamically interact with the open-source simulation platform “Simulation of Urban MObility” (SUMO) [30], which is a microscopic and continuous road traffic simulator as shown in Figure 3. We utilise a 3-days period (2015/08/13 - 2015/08/15) realistic vehicle trip data collected in Beijing City as the input of the SES. Each day is divided into 12 consecutive 2-hour periods to accelerate the experiments by conducting multiply simulations in a parallel manner.

![Fig. 3. The graphical user interface of the SUMO-based simulator.](image)

As a preliminary step of the experiments, the raw vehicle trip data are pre-processed in the following 3 steps: first, the vehicles that traversed the designated area are extracted from the raw data; second, the GPS traces of the trips are transformed into the X,Y-coordinates in the SUMO platform and then are linked into integrated routes; third, the processed routes as well as the origin, destination, and the start time of the trips are added into the event engine of the SES in chronological order. When the simulation starts, the trips will be replayed unless a vehicle accepting a re-routing command.

In the experiments, we use 2 scenarios to validate the effectiveness of the proposed mechanism. The first scenario, which is used for the comparison propose, is the original courses of action of the vehicles in the road network. The second scenario is the courses of action of the vehicles under the impact of the proposed reward mechanism.

As can be seen in Figure 4 to Figure 6, the average travel time of vehicles decreases remarkably with the use of the reward mechanism. This is mainly because the reward mechanism encourages the vehicles to choose alternative paths to destinations and avoid the congested original paths. Meanwhile, to finish the reward-scoring tasks, the vehicles are naturally more distributed in the network and are likely to increase the occupancy rates of less popular routes. This can be also proved by the Figure 7 and Figure 8, which show the average number of traversed vehicles for each edge in the network during the 2-hours periods. As it is clearly shown, the use of the reward mechanism can efficiently balance the average visiting times of among all edges. The visiting times of the most visited edge is reduced from 496 to 410.

![Fig. 4. The average travel time of vehicles in 2-hour periods during 2015/08/13.](image)

On the other hand, the average response time for a score-rewarding task to be accepted in the network is shown in Figure 9. It is affected by the number of vehicles at a task-publishing intersection, the type of the task and the reward value of the task. As it is shown in the figure, most of time periods of a day (from 6 am to 24 pm), the average response time of a task is less than 500 seconds. However, the average response time exceeds 500 seconds during 0 am to 6 am. This is mainly because during wee hours there are not sufficient vehicles to conduct tasks. It is also worth noting that day 2015/08/15 possesses the lowest average response time of a task among the three days. This is because it has the largest average amount of vehicles in the targeted area, and therefore has more candidates to conduct tasks. Similarly, day...
2015/08/13 possesses the highest average response time due to the fact that least vehicles traversed the area during the day.

The reward points cost by the task publishing system are shown in Figure 10. The pre-set exchange ratio between RENMINBI and the reward point is 1:1. As can be seen clearly in the figure, day 2015/08/13 consumes the lowest reward points while day 2015/08/15 consumes the highest reward points. This is because day 2015/08/15 has the largest amount of vehicles in the targeted area, and therefore the task publishing system has to publish more tasks to maintain the optimal route flow ratio. Regarding the cost, in the worst case (2015/08/15), the task publishing system consumes about 410,000 yuan per day to guide the vehicles in order to achieve overall system optimization. The annual cost would be around 149 million yuan, which is much less than the annual financial loss caused by congestion in Beijing, which is reportedly around 173.5 billion yuan [31], [32]. Moreover, the system can use certain vouchers or other authorized benefits to reduce the monetary cost.

V. CONCLUSIONS AND FUTURE WORK

In this paper, we propose a reward mechanism to motivate and increase the possibility for involved civilians in the system to use their intelligence, and therefore forming a human intelligence augmented transportation system to improve the sensing, cooperation and pro-social behaviours in the routine transportation. With the build of a task publishing system on top of the existing ITS, the collective behaviours of participants are regulated and optimised by carrying out different
types of tasks. The simulation results show the average travel time of vehicles within the ITS can be significantly reduced with the aid of the reward mechanism in comparison with the non-reward counterpart. Meanwhile, the use of the reward mechanism can encourage civilians to report the dangerous situations which are difficult to be identified for the traditional mechanism. Furthermore, we evaluate the cost of the task publishing system and the simulation results show that the monetary cost is only the 0.08% of the annual economic loss caused by congestion in Beijing City.

REFERENCES


A Priority-based Dynamic Link Scheduling Algorithm Using Multi-criteria Decision Making in Wireless Body Area Networks

Beom-Su Kim
Dept. of Computer Science and Engineering
Chungnam National University
Daejeon, Korea
bumsou10@cnu.ac.kr

Ki-Il Kim*
Dept. of Computer Science and Engineering
Chungnam National University
Daejeon, Korea
*kikim@cnu.ac.kr

Abstract—IEEE 802.15.6, which is the standard for wireless body area networks (WBANs), defines a unique link allocation interval called the managed access phase to meet the QoS requirements of medical applications. The hub grants an uplink allocation interval to the node according to predetermined scheduling policies; however, the IEEE 802.15.6 standard does not consider user priorities in the uplink scheduling. To address this problem, many priority-based link scheduling algorithms have been proposed; however, these algorithms ignore the unique characteristics of WBANs, such as changes in body posture and damage to body tissue, thereby leading to a waste of resources. To overcome these limitations, in this study, we propose three scheduling parameters that are suitable for WBANs, and logically determine the relationship between the proposed parameters using the multi-criteria decision making (MCDM) method. Furthermore, a priority-based dynamic link scheduling algorithm is proposed, which is based on the results of the MCDM. The simulation results prove that our proposed approach can balance the resource allocation, while maintaining the performance of high-priority nodes in terms of throughput, delay, and power consumption.

Index Terms—wireless body area networks (WBAN), dynamic link scheduling, multi-criteria decision making (MCDM)

I. INTRODUCTION

IEEE 802.15.6 [1] is a standard for wireless body area networks (WBANs). WBANs obtain vital signs of patients using low-power and miniaturized medical sensors, and these data are forwarded to physicians for remote health monitoring service. The main purpose of the IEEE 802.15.6 standard is to enable the efficient use of resources owing to the limited sensor size and quality of service (QoS) support for emergency traffic [2], [3]. To achieve these objectives, the IEEE 802.15.6 MAC defines a unique link allocation interval called the managed access phase (MAP) in the superframe. The hub grants an uplink allocation interval to a node according to an appropriate scheduling policy such as a first-in-first-service (FIFS) basis or round-robin (RR). However, existing link scheduling algorithms cannot support heterogeneous traffic flow because user priorities are not considered in the scheduling policy. Thus, it is essential to develop priority-based link scheduling schemes for intra-WBAN communications because the loss or excessive delay of important data acquired by the medical sensor can cause fatal accidents.

To address this problem, many priority-aware link scheduling algorithms have been proposed [4]–[9]. The authors [4] propose a superframe-based time division multiple access protocol for WBANs. The proposed scheme is based on a time slot allocation, which considers the heterogeneity of nodes and delay limitations. The hub sets the priority of each node and decides the length and order of slots that are allocated. The authors [5] formulate an algorithm for the priority-based allocation of time slots that considers a fitness parameter characterizing the importance of the data, energy consumption rate when transmitting the packet, and other crucial properties for WBANs. In [6], the authors design the unified framework of an energy efficient resource allocation mechanism, where both the requirements of QoS and the characteristics of dynamic links are considered. In addition, a transmission rate allocation policy is proposed to adaptively adjust the transmission rate at each sensor. The authors [7] propose a priority-weighted RR scheduling strategy that aims to support QoS for emergency traffic. The proposed algorithm is a combination of a priority scheduling and a weighted RR that employs the user priorities of physiological data to determine how to schedule and send them off the WBAN. In [8], a link quality-aware resource allocation scheme is proposed. To maintain the QoS of the emergency traffic, the proposed algorithm predicts correlations among different aspects of link quality. Based on the available correlated link qualities, the subchannel allocation phase divides the available bandwidth into several subchannels. In [9], an energy efficient protocol for routing and scheduling is proposed by considering the node energy, path loss, traffic type, and other relevant factors for WBANs. In addition, the authors propose an adaptive slot assignment method that is designed to meet the requirements of data rate and priority in the slot assignment phase.

As previously reviewed, several priority-aware link scheduling algorithms for WBANs have been proposed to save energy and QoS support; however, WBANs have the following addi-
The link scheduling for intra-WBAN communications is modeled using the MCDM method along with multiple criteria such as user priority, body posture, temperature rise, and energy saving. The link scheduling scheme needs to handle aspects such as user priority, body posture, temperature rise, and energy saving. However, these challenges have not been completely addressed in most existing studies. For example, dynamic link scheduling algorithms that consider current channel conditions have already been proposed [10], [11]; however, the main purpose of the algorithms is to mitigate channel interference between neighboring BANs. Thus, the primary concern is that the network performance is affected by the frequent body movements. In such an environment, channel interference may occur due to overlapping transmission ranges of nearby WBANs. In addition, temperature-aware routing protocols have been proposed [12], [13] to address the overall temperature rise problem of WBANs. In the algorithm, each node forwards the packet to the neighboring node with the lowest temperature until the packet arrives at the destination. However, the temperature rise problem has not yet been considered in the resource allocation of the MAC protocol.

Technically, it is challenging to implement such an intra-WBAN resource allocation that reflects all the requirements that exist in WBANs because it is difficult to define the relationship between multiple criteria. In other words, we need a logical approach to accommodate multiple criteria. In this paper, we propose a priority-based dynamic link allocation algorithm using the multi-criteria decision making (MCDM) method to reduce the data loss rate due to variations in the body posture and the damage to body tissue, as well as to improve the energy efficiency of WBANs while designing their MAC protocol.

The main contributions of our paper are as follows.

- Several scheduling parameters for WBANs are formulated to reduce the data loss rate due to variations in the body postures and the damage to body tissue and to improve the energy efficiency of WBAN.
- After defining the relationship and priority between multiple criteria, a priority-based dynamic link scheduling algorithm is proposed, which is based on the outcome of the MCDM method.
- Unlike previous studies, we evaluate the performance of the proposed scheduling algorithm [14] by thoroughly following the association procedures and frame transactions that are specified in the IEEE 802.15.6. This improves the reliability of our experimental study.

II. SYSTEM MODEL

In this section, we first introduce the intra-WBAN network model and describe the resource allocation scheme specified in the IEEE 802.15.6. Subsequently, scheduling parameters used for resource allocation in WBANs are formulated.

A. Network Model

The IEEE 802.15.6 stipulated that a BAN includes a hub and a range of nodes between 1 and 64. As shown in Figure 1, a hub and nodes are interconnected in a one-hop star topology. Each node collects different medical data and forwards them to the hub according to the priority of the data. We adopt beacon mode with superframes as an access mode and set a scenario where the hub broadcasts a beacon frame in every beacon period. Table I shows the important physiological parameters that can be monitored using WBANs. To support various types of data, the hub provides exclusive access phase (EAP), random access phase (RAP), and MAP in the superframe. The EAP is a random access phase to provide differentiated QoS for high-priority traffic. In RAP, all types of traffic can acquire access channels through a fair contention. The MAP used to arrange the scheduled uplink/downlink allocation intervals. Consequently, the node with a higher priority may have a greater advantage in receiving more dedicated time slots or gaining an access channel through a priority-based random access phase.

<table>
<thead>
<tr>
<th>Sensor type</th>
<th>Priority (0 to 7)</th>
<th>Data rate</th>
<th>bandwidth</th>
</tr>
</thead>
<tbody>
<tr>
<td>EMG</td>
<td>6 - 7</td>
<td>320 kbps</td>
<td>0 - 10000 Hz</td>
</tr>
<tr>
<td>ECG</td>
<td>6 - 7</td>
<td>288 kbps</td>
<td>100 - 1000 Hz</td>
</tr>
<tr>
<td>EEG</td>
<td>6 - 7</td>
<td>42.3 kbps</td>
<td>0 - 150 Hz</td>
</tr>
<tr>
<td>Glucose</td>
<td>4 - 5</td>
<td>1600 bps</td>
<td>0 - 50 Hz</td>
</tr>
<tr>
<td>Temperature</td>
<td>0 - 3</td>
<td>120 bps</td>
<td>0 - 1 Hz</td>
</tr>
<tr>
<td>Cochlear implant</td>
<td>0 - 3</td>
<td>100 kbps</td>
<td>-</td>
</tr>
<tr>
<td>Voice</td>
<td>0 - 3</td>
<td>50 - 100 kbps</td>
<td>-</td>
</tr>
</tbody>
</table>

B. Connection-oriented Resource Allocation

To obtain scheduled uplink allocation intervals in the MAP, an association should be established between a node and a hub. As shown in Figure 2, all of the unconnected nodes send a connection request frame during random access periods using
carrier sense multiple access/collision avoidance (CSMA/CA) after receiving the beacon frame. The connection request frame contains a link request IE field. The link request IE includes an element ID, length, and allocation request field. The element ID field is used to identify the type of link request. The length field is used to specify the total bytes of the allocation request. In order to request the uplink allocation interval in MAP, a node with high priority must specify the number of time slots that can satisfy the QoS requirements in the allocation request field. To approve an uplink allocation request, the hub must send a connection assignment frame containing the link assignment IE. To initialize the link assignment IE, the hub needs to apply an appropriate link scheduling algorithm according to the link request IE of the connection request frame. However, the IEEE 802.15.6 standard did not deal with a specific link scheduling algorithm, and implied that the hub would sequentially approve the received uplink allocation request.

C. Scheduling Parameters

We propose three scheduling parameters to reduce the data loss rate due to variations in the body posture and the damage to body tissue as well as to improve the energy efficiency of WBANs. These parameters help to make a more flexible decision in the proposed scheduling algorithm.

1) Packet Error Rate (PER): The WBAN topology can change over time due to the posture or movement of the body. Frequent body movement and posture changes can cause channel fading, which significantly degrades network performance in terms of energy consumption, throughput and delay. The purpose of the proposed scheduling is to prevent the allocation of resources to nodes with low link quality (i.e., nodes with a high probability of transmission failure) rather than to allocate resources to nodes with a high link quality preferentially. In WBANs, the propagation loss model must calculate the receiving (RX) power based on the location of the human body the node is attached to and the output power of the transmitter. We set the average channel gain (dB), as shown in Table II [15].

<table>
<thead>
<tr>
<th>Node Pairs</th>
<th>Gain (dB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belt to head</td>
<td>-48.1</td>
</tr>
<tr>
<td>Wrist to head</td>
<td>-53.8</td>
</tr>
<tr>
<td>Wrist to belt</td>
<td>-30.0</td>
</tr>
</tbody>
</table>

After estimating the RX power using the propagation loss model, the receiver can calculate the signal-to-interference plus noise ratio (SINR). The SINR and RX power are used to calculate the bit error rate (BER) and packet error rate (PER). The BER between the transmitter $i$ and receiver $j$ can be computed using the SINR and the transmission parameters of the physical layers (PHY) ([15]).

$$BER_{ij}^{t} = \begin{cases} 0.5 * e^{-Eb/No} \\ O(\sqrt{4Eb/No} * \sin(\sqrt{2} * \pi/4)) \end{cases}$$ (1)

where $Eb/No$ is the energy per bit-to-noise power spectral density ratio in dBm, which is computed as [15],

$$Eb/No[dB] = SINR_{ij}^{t}[dB] + 10 * \log_{10}(BW/R)$$ (2)

where $BW$ represents the bandwidth and $R$ is the symbol rate. Finally, the PER can be calculated as [15],

$$PER_{ij} = 1 - (1 - BER_{ij}^{t})^{n}$$ (3)

where $n$ represents the frame length in bits.

2) Estimated Temperature ($T$): Considerable amounts of RF energy may increase body temperatures and damage body tissues, particularly to the testicles and eyes, which are more vulnerable because these areas have less blood flow to dissipate heat when compared with other parts of our bodies. The specific absorption rate (SAR) is a measure of the rate at which body tissues absorb energy when the body is exposed to a radio frequency (RF) radiation. The SAR can be estimated as follows:
where \( \sigma \) denotes the electrical conductivity, \( E \) denotes the induced electrical field, and \( \rho \) represents the tissue density. Then, we can estimate body temperature using the SAR and the Pennes’ bio heat equation as follows:

\[
\rho c \frac{\partial T}{\partial t} = \Delta(k \Delta T) - \rho_b w_b c_b (T - T_a) + \rho SAR + P_c
\]

where \( c \) is the specific heat of the tissue, \((\alpha T/\alpha t)\) specifies the rate of change of the tissue temperature \( (T) \) over time \( t \). \( \Delta \) is the vector gradient operator and \( k \) is the thermal conductivity. \( T_a \) is the arterial blood temperature, \( \rho_b \) is the blood density, and \( w_b \) and \( c_b \) are respectively the perfusion rate of blood and specific heat of blood. \( P_c \) represents the absorbed power density. The specific values for each parameter are given in [16].

3) Average Power Consumption (\( E \)): The amount of energy consumed by a node is an essential parameter that should be considered in the scheduling algorithm to increase the overall network lifetime. To calculate the energy consumption of the PHY, we estimate the power consumption according to transceiver states such as Tx, Rx, Idle, and Sleep in the PHY. The average power consumption (\( E \)) can be calculated as follows:

\[
E = (\alpha - 1) T_{cu} * Idle_e + (\alpha) L * Tx_e + (\alpha * D) Rx_e + Sleep_e
\]

where \( T_{cu} \) represents the cumulative delay caused by the backoff time for a frame that was successfully transmitted, or the frame retransmission time, which includes the inter-frame space time. This means that the node has to wait a certain period between pSIFS and pExtraIFS before retransmitting the frame. The \( \alpha \) represents the frame retransmissions or number of backoff stages, and \( L \) is the payload length. \( Idle_e, Tx_e, Rx_e, \) and \( Sleep_e \) represent the energy consumption for each transceiver state.

III. Decision Making Procedure for Scheduling Order Based on MCDM Method

The purpose of using the MCDM method is to logically define the relationship and priorities between scheduling parameters, and then to calculate the scheduling order. We use the analytic hierarchy process (AHP) to define the relationship between the proposed parameters. The AHP requires pairwise comparisons to determine the relative importance of the criteria. The results of pairwise comparisons are expressed as a square matrix \( X_{ij} \). The \( X_{ij} \) represents the relative importance of factor \( i \) to \( j \). The following example is given for ease of understanding.

The pairwise comparison matrix is defined by a decision maker, who needs to determine the relative importance of the criteria based on network conditions. After defining the pairwise comparison matrix, we can calculate the priority vector, which is expressed as a normalized eigen vector of the matrix. To obtain the eigen vector \( v \), the pairwise comparison matrix needs to be normalized as

\[
X_{ij} = \frac{X_{ij}}{\sum_{i=1}^{n} X_{ij}}
\]

After normalization is complete, we compute the eigen vector of the factor \( i \) as

\[
v_i = \frac{\sum_{j=1}^{n} X_{ij} v_j}{n}
\]

where \( v_i \) represents the relative priority of \( i \) among the criteria. Then, we obtain the following results.

\[
vX_{ij} = \begin{pmatrix} 0.160 \\ 0.251 \\ 0.589 \end{pmatrix}
\]

That is, \( E \) plays a major role among the criteria. However, \( PER \) has the lowest importance. After we obtain the weights of each criterion, the next step is to synthesize the values of the parameters using the weighted sum equation. To obtain the final score of each alternative, we adopt the simple additive weighting (SAW) method. The purpose of the SAW method is to find the weighted sum of the performance ratings on each alternative. To achieve a high rating, decision makers must evaluate whether each criterion should be minimized or maximized. In this study, \( PER \) and \( T \) are normalized as

\[
n_{ij} = \frac{\min(X_{ij})}{X_{ij}}
\]

where \( n_{ij} \) is the normalized value related to criterion \( X \) corresponding to row \( i \), and column \( j \) in the neighboring table (i.e., the hub should maintain a table of parameter information for each node). The \( \min(X_{ij}) \) represents the minimum value of \( X_{ij} \). \( E \) is normalized as,

\[
n_{ij} = \frac{X_{ij}}{\max(X_{ij})}
\]

where \( \max(X_{ij}) \) represents the maximum value of \( X_{ij} \). Then, we can obtain the weighted normalization value as follows.

\[
W_{ij} = n_{ij} * v_i
\]
where $W_{ij}$ denotes the weighted normalization value, $n_{i,j}$ is the normalized value corresponding to row $i$, and column $j$, $v_i$ is the relative weight of the criterion $i$. Based on the sum of weighted normalization values for each parameter, we can determine the scheduling order for each node.

IV. PRIORITY-BASED DYNAMIC LINK SCHEDULING ALGORITHM

As illustrated in Figure 2, each node should request the number of time slots when they start an association with the hub. The requested slots are assigned to the scheduled uplink interval of the MAP, and the hub transmits the assignment information to the node by including the link assignment IE (i.e., interval start and interval end field) in the connection assignment frame. The allocation slot length requested by a node is set to $p\text{AllocationSlotMin} + L \times p\text{AllocationSlotResolution}$. Note that $p\text{AllocationSlotMin}$ and $p\text{AllocationSlotResolution}$ are set to 500 us. The requested allocation slot ($L$) is calculated as follows:

$$L = \frac{T_f}{p\text{AllocationSlotResolution}} \quad (14)$$

where $T_f$ represents the time required to send a management or data frame, and is calculated as

$$T_f = T_s \times (N_{\text{preamble}} + N_{\text{header}} + S_{\text{header}} + \frac{N_{\text{total}}}{\log_2(M)} \times S_{\text{PSDU}}) \quad (15)$$

where $N_{\text{preamble}}$, $N_{\text{header}}$, and $N_{\text{total}}$ are the length of the preamble, physical layer convergence protocol (PLCP) header, and PLCP service data unit (PSDU), respectively. $T_s$ and $M$ are the symbol rate and cardinality of the constellation of a given modulation scheme, respectively. In addition, $S_{\text{header}}$ and $S_{\text{PSDU}}$ respectively refer to the spreading factor of the PLCP header and PSDU.

Algorithm 1 Priority-based dynamic link scheduling

**Input:** Requested allocation slot $L \geq 0$, packet error rate PER, estimated temperature $T$, average power consumption $E$.

**Output:** Scheduling queue $S$, element type is $pair(L,W)$.

**Initialization :**
1. $Th_{\text{PER}}, Th_T, Th_E$ : Threshold for each parameter
2. $G1, G2, G3$ : Priority queue for grouping
3. $W$ : Weighted normalization value
4. $UP$ : User priority (0 to 7)
5. $Slot_{\text{max}}$ : Maximum number of uplink allocation slots

**Scheduling :**
6. for $i = 1$ to $n$ do
7. $W_i = MCDM(PE_{ri}, T_i, E_i)$
8. if $UP_i \geq UP_k$ then
9. Push $pair(L_i, W_i)$ to $G1$
10. else if $UP_i \geq UP_k$ then
11. Push $pair(L_i, W_i)$ to $G2$
12. else
13. Push $pair(L_i, W_i)$ to $G3$
14. end if
15. end for
16. if $G1$.size + $G2$.size + $G3$.size $\leq Slot_{\text{max}}$ then
17. Push $G1, G2, G3$ to $S$
18. return $S$
19. else
20. Goto Optimization
21. end if

**Optimization :**
22. while $G1$.size + $G2$.size + $G3$.size $>$ $Slot_{\text{max}}$ do
23. if $G3$ $\neq$ empty then
24. Remove an element at the end of $G3$
25. else
26. Remove all elements exceeding $Th_{\text{PER}}, Th_T, Th_E$ from $G1, G2, G3$
27. if $G1$.size + $G2$.size + $G3$.size $>$ $Slot_{\text{max}}$ then
28. Pop an element $pair(L, W)$ at the end of $G2, G1$
29. if $L \neq 0$ then
30. Push $pair(L - 1, W)$ to $G2, G1$
31. end if
32. end if
33. end if
34. end while
35. Add $G1, G2, G3$ to $S$
36. return $S$

First, the hub divides each node into three groups according to their priority (i.e., the priorities of Group 1, Group 2, and Group 3 are 6-7, 4-5, and 0-3, respectively). Next, the hub arranges the slots requested by the nodes based on the scheduling order obtained through the MCDM method for each group. If the total allocation slot length requested by the node is larger than the uplink allocation interval of the

Fig. 3: Optimization process for uplink scheduling.

However, according to the IEEE 802.15.6 standard, the superframe period cannot exceed 255 ms, i.e., the maximum length of the MAP excluding the random access period is set to 175 ms. To efficiently distribute limited resources to the node, we propose a priority-based dynamic link scheduling algorithm using the results of the MCDM method.
MAP, then three optimization steps are performed for uplink scheduling, as shown in Figure 3. The first step is to remove the allocation slots from Group 3 until the total allocation slot length fits into the uplink allocation interval (STEP 1). The second step is to remove all slots assigned to nodes whose temperature ($T$), energy level ($E$), and PER exceed the threshold for each group. If the total allocation slot length is still larger than the uplink allocation interval after the previous two-step optimization, then one slot ($L$) with the lowest scheduling order is removed for each group in the order of lower group priority until the length limit is satisfied (STEP 3). The scheduling order is dynamically changed according to the proposed three parameter values and the scales of the pairwise comparison matrix. The allocation slots for each node are recalculated by the hub in each beacon period and unicast to the node in the downlink interval of the MAP in the form of the connection assignment frame. Algorithm 1 returns the scheduling queue (S) containing the allocation slot information for each node (i.e., interval start and interval end) when the number of requested allocation slots ($L$) and three parameter values are given to the hub.

V. PERFORMANCE EVALUATION

A. Simulation Setting

To perform the simulations, we choose the NS-3 network simulator version 3.29 among possible platforms [17]. The network models and simulation parameters used for the simulation are shown in Table III.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PHY</td>
<td>Narrowband PHY</td>
</tr>
<tr>
<td>Frequency band</td>
<td>2400-2483.5 MHz</td>
</tr>
<tr>
<td>Data rate</td>
<td>971.4 kbps</td>
</tr>
<tr>
<td>Access mode</td>
<td>Beacon with superframes</td>
</tr>
<tr>
<td>Random access mechanism</td>
<td>CSMA/CA</td>
</tr>
<tr>
<td>Mobility</td>
<td>walking, run, stand, sitting</td>
</tr>
<tr>
<td>Number of BANs</td>
<td>1</td>
</tr>
<tr>
<td>Number of hubs</td>
<td>1</td>
</tr>
<tr>
<td>Number of nodes</td>
<td>15</td>
</tr>
<tr>
<td>Payload size</td>
<td>0 to 255 bytes</td>
</tr>
<tr>
<td>$T*<em>{\text{power}}/RX</em>{\text{sensitivity}}$</td>
<td>-15 dBm/83 dBm</td>
</tr>
<tr>
<td>RX/Idle/Sleep</td>
<td>18 mA/0.4 mA/0.03 mA</td>
</tr>
<tr>
<td>$p_{\text{AllocationSlotMin}}$</td>
<td>500 ns</td>
</tr>
<tr>
<td>$p_{\text{AllocationSlotResolution}}$</td>
<td>500 us</td>
</tr>
<tr>
<td>Beacon length</td>
<td>80 to 255 ms</td>
</tr>
<tr>
<td>EAP1/RAP1/EAP2/RAP2</td>
<td>20 ms/20 ms/20 ms/20 ms/20 ms</td>
</tr>
<tr>
<td>MAP length</td>
<td>0 to 175 ms</td>
</tr>
<tr>
<td>$p_{\text{ExtraFpsMIFS/pMIFS}}$</td>
<td>10 us/20 us/75 us</td>
</tr>
<tr>
<td>$m_{\text{Timeout}}$</td>
<td>30 us</td>
</tr>
</tbody>
</table>

As shown in Table I, the IEEE 802.15.6 specified that the sensor types are classified into eight levels based on the user priority. We use the priority levels as a backoff bound for CSMA/CA. The supported frequency bands follow narrow band PHY. The hub and nodes are interconnected in a one-hop star topology. The hub provides three types of access phases in the superframe, such as EAP, RAP, and MAP. The MAP contains the scheduled uplink/downlink allocation intervals, as well as unscheduled bi-link allocation intervals. However, we only consider the scheduled uplink allocation interval, excluding other allocation intervals. The Priority group is divided into three types according to user priority, and the number of nodes constituting each group is set to five. The number of requested allocation slots ($L$) is determined according to the payload size, and performances are evaluated for one representative node of each group. In this experiment, we compare the performance according to three test cases to analyze how the proposed scheduling algorithm affects the throughput, delay, and power consumption. We performed the simulation 100 times with 95% confidence interval.

1) Test Case 1: If the body temperature around a node exceeds 43 °C [18], it is recognized as a hotspot and the node turns off its transceiver. In other words, it is important to avoid temperature rises to increase the performance in normal situations. In this case, we assume that the node rarely moves (i.e., body postures such as standing and sitting). We use this test case to analyze whether a node can adaptively stop resource allocation before reaching the temperature threshold. Based on this fact, the pairwise comparison matrix can be set as follows.

\[
\begin{array}{ccc}
\text{PER} & T & E \\
1 & 1/4 & 1/2 \\
4 & 1 & 3 \\
\end{array}
\]

(16)

2) Test Case 2: In this case, we assume that the node moves frequently (i.e., body movements such as walking and running). In other words, we analyze whether a node can stop allocating resources to nodes that are likely to fail transmissions to increase the overall throughput and prevent resource leaks. Based on this, we can determine the pairwise comparison matrix as follows.

\[
\begin{array}{ccc}
\text{PER} & T & E \\
1 & 3 & 3 \\
4 & 1/3 & 1/2 \\
\end{array}
\]

(17)

3) Test Case 3: In this case, we assume that the movement of the body is stationary, and we set the relative importance of the temperature ($T$) and average power consumption ($E$) to be equal to each other. By applying the following pairwise comparison matrix to the algorithm, we expect a steady decrease in the number of slots allocated to nodes with low energy levels.

\[
\begin{array}{ccc}
\text{PER} & T & E \\
1 & 1/2 & 1/3 \\
2 & 1 & 1 \\
\end{array}
\]

(18)
B. Results and Discussion

We compared the proposed scheduling algorithm with a priority-based FIFS algorithm to verify that the proposed algorithm operates adaptively according to parameter values. The priority-based FIFS approach guarantees link allocation to high-priority nodes, and reserved link information is not changed after the initial association is established with the hub. However, the proposed algorithm can maintain the balance of resource allocation because it determines the scheduling order according to the scheduling parameter values and MCDM results, and it does not allocate resources to nodes with low scheduling orders. In addition, the proposed algorithm is capable of adaptive scheduling according to network conditions because the changed link information is transmitted to the node through the scheduled downlink interval in the MAP in each beacon period.

Based on these facts, from Figure 4 (a), (b), and (c), we can see that the proposed algorithm reduces the power consumption while maintaining the performance of nodes with high priority. As the payload size increases, the node needs to transmit more frames, which increases the temperature around the node. As described earlier, when the temperature exceeds an acceptable threshold, the node should turn off its transceiver. To prevent a performance decrease resulting from a temperature rise, it is necessary to force the node to transmit frames in random access phases rather than the scheduled uplink allocation interval of the MAP for a period of time. In this test case, the relative importance of temperature ($T$) was set to be the highest, so the sum of the weighted normalization value will be low if the temperature has a higher value than other scheduling parameter values. If a node with a higher priority (i.e., UP7) is assigned slots in the uplink interval first, after which its temperature rises, the node is pushed to the end of the scheduling queue and the number of allocated slots ($L$) will be reduced. This operation increases the efficiency of the resource usage because it allows a node with a low temperature to be assigned slots in the uplink interval of the MAP. However, because the priority-based FIFS scheduling algorithm allocates resources to nodes having a high priority regardless of the temperature of the node, the efficiency of the resource usage is significantly reduced as the payload size increases. In particular, a resource leak may occur owing to continuous resource allocation to a node that turned off its transceiver, thereby increasing the power consumption substantially and reducing the overall network lifetime.

In test case 2, the body posture is frequently changed; therefore, the overall throughput is reduced and the delay is increased owing to retransmissions caused by channel fading.
as shown in Figures 4 (d) and (e). However, because the relative importance of \( PER \) in the proposed algorithm was set to be higher than that of other scheduling parameters, it is less likely that resources are allocated to nodes that may fail to transmit owing to body posture. In particular, we can see that the performance of UP4 of the proposed algorithm and the performance of UP7 of the FIFS algorithm are similar to each other. The reason is that the FIFS algorithm performs biased resource allocation to high-priority nodes even if there is a high probability of transmission failure. As a result, retransmissions may frequently occur in UP7, and the throughput decreases as the node temperature reaches a threshold. In addition, as shown in Figure 4 (f), the biased resource allocation for UP7 significantly increases the energy consumption. However, the proposed algorithm does not perform resource allocation to a node likely to fail transmission, and induces the node to transmit data frames in random access phases. Therefore, it is possible to increase the efficiency of energy use.

In test case3, the relative importance of parameters \( T \) and \( E \) is set to be equal to each other. Although the throughput and delay performance of UP7 were slightly reduced compared to those of test case 1, the performance difference between each group was reduced, as shown in Figures 4 (g) and (h). However, as illustrated in Figure 4 (i), the power consumption of UP7 decreases because the number of allocation slots is adaptively reduced for nodes with low energy levels, even with high priority in the proposed algorithm.

Based on the results of the three test cases, we conclude that the proposed algorithm can adaptively perform resource allocation based on the relative importance between scheduling parameters. However, there is a challenge in that the pairwise comparison matrix also needs to be changed according to the body characteristics or network conditions.

VI. CONCLUSIONS

In this paper, we propose a priority-based dynamic link scheduling algorithm in WBANs. To make flexible decisions according to network conditions, the relationships and priorities between the proposed scheduling parameters are logically defined using the MCDM method. Based on the outcome of the MCDM method, we can determine scheduling orders and optimize the length of the scheduled uplink allocation interval in the MAP. The simulation results prove that our proposed approach can balance the resource allocation while maintaining the performance of the high-priority nodes. Although the MCDM method can logically define the relationship between multiple criteria, it is considerably difficult to determine the relative importance. In the future, we will study preferential mechanisms that dynamically determine the scales of the pairwise comparison matrix based on either body characteristics or network conditions.

ACKNOWLEDGMENT

This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2018R1D1A1B07043731 and NRF-2019R1A6A3A13091006).

REFERENCES


Symbolic Execution for Network Functions with Time-Driven Logic

Harsha Sharma
harshasha256@gmail.com
Indian Institute of Technology Roorkee

Wenfei Wu
wenfeiwu@tsinghua.edu.cn
Tsinghua University

Bangwen Deng
dbw18@mails.tsinghua.edu.cn
Tsinghua University

ABSTRACT
Symbolic Execution is a commonly used technique in network function (NF) verification, and it helps network operators to find implementation or configuration bugs before the deployment. By studying most existing symbolic execution engine, we realize that they only focus on packet arrival based event logic; we propose that NF modeling language should include time-driven logic to describe the actual NF implementations more accurately and performing complete verification. Thus, we define primitives to express time-driven logic in NF modeling language and develop a symbolic execution engine NF-SE that can verify such logic for NFs for multiple packets. Our prototype of NF-SE and evaluation on multiple example NFs demonstrate its usefulness and correctness.

I. INTRODUCTION
Network Functions (NF) are a family of software widely deployed in networks for the purpose of security (firewall, NAT, IPS/IDS), performance (proxy), improving bandwidth consumption (WAN optimizers) and management (load balancer, rate limiter, packet/byte counter). Network function virtualization (NFV) allows deployment of network functions without any change in the physical infrastructure, and eases the network management (handling software rather than hardware). Software NFs are widely deployed which makes it important for the network operators to verify their correctness before deploying them to production networks (avoiding runtime outage) and network researchers have proposed several verification solutions for NFs [1], [2], [3].

Symbolic execution for network functions (NFs) is a promising verification technique that can statically explore all possible runtime execution paths and generate concrete input test traffic to test all possible valid execution paths. It plays an important role in various network management applications, including network verification [4], configuration validation, and network testing [5], [2], [6]. Existing NF symbolic execution solutions use a domain specific language (DSL) to describe NF behaviors (i.e., model) and inject symbolic packets to execute the NF model and generate concrete test packets from these symbolic packets with constraints on their header fields. [4], [2], [6].

The expressiveness of a DSL decides whether an NF model can be used to represent its actual implementation. Most of the existing symbolic execution solutions assume NF logic is triggered by the packet arrival events, however, we realize that a variety of stateful NFs contain another kind of logic — time-driven logic, where NF states depends on the elapsed time. Without considering this kind of logic, existing symbolic execution engines (SEE) actually execute on a snapshot of the NF, and could lead to false negative/positive results. For example, a stateful firewall with state expiration would reject long-term inactive flow, but the existing SEE predicts the flow to pass through.

In this work, our goal is to extend the existing NF DSL with time-driven logic, enhancing the symbolic execution engine to execute this kind of logic and generating concrete test traffic from symbolic execution. We first study the implementation and application of time-driven logic in existing NFs, and summarize three primitives for expressing such logic. These include a data structure for time value, a function call to get the present time, and a timer to schedule a future event. Then we build a symbolic execution engine for NF models in the enhanced DSL. The SEE processes the packet-driven logic in NF model by exploring all valid execution paths. For time driven logic, it adds timing constraints for packet arrival and timer events. We use these abstract symbolic packets to construct concrete test traffic for the purpose of active testing.

We name our solution NF-SE, and prototype it. Our evaluation shows that NF-SE can verify NFs with time-driven logic, correcting the false positive/negative results of NF verification without time-driven logic. The performance of NF-SE (execution time) is acceptable for several typical NFs (specifically, state-expiration NFs (Section V)); for other NFs (counter-attenuation NFs, (Section V)) NF-SE is not very efficient (still usable) and we suggest ways to optimize the execution time (Section VII-B)).

II. BACKGROUND AND MOTIVATION
Symbolic Execution on NFs is used for verifying the correctness of NFs, and existing symbolic execution solutions need to consider time-driven logic for a more precise result.

A. NF Symbolic Execution
In individual NF verification, exploring all the execution paths helps in finding out implementation or configuration errors [7], [6]; in network-wide NF verification [5], [4],
combining individual NF execution paths with the network topology information helps to reason about network-wide behaviors (e.g., reachability, isolation, etc).

Symbolic execution is a widely used method to infer NF execution paths. It statically executes instructions in NF code using symbolic packet, forks for each conditional branch, keeps a track of constraints on each branch, and outputs abstract input packet for each valid path satisfying all constraints on that path. Concrete test traffic is then generated from these abstract input packets and the behavior of data plane is tested by injecting this concrete test traffic. This is called active testing [5]. Some of the paths in network function code, depends on previous packets arrived i.e. the histories over multiple packets (e.g. prior established connections). To verify such paths, we need to inject multiple symbolic packets, such that the packets are injected sequentially after the execution of previous packets is completed.

Applying general symbolic execution engine (e.g., KLEE [8]) directly on NF code would cause (unnecessary) path explosion, causing intolerable execution times and network irrelevant path constraints. For example, sprintf(ip_str, "%d.%d.%d.%d", ip >> 24 % 256, ip >> 8 % 256, ip % 256) would cause $3^4$ branches, because each 8-bit segment can be 1 or 2 or 3 digits when printed to string in decimal. But in NF specific verification, these branches hardly reflect meaningful network semantics. Thus, symbolic execution is usually performed on NF models which are specified by a domain specific language, called NF modeling language. SymNet [2], BUZZ [5], VMN [4], Kinetic [9] and Vera [6] are examples of such symbolic execution engines with domain specific languages.

B. Time-Driven Logic in NFs

The correctness of the symbolic execution results depends on its fidelity whether the NF behavior model represents the real NF program (i.e., the code). In most of the existing NF symbolic execution work, it is assumed that NF logic is driven by packet arrival events, i.e., the arrival of a packet triggers a series of instructions that processes the packet and updates the NFs internal states. But we observe that there exists another kind of logic — time-driven logic — in NF programs.

Time-driven logic includes the logic that is triggered by timing events and the logic where timing information is utilized for packet processing. We formally define the basic programming primitives for time-driven logic in (Section III). In practice, many NFs contain time-driven logic; for example, a NAT would store established address mapping between internal addresses and external addresses and expire the states after some time, a rate limiter (e.g., leaky bucket algorithm) needs to accumulate “budget” with time and consume the budget by sending packets, and a stateful firewall would preserve the information of valid flows and expire the information after a threshold time.

Example. We use a simplified stateful firewall as a running example in this paper. Fig. 1 shows the basic logic of the

```plaintext
state = CLOSE;
for each packet {
    if (syn packet){
        state=OPEN;
pass;
    } else if (state == OPEN){
pass;
    }
    else{
        drop;
    }
}
```

Fig. 1: Pseudocode of a stateful firewall

A stateful firewall — the SYN packet of a TCP flow punches a hole in the firewall, and all subsequent packets from internal host A to external host B are allowed; without the SYNC packet, any other packets are dropped by the firewall. In the implementation, a state variable “state” is used to record whether a SYN packet has arrived, and has a value either “CLOSE” or “OPEN”. This state can also be used to allow traffic from the external host B to internal host A.

While in previous NF active testing solutions, a stateful firewall is represented in this way (Fig. 1)[10], the practical implementation usually contains another kind of logic — the state would expire after a certain amount of time if no packet arrives during that time. If no packet has arrived for some threshold time, the state would return to “CLOSE”, meaning further communication requires a preceding SYN packet to again reopen the connection. Fig. 2 shows such an implementation: a timer would be triggered to check the state refreshment, if no packet arrives for a threshold time period, the state is reset to “CLOSE”.

Without modeling such time-driven logic in NF models, the symbolic execution results could possibly mismatch the actual
NF behaviors and this may cause false negatives (i.e., reporting unsafe behaviors as safe): in the stateful firewall example (Fig. 2), the actual firewall may stop a flow due to its long-term inactiveness, but due to lack of time-driven logic modeling, the symbolic execution engine would not verify such logic and report pass for such flows. There may also be false positives (i.e., report safe behaviors as unsafe); for example, in a SYN flood detection NF, SYN packets are recorded in a counter, and the counter attenuates with time, if the attenuation time period (i.e., report safe behaviors as unsafe); for example, in a SYN flood detection NF, SYN packets are recorded in a counter, and the counter attenuates with time, if the attenuation time period (time-driven logic is not considered in the verification, all SYN packets in a long-time period would be falsely reported as bursty SYN flood attack.

Goal. Thus, our goal in this paper is to complement NF modeling language with easily verifiable time-driven logic, build a symbolic execution engine for NFs with time-driven logic, and show a few applications where such a complement improves NF verification results.

III. MODELING TIME-DRIVEN LOGIC

We summarize primitives to express time-driven logic, and add them to the NF modeling language.

A. NF Modeling Language

We summarize NF modeling language from several existing solutions [2], [11], [4], and its syntax is shown in the figure 3. This language has the following features:

- Syntax. The language contains variables and constants as basic operators, and commonly used operators such as arithmetic (+, -, *, /, %), relational (>, <, ! =, ==), boolean (&, |, !, &), bitwise (&, |, <<, >>) and indexing ([ ]) operators. Operands and operators together compose expressions. An NF program consists of simple statements such as assignments and complex statements of branching (if-else-then).

- Semantics. In the language, the semantics of expressions follow their mathematical definitions, an assignment statement means to set the value of the left-hand symbol to be that of the right-hand expression, and a branching statement means if the condition is true, execute the if branch, otherwise execute the else branch. The whole program executes each state sequentially from the beginning to the end.

- NF Programming Abstractions. Specifically, a few variables and expressions are summarized and defined as keywords, which expresses NF semantics. In the syntax above, all symbols derived from “header fields” are variables with special meaning, denoting correlating packet header fields; the index operator with a field (e.g., f[sip]) stands for parsing a packet and fetching the field; and states are set variables that are created, retrieved, and updated by flows (e.g., counter[f++]). These NF programming abstractions (1) simplify the NF model representation (avoiding tedious implementation) and (2) avoid path explosion in later symbolic execution (see Section II).

- Loop-freedom. The NF-SE language does not contain loop statement (e.g., while, for). The reason is that symbolic execution needs to statically find all execution paths, but a loop with an unpredictable number of execution times might cause the path search to not terminate. Most of the existing network verification solutions make the same assumption [4], [2], and many NF development frameworks use loop-free program structures (e.g., match-action table in SDN, stateful match-action table in Microsoft VFP [13]).

B. Adding Time-driven Logic

We studied the time-driven logic in typical NFs such as stateful firewall, rate limiter, and intrusion detection systems and summarize the following primitives to express time-driven logic.

- timevalue is a data structure to store time. It can be a timestamp or a time interval. In NF-SE language, timevalue is used as a variable or constant.
- currentTime() would return the timestamp of the current time.
- timer(TIME_INTERVAL, HANDLER, ARGS) is a function call, which schedules the timer logic in HANDLER with arguments of ARGS at a future time TIME_INTERVAL from now.

Execution Model. Among the three primitives, timevalue and currentTime() are variables and a function call that can be embedded into the NF-SE language (as operands or an operator), but timer() needs special notation. Usually the semantics of “triggering some logic at some time” is maintained by a timing framework (e.g., callout timer in early Linux or timing wheel [14]), and the timing framework executes in parallel with the original logic. The two parallel processes usually interact with each other by operating on the shared variables, and there are three execution models:

1) Preemption. Whenever the timer() is triggered, it interrupts the current process and preempts the control flow; after the HANDLER is completed, the control flow returns to original execution location before preemption.
2) **Concurrency.** Both the timer() and the current process execute simultaneously; if there are critical sections (e.g., shared variables) in both processes, concurrency control mechanisms such as locking or mutex are needed.

3) **Sequential.** The current process pauses periodically and checks whether timer() is triggered; if yes, the HANDLER executes to complete, and then the process resumes.

In NF specific domain, we have the following observations and assumptions, which leads us to choose the *sequential execution model* for NF-SE.

1) The packet process and timing process share critical NF states. The timing process usually updates these states and indirectly influences packet processing (e.g., the “state” in Fig. 1, the “token” in Fig. 4).

2) The timing process usually does not operate on packets directly but on NF states. Because packet arrival events are independent of the time elapsed, and it is difficult to execute the HANDLER logic on the packet once packets are not in the packet processing pipeline.

3) If an NF uses parallel execution model for timing process and packet process, we assume it has correct concurrency control on shared states. For example, in Fig. 2, expiring the state by timer and checking the state of a packet should have concurrency control (e.g., locks or mutex).

By the first observation, NF-SE excludes preemption model; and by the third assumption, a correct parallel model should be logically equivalent to a sequential execution of two processes. Thus, NF-SE assumes the sequential execution of timing process and packet process, i.e., when a packet is processed, timer() events are temporarily masked and after the packet processing is completed, the timer events are checked and if there are triggered events, the HANDLER is executed. This assumption also complies with the actual implementation (e.g., P4 rate limiter [15]).

The stateful firewall example in Fig. 2 follows the syntax, and we also show another example of a rate limiter (leaky bucket algorithm) (Fig. 4). It maintains a token variable to record the budget to send packets. Sending packets would consume the token until token is zero and if the token is less than the size of the packet, the packet is dropped; the token is refreshed periodically by a timer.

**IV. SYMBOLICALLY EXECUTE TIME-DRIVEN LOGIC**

In the symbolic execution engine, we use static analysis and constraint solver Z3 to verify packet-driven logic [16]. For time-driven logic, we treat the three primitives as variables and add extra timing constraints (such as constraints of execution order in the time domain). We assume the timestamps of incoming packets to be monotonically increasing and add constraints such as timestamp of the packet 2 is greater than timestamp of packet 1 and construct abstract packets with such constraints.

1 It is possible that timer() triggers packet generation. We categorize this kind of logic to be a control plane message, and is not in the scope of the data plane verification tool NF-SE.

**A. Packet-driven Logic**

Packet arrival would trigger the NF to execute a series of instructions to process it. With the assumption of loop-freedom, the execution is unidirectional from packet input to packet output or drop.

Our symbolic execution engine, NF-SE injects symbolic packets, and statically analyzes all execution paths. NF-SE maintains a trace of instructions, which is organized as a tree; starting from the packet input, it sequentially adds each instruction to the trace: simple assignment instructions would be added to the trace linearly and sequentially (growing the tree by one child), but for a branching instruction (i.e., if-else), the trace tree branches into two children — one further goes to the positive branch (recording the condition of the if statement) and the other goes to the negative branch (recording the negation of the condition). And finally, the static analysis would arrive at the end of the model, exploring all possible paths in the input NF.

A constraint solver solves each path and if the path’s constraints are satisfiable, the solver would output an example of symbolic packet/packets that satisfies all constraints on the valid path.

We made an optimization to delete the trace tree nodes along with its children if branching at that node creates unsatisfiable path. In this way, the static analysis would not explore along that path. This reduces the size of trace tree and thus, execution time for SMT solver to check invalid branches. So, our trace tree only contains valid execution paths for different input packets.

For multiple packets execution, we recursively create and execute different possible paths of trace tree depending on the number of packets required, updating the state variables and placing constraints from previous path execution, and finally outputting all possible execution cases of multiple packets.

**B. Time-driven Logic**

When NF-SE parses an NF model and builds the trace tree, it integrates constraints from the timing primitives. (1) In NF models, timestamps (implemented by timevalue) are usually associated with a variable (e.g., the timestamp when a

```
int token = 1000;
timer(1s, handler);
void handler(){
token=1000;
timer(1s, handler);
}
foreach packet{
  if( token<f[size] ) {
    drop;
  }
  else {
    token-=f[size];
    pass;
  }
}
```

Fig. 4: Psedudocode of a rate limiter
“packet” is received, the timestamp when a “state” is updated). A timestamp’s initial value is set the same as the beginning time of the variable’s lifetime. As NF-SE builds the trace tree, constraints on timestamp initialization are added to the path. For example, the 2nd packet’s timestamp is always greater than the 1st packet, thus, constraint \( \text{pkt1}[ts] < \text{pkt2}[ts] \) is added to the path.

(2) Timestamps may appear in assignments and conditional constraints (e.g., last modified time is within 30s in Fig. 2), and are treated the same as other variables and constraints.

(3) `currentTime()` returns a timestamp of the current time; NF-SE adds a declaration of a new timestamp variable to replace `currentTime()`, and similarly adds constraints that this new timestamp is greater than its previous neighboring timestamp assuming timestamp for incoming packets are increasing in nature.

(4) When \( \text{timer(TIME, HANDLER, ARGS)} \) is “called” to invoke an event in the future time, NF-SE declares a timestamp to record the current time and a constraint which records an association of the timer and handler with this timestamp (for the execution in the future).

(5) When NF-SE gets to a \( \text{timer(TIME, HANDLER, ARGS)} \) execution, it first adds a timestamp (e.g., \( t_1 \)) of current time (as well as its constraints), extracts the timestamp when this timer is “called” (assume \( t_0 \)), and then makes two branches: one with the assumption that the event is triggered (with the constraint \( t_1-t_0 = \text{TIME} \)) and the other that not (with the constraint \( t_1<t_0<\text{TIME} \)). The former branch would first proceed with `HANDLER` which is analyzed as packet processing (like (1), (2), and (3)) and then to the next packet; the latter branch proceeds to process the next packet directly.

(6) There may be multiple timer events after one packet is processed, and they are organized as a queue and symbolically executed similarly as (5). In some cases, the handler of a timer may invoke another timer. NF-SE adds the new invoked timer to the end of the queue.

We add an assumption to avoid infinite loops — the number of recursive invocations from one timer handler to another is bounded. Several practical observations support this assumption, (1) most recursive timer events are idempotent (i.e. multiple execution is equivalent to one execution; examples are state expiration); (2) variables that periodically and monotonically change usually have a threshold (e.g., rate limiter’s token is accumulated to the burst size and does not change any more).

V. IMPLEMENTATION

We use Antlr[17] to build a parser (247 lines of code in Java) for the NF-SE language syntax, which parses an NF model and builds a trace tree. We create a symbolic execution engine using Z3 SMT solver[18], which has 1600+ lines of code in C++.

We implement 14 NFs from existing literature[11], [19], [12], and add time-driven logic to 5 of them; we show the models of NFs with and without time-driven logic in appendix[20]. Time-driven logic usually exists in two formats in these NFs: state expiration and counter attenuation. Stateful firewalls usually has state expiration, such as the firewall in Fig. 2 and TCP 3-way handshake checking firewall and TCP retransmission timeout. Counter attenuation usually exists in rate limiters and intrusion detection systems (IDS), such as super spreader detector (SSD), heavy hitter detector (HHD)[11], [21], and SYN flood detector.

VI. EVALUATION

Our experiments are conducted on a quad-core, Intel i3 laptop with 8 GB RAM. We show that NF-SE can correctly verify NFs’ execution paths, and we measure the execution time taken by our SEE.

A. Functional Validation

In experiment results, we observe that NF-SE overcomes the false positive/negative issues mentioned in section II. It generally outputs more data paths (with more constraints) for time-driven NFs as compared to NFs without time-driven logic.

Case Study: IDS. In an intrusion detection system, if the packet rate (Packet Per Second, PPS) is not high, i.e., the time gap between two consecutive packets is high, then the network should not be assumed under attack/abuse. NF-SE could explore such execution paths where if the two consecutive packets timestamp difference exceeds a threshold limit, then counter is multiplied by a ratio less than 1 to decrease the counter, preventing a false positive of marking this scenario as an attack.

Case Study: Stateful Firewall. We show an example of verifying the NF in Fig. 2. In table I , we list some of the possible packet traces of several execution paths, however, due to space limitation, we only show the first two packets. We can see that path 3 is the case when there is no state expiration and the second packet gets through and path 4 is the case when the state expires and the second packet is dropped. Thus, NF-SE can overcome the false negative case in section II.

![Stateful firewall](image)

Fig. 5: No. of constraints vs no. of input packets for stateful firewall example

We compare the number of execution paths between stateful firewall with and without time-driven logic in Fig. 5 and 6.
TABLE I: Packet trace on some execution paths in the stateful firewall

<table>
<thead>
<tr>
<th>packet index</th>
<th>Execution Path 1</th>
<th>Execution Path 2</th>
<th>Execution Path 3</th>
<th>Execution Path 4</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>packet</td>
<td>SYN</td>
<td>SYN</td>
<td>SYN</td>
<td>SYN</td>
<td>-</td>
</tr>
<tr>
<td>timestamp(s)</td>
<td>1.123456</td>
<td>2.123456</td>
<td>3.123456</td>
<td>3.123456</td>
<td>-</td>
</tr>
<tr>
<td>state transition</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>OPEN</td>
<td>-</td>
</tr>
<tr>
<td>action</td>
<td>pass</td>
<td>pass</td>
<td>pass</td>
<td>pass</td>
<td>-</td>
</tr>
</tbody>
</table>

Fig. 6: No. of constraints vs no. of input packets for stateful firewall example not containing any time-driven logic

In both the figures, the x-axis denotes the number of packets processed by the NF (which is given as an input to SEE), and the y-axis shows the number of constraints in different paths for processing those packets; each path has several constraints, and the vertical bar shows that min-25%-50%-75%-max of the number of constraints of all paths when the NF processes a certain packet. We note the following observations: first, the number of constraints increases almost linearly with the number of packets, as we recursively travel the trace tree for multiple number of packets; second, NFs with time-driven logic have more constraints because of the additional time-driven logic, e.g., median 42 v.s. 32 when processing the 4-th packet; third, the additional constraints for time-driven logic in stateful firewall are not very significant, but this depends on the complexity of the time-driven logic in the NF.

B. Overhead

We make extensive evaluation on the overhead of verifying time-driven logic. Fig. 7 shows the execution time to verify NFs with time-driven logic, Fig. 8 shows execution time for NFs without time-driven logic, and Fig. 9 shows the number of branches (execution paths) for NFs with time-driven logic. We have the following observations.

First, we see that NFs with state expiration (stateful firewall) takes less verification time as compared to NFs with counter attenuation. For example, to symbolically execute 2 packets, firewall takes about 10s, and HHD takes about 100s. Because “state” variables are usually of enumeration type and have less branching choices, but “counter” variables are usually integers, whose value space can be very large. For example, an HHD with a threshold of 100 needs 100 symbolic packets to reach the state transition.

Second, the total execution time can be estimated by the product of the number of branches and the constraint solving time of each branch. More than 96% of the time is consumed in solving constraints using Z3; the branching factor (i.e., the number of execution paths) increases exponentially with the number of input packets, while the number of constraints on each execution path increases linearly with number of input packets.

Third, adding time-driven logic to NFs causes extra overhead; the overhead is acceptable for state-expiration NFs but is quite significant for counter-attenuation NFs. The difference of the overhead for these two types of NFs is caused by the size of the value space for states (usually enumerations) and counters (usually integers).

VII. DISCUSSION

A. Application Scenarios

We discuss how to use NF-SE to explore all execution paths of NFs with time-driven logic, and use the execution paths to correct the false positives/negatives in previous SEE solutions. The individual NF verification can be extended to more NFs, e.g., off-path services such as DNS and DHCP.

Active Testing Trace Generation. There is a trend to apply model-centric programming in both the networking and software engineering community [9], [4], [2], [22], [23], [24], [11], [25]. In model-centric programming, developers use modeling language to describe the software (NF in our case) behaviors and use compilers to translate the model to runnable code. We similarly use NF models to generate execution paths and symbolic packets to exercise those path, then these symbolic abstract packets are translated to actual packet traces; with the compiled code and the packet trace, active testing can be conducted by injecting the concrete packet traces to deployed NFs for verifying the correctness of both the model and the compiler implementation.

Verifying NFs for flows with statistical properties. Traditional NF verification tools answer queries like “Can a flow A get through an NF B with configuration C?” While NFSE adds time-driven logic and can give detailed answers like “When flow A gets through an NF B, X% packets would be dropped depending on timestamps of each packet in flow”.

Verifying network-wide invariants. With NF-SE, verification of multiple NFs along with network topology information will help to reason about the network’s end-to-end behavior.
In such network-wide verification, adding time constraints between NFs can mimic their different processing speed, so that more runtime possibilities can be explored.

### B. Optimization Directions.

The evaluation above reveals a few optimization directions.

1. Define a traffic equivalent class (EC) and solve constraints for one EC, which avoids repeatedly solving constraints for each packet instance in the same EC. There are three dimensions to define an EC: flow space, data path space, and state space. All packets in one EC should be in the same flow (could be a group according to the packet filter rules), go through the same data path, and operate on the same state variables. For example, in HHD and SSD with a threshold of 100 packets, all execution paths other than the case for exceeding the threshold can be verified using 3-4 packets. For solving the execution path in which threshold exceeds, we can use the constraint solving result of one packet for 100 packets EC, and the corresponding action takes place for 100th packet.

2. For timing constraints, we could group a few packets and use one timestamp intervals for the group and execute timer between groups, which reduces the number of timing constraints on each execution path. This sampling-like estimation is a tradeoff between the result precision and the execution efficiency.

### VIII. RELATED WORK

**Individual NF Verification.** Software network functions have large code base, and applying verification techniques such as symbolic execution on large NF implementations results into state explosion because its complexity increases exponentially with number of match action entries. Works such as Vera [6] and P4V [26] target P4 programs and find bugs such as parsing/deparsing errors, invalid memory accesses, loops and tunneling errors. Whereas NF-SE complements them with time-driven logic.

**Network-wide Verification/Testing.** Network verification is a combination of topology discovery and individual NF behavior exploration for verifying chains of NFs in the network. Existing solutions [5], [2], [4], [27], [28], [26] employ model checking techniques which involve creating behaviour models of NFs assuming some oracles or context dependent policies, and applying symbolic execution on these models to systematically explore all possible execution paths of the system. NF-SE can enhance network-wide verification by providing more precise and correct behavior models and adding network-wide timing constraints.

Emulating NFs’ processing in discrete time steps is another approach to explore NFs’ behaviors in time domain (and the verification is described by linear temporal logic)[4]. NF-SE could accelerate this process by using timing constraints to represent multiple discrete time steps.

Another set of works focus on control plane verification[29], [30], and Kinetic[9] verifies network configuration changes, whereas NF-SE provides more precise data plane behaviors. SLA-Verifier [31] focuses on verifying performance metrics, which is another domain. Alembic [32] automatically infers behavioral models of stateful NFs viewed as an ensemble of finite-state machines. It injects input packets at constant interval for each NF and does not verify the temporal effects and cases where output packets depend on histories of previous input packets.

**Time Related Logic.** Varanus [33] is a network monitoring solution with “timeout” semantics, whereas NF-SE is a generic NF modeling and SEE. Kinetic[9] verifies the controller programs against user-specified computation tree logic (CTLs) whereas NF-SE facilitates active testing by generating concrete test traffic.

### IX. CONCLUSION

We built NF-SE, a symbolic execution solution for NFs with time-driven logic. NF-SE includes a DSL with time-driven logic primitives to model NF behaviors and a SEE to explore all the execution paths of the model. Our prototype of NF-SE and evaluation on 5 NFs shows that NF-SE can be used to verify the implementation and configuration of individual NFs; and we show how NF-SE complements and corrects false positives/negatives in existing NF symbolic execution solutions. We also show the potential application scenarios and optimization directions for NF-SE.
Reliable Reverse Engineering of Intel DRAM Addressing Using Performance Counters

Christian Helm  
The University of Tokyo  
Tokyo, Japan  
christian@eidos.ic.i.u-tokyo.ac.jp

Soramichi Akiyama  
The University of Tokyo  
Tokyo, Japan  
akiyama@ci.i.u-tokyo.ac.jp

Kenjiro Taura  
The University of Tokyo  
Tokyo, Japan  
tau@eidos.ic.i.u-tokyo.ac.jp

Abstract—The memory controller of a processor translates the physical memory address to hardware components such as memory channels, ranks, and banks. This DRAM address mapping is of interest to many researchers in the fields of IT security, hardware architecture, system software, and performance tuning. However, Intel processors are using a complex and undocumented DRAM addressing. The addressing can be different for every system because it depends on many aspects such as the processor model, DIMM population on the motherboard, and BIOS settings. Thus an analysis for every individual system is necessary. In this paper, we introduce an automatic and reliable method for reverse engineering the DRAM addressing of Intel server-class processors. In contrast to existing approaches, it is reliable, measurement errors are unlikely to occur, and can be detected if they occur. Our method mainly relies on CPU hardware performance counters to precisely locate the accessed DRAM component. It eliminates the problem of wrong attribution that is common in timing based approaches. We validated our method by reversing engineering the DRAM addressing of a diverse set of Intel processors. This set includes Broadwell, Haswell, and Skylake micro-architectures, with various core counts, DIMM arrangements, and BIOS settings. We show the correctness of the determined addressing functions using micro-benchmarks that access specific DRAM components.

Index Terms—DRAM, Reverse Engineering, Address Mapping, Performance Counters

I. INTRODUCTION

The memory subsystem of a modern computer is complex. The memory is split into different channels to provide higher bandwidth. Organization of DRAM chips in bank groups and banks provide the opportunity for pipelining requests. This has led to increased throughput of DRAM systems over the years without a significant performance increase in the single DRAM cell [1]. The memory controller must interface with those different DRAM components and address them individually. The memory controller receives requests to load data at specific physical addresses. From the physical address, the DRAM controller must determine the channel, rank, bank, row, and column in which the data is stored. The definition of how addresses are translated is called DRAM address mapping.

If the DRAM address mapping is known, it enables a wide range of applications. In hardware architecture and system software research, approaches for better usage of memory channels and banks are being explored. For example, application-aware memory channel partitioning [2], adapted page sizes for better usage of row buffers [3], efficient use of new hybrid memory technology [4], effects of unreliable memory [5], or DRAM layout aware memory allocators [6], [7], [8]. For the evaluation of such new concepts, simulated hardware is often used. If the address mapping is known, such a system can be implemented and evaluated using real processors and applications. The few cases where evaluations are done using real hardware rely on specific processor models where the address mapping is documented or require manual reverse engineering effort. For example, Chandru and Mueller [8] use a Tilera processor, Pan et al. [7] use an AMD Opteron 6128 processor, and PALLOC [6] is implemented on a Xeon W3530 and Freescale P4080. The above-mentioned concepts also showed performance gains which we cannot utilize on other machines without knowing the address mapping.

Researchers in IT security are also interested in the DRAM address mapping to evaluate their concepts. Covert communication channels across CPUs were demonstrated by Pessl et al. [9]. A variation of Rowhammer [10] attacks was introduced by Gruss et al. [11]. And Song et al. show a method for hiding rootkits [12].

The address translation is done in hardware, it is different for every system, and it is mostly undocumented except for a few specific processor models. For example, the documentation of the outdated Intel Xeon 5500 contains a description of the address mapping [13]. However, for newer generations of Intel processors, this information is not published. Up to a certain extent, the mapping is configurable in hardware. At the startup of the system, the BIOS reads the DIMM configuration and programs the configuration registers in the processor. After the initialization, the configuration cannot be changed. The mapping can also be influenced by the BIOS settings. For example, the activation of on-chip NUMA domains changes the addressing [14]. Thus the address mapping depends on many factors and may be different for every system. A general addressing function, for example for a specific processor generation, does not exist.

We introduce an automatic and reliable method for reverse engineering the DRAM addressing on Intel processors. Our tool can automatically find the addressing functions of mem-
ory channels, ranks, bank groups, and banks. It is available online at https://github.com/helchr/reMap. The main idea is to directly and reliably measure the number of accesses to each component (e.g., bank), unlike existing approaches that leverage unreliable numbers such as access latency or the number of bit-flips by rowhammer. With a smart selection of probed addresses, we can determine the addressing functions in a short time. Because we use boolean algebra to resolve the addressing functions, we can differentiate measurement errors from insufficient sampling. In the case of asymmetric DIMM population, we gather additional information from configuration registers. Our tool supports server-class processors with a large amount of memory and also supports asymmetric DIMM population. We demonstrate that we can reverse engineer the address mapping on an Intel Haswell, two Broadwell, and a Skylake system. They are equipped with up to 2TB of RAM and include a system with asymmetric memory channel population. Based on benchmarks that access certain components of the DRAM using the reverse engineered address mapping, we confirm that our method can find the correct address mapping on recent Intel server-class processors.

II. DRAM FUNDAMENTALS

Modern DRAM is organized in a hierarchical arrangement of channels, ranks, bank groups, and banks as shown in Figure 1.

![Figure 1: High level DRAM system organization.](image)

A processor has a memory controller for interfacing the DRAM system. There can also be multiple memory controllers within one CPU. Each of them can have multiple channels. Often, only the total number of channels, but not the separate memory controllers are mentioned in hardware descriptions. E.g. the system in Figure 1 could be seen as a processor with 2 controllers, each with two channels or a processor with four channels. The memory channels can be accessed in parallel. It is the highest degree of parallelism in a DRAM system.

Each channel consists of one or more ranks. A rank consists of multiple banks. Each bank can be used at the same time that other banks are being used. DDR4 RAM consists of 16 banks that are organized in four bank groups, each with four banks. A bank consists of rows and a row buffer as shown in Figure 2. A row is a group of storage cells that are activated in parallel.

![Figure 2: A DRAM bank consists of rows and a row buffer.](image)

A row can only be accessed from the row buffer. The row buffer is essentially a cache that can hold a single row. There are three different states upon a row access. First, if the requested row is cached, it can be accessed immediately. Second, if the row buffer is empty, the requested row needs to be loaded into the buffer before the access is possible. This increases the access latency. Third, if the row buffer is occupied with a row different from the requested row, the currently cached row needs to be written back first. Then the requested row can be loaded into the buffer. This further increases the access latency. A DRAM row is also called a DRAM page.

The memory controller in the CPU is responsible for enabling and addressing those components based on the incoming physical memory address. First, the memory controller selects a channel based on the address mapping and uses the channel’s own address and data lines. The ranks and banks within a channel share the same address and data lines. The correct rank must be activated by using additional rank and bank activation signals. Again, the mapping function defines which rank and bank is activated when accessing a physical address. Except for the configuration at system startup, the mapping of physical addresses to components is static. The address mapping influences how well parallelism and pipelining opportunities can be used [15, Section 13.3]. If we want to implement a custom mapping on existing hardware, we need to know the hardware mapping function and use the physical addresses that result in an access of the desired component.

In multi CPU systems, the DRAM can be organized as NUMA aware or NUMA unaware (i.e., interleaved) mode. In NUMA aware mode, the OS sees each processor with its own distinct DRAM, and the OS (or application) can explicitly access the memory of a CPU. In the NUMA interleave mode, the OS sees only one memory space and the hardware will interleave accesses to all memories with a similar address mapping as for the other DRAM components.

III. RELATED WORK

Seaborn [16] shows manual reverse engineering of the address mapping of a Sandy Bridge processor. First, the author
uses information about the DIMM configuration from the Serial Presence Detect (SPD) ROM stored on the DIMMs themselves to build a hypothesis about the mapping. Then, the author uses a rowhammer tool that causes bit flips in the RAM. Such bit flips can be caused in neighboring rows that are in the same bank. This approach has the following disadvantages. First, the sample generation is not accurate. Bit flips are not guaranteed to occur and may also occur in rows that are not next to each other but further apart. The author describes that this occurred in the experiment and it required manual detection and removal of the outliers. Second, there is no algorithmic method for determining the addressing function. The author manually analyzes the reported addresses of successful bit flips to determine the addressing functions. While it works for this relatively old and simple PC-class processor, it is hardly possible to do a manual analysis on a more modern processor which has more complicated hashing and region based mappings.

A timing-based approach is introduced by Pessl et al. [9]. It is based on the principle that a row buffer hit results in a lower access latency than a row buffer conflict. They use pairs of addresses and repeatedly access the pairs. If both addresses in a pair are in the same bank, alternating accesses will lead to a relatively long delay due to row buffer conflicts. First, these address pairs and timing results are collected. In a second step, the linear xor functions are recovered from the data using a brute force search. They present results for several systems including Sandy Bridge, Ivy Bridge, Haswell, and Skylake, as well as Qualcomm and Samsung mobile processors. They have at most two channels and two ranks. The main disadvantage of this approach is the inaccurate attribution of physical addresses to components. It is based on measuring the timing of accesses, which can be easily disturbed. For example, the processing of other instructions in the pipeline may introduce additional delay. The memory controller is another source of inaccuracies because it can re-schedule DRAM access requests. This changes the timing and can change row buffer access behavior. Despite our best efforts, we could not reproduce the results on our machines. We suspect that such inaccuracies in the measurement lead to the inconsistent results that we have observed.

A performance counter based approach for L3 caches is presented by Maurice et al. [17]. The L3 cache is typically split into slices. The slices are addressed in a similar way as the DRAM components. Each slice has separate access counters, thus for each physical address, it can be determined which slice was accessed. Their approach uses two addresses that differ only by one bit. If the output (accessed cache slice) is the same for both addresses, the bit does not play a role in the result. If the output is different, then this bit is included in the calculation of the cache slice index.

The address mapping is configurable, and there are hardware registers that store the configuration. Reverse engineering of those configuration registers is a method introduced by Hillenbrand [18]. The approach is to change the DIMM configuration of the servers, and then to monitor the changes in the configuration register space. The result is documentation of registers that goes beyond what is officially available by Intel. This study covers Intel Haswell and Broadwell systems.

IV. METHODOLOGY

Our reverse engineering approach has two steps. In the first step, pairs of the physical address and accessed component are collected. This is done by picking an address from a pool and then finding the component that this address accesses. The component is identified using performance counters. After this process, there is a list of physical addresses and the component that the address accessed. In the second step, address mapping functions are calculated from this list of samples. The steps are summarized in Algorithm 1. The remainder of this section explains the individual steps.

Algorithm 1 Pseudocode of the reverse engineering method.

Allocate pool of memory
while not enough addresses do
Pick a virtual address from the pool
Get the physical address
for all components do
Set up measurement for component
Repeatedly access the address
if counter value > number of accesses then
Record pair of physical address and component
end if
end for
end while
Calculate mapping function from samples

A. Memory Allocation

The DRAM address mapping is based on the physical address. If we can control individual bits of the physical address, we can use a structured address selection method, such as the one described in Section IV-B. Only the bits that express offset within a page frame directly translate to bits of the physical address. The bits for the frame number are not under our control. Thus we increase the page size to 2MB, which leads to 21 bits of the physical address being under the direct control of our tool.

B. Address Selection

Theoretically, it is possible to use random addresses from a large pool to gather samples. However it would require the collection of many samples to get enough coverage to reconstruct the addressing function. We want to find out for every bit of the physical address if it influences the accessed DRAM component. Thus addresses that differ in only one bit would allow us to directly judge the influence of this one bit on the result. To generate such addresses we use the following mechanism. First, we take a random address from the allocated memory pool. The next address is generated by reversing the last bit change and then changing the next more
significant bit. In other words, a shifted bitmask with a single one is xored with the initial address. Once we run into the next page frame or out of the boundaries of the address pool, we choose a new random address and start again with modifying the least significant bit.

The memory controller takes physical addresses as the input of the mapping function. Thus we need to gather physical addresses samples. Through the /proc/self/pagemap interface, we can translate the virtual addresses to physical addresses.

C. Performance Counters

For each physical address, we need to know which component is accessed. We use performance counters for each component to measure if they are accessed. Each channel has its own Performance Monitoring Unit (PMU) with its own counters. By selecting the right PMU we count the number of transfers on this channel. For each rank, there is a separate performance event with a separate umask for each bank or bank group. The event definitions can be found in the Intel uncore performance monitoring guide [19]. Each measurement checks one specific channel, rank, and bank. We cycle through all possible components until we found one where the counter value is equal or higher than the number of programmed DRAM accesses. The performance counters that we use require root access or the perf event paranoid flag to be set accordingly. The use of performance counters is a significant difference from a previous timing based approach [9]. The measurement is more reliable and practically eliminates attribution errors. A disadvantage of this method is that only CPUs which have the appropriate performance counters are supported.

D. Enforcing DRAM Accesses

For the measurement, it is required to repeatedly access a certain address, and every access must cause a load from DRAM that we can count. Registers and caches exist to avoid such redundant loads from DRAM. Thus we use the code in Figure 3 with cache line flush and fence instructions that enforce a load from DRAM with every access to the same address.

```c
volatile uint64_t *p = (volatile uint64_t *) addr;
for (unsigned int i = 0; i < NUM_ACCESS; i++)
{
    __mm_clflush((void*)p);
    __mm_lfence();
    *p;
    __mm_lfence();
}
```

Figure 3: The C code that enforces memory loads from DRAM when repeatedly accessing the same address.

The performance counters we use count for the whole system, not just for our application. Thus co-running applications or transfers by the OS cause noise. We need to set the NUM_ACCESSSES variable high enough so that those other accesses do not disturb our measurements. If the number of accesses is too high, it causes unnecessary delays. In our experience, on a system that is not running other applications than the default OS services, 2000 or more accesses allow accurate measurements. On a system that executes other tasks in the background, a higher value may be required.

E. Computing Addressing Functions

The list of samples that contain physical address and accessed component is not useful on its own. We need to extract an addressing function from those samples. We introduce a novel method to resolve the bits of the physical address that are used for addressing components. It reports exact results for used, unused, and unconfirmed bits, as well as other types of errors.

1) Constructing an Equation System: All previous work [9], [16], [17], [18] indicates that the mapping either uses a single bit or a xor combination of multiple bits of the physical address to calculate the component index. Because of the limitation to xor functions, this problem can be formulated as a boolean equation system consisting of two operations (\textit{xor}, \textit{and}). The idea for the construction of the equation system is as follows. The input is a list of samples as shown in Figure 4.

![Figure 4: A list of physical addresses and the accessed component.](image)

We split all the collected samples into a one-bit component address. E.g. If there are four memory channels, two bits are needed to address those four channels. We duplicate the list of physical addresses and build two lists of samples, one for the first bit of the channel address and one for the second bit of the channel address. Each of the bits of the physical address could be used for the calculation of the component address bit. Thus we add a switch (the boolean \textit{and} operation) to every bit. This switch can turn the usage of this bit on or off. If there are multiple bits used, we know that they are combined using the xor operation. Thus we add a xor between every bit. The resulting structure is visualized in Figure 5.

![Figure 5: The list of samples converted into equations with bit switches.](image)
The formalization of this concept as an equation system is shown in Equation 1.

\[
\begin{align*}
    x_{0,0} \cdot b_0 \oplus x_{0,1} \cdot b_1 \oplus \cdots \oplus x_{0,n} \cdot b_n &= c_0 \\
    x_{1,0} \cdot b_0 \oplus x_{1,1} \cdot b_1 \oplus \cdots \oplus x_{1,n} \cdot b_n &= c_1 \\
    &\vdots \\
    x_{m,0} \cdot b_0 \oplus x_{m,1} \cdot b_1 \oplus \cdots \oplus x_{m,n} \cdot b_n &= c_m
\end{align*}
\]  

In Equation 1 the \( \cdot \) symbol denotes the and operation and \( \oplus \) symbol is the xor operation. There are \( n \) unknown parameters \( b \) that express if a bit is used or not. And there are \( m \) physical address samples taken with their individual address bits \( x \). The \( c \) on the right-hand side represents the one-bit component address.

2) Solving the Equation System: The equation system can be solved in the \( F_2 \) space, where xor is an addition and and is a multiplication, with any equation system solver. We use Gaussian elimination. The usage of established linear equation system mathematics brings the advantage of a clear differentiation of the results.

The equations system either has a solution, is partially solvable, or it has no valid solution. If it is not solvable, there are contradicting equations. This can happen in case of wrong measurements or if a more complex address mapping, such as one with multiple regions, is not correctly considered. Theoretically, it could happen that wrong measurements lead to an equation system that is solvable and produces wrong results. However, this would require a systematic error in the measurement. For example, if a performance counter always reports accesses to a different component than the one specified in the measurement setup. Such an error is unlikely to occur, and we never observed such a case in our experiments.

If there is a solution or a partial solution, for every bit of the physical address there are three possible states. A bit can be used for calculating the index, a bit can be unused, or it is unknown if a bit is used. The unknown state happens if there is a partial solution with dependent equations, and there are no samples that cover this specific bit.

This accurate reporting is an advantage over the brute force solver by Pessl et al. [9] because it can identify wrong measurements and differentiate it from unknown bits caused by too few samples.

F. Region Based Mapping

In the case of asymmetric DIMM population or if the number of DIMMs in a channel is not a power of two, the hardware uses more complex region based address mapping. For example, a two-bit wide channel address, calculated using the xor combination of bits, targets four different channels. A space of three different channels can not be expressed. Thus a region based mapping is used in hardware. The regions are address ranges. For each region, a different addressing function can be used. The regions help to get a balanced distribution of requests over all of the three channels. The regions are defined in registers in the memory controller and set up during system startup. Those registers are mostly undocumented, but Hillenbrand [18] provides the locations and decoding for Intel Haswell and Broadwell systems.

The address sample collection works the same, no matter if regions are used or not. For the calculation of the addressing functions, additional steps are necessary. First, we read the region limit addresses from the registers. Then we group all captured addresses into their respective region. Finally, for each of the regions, the addressing functions can be computed as described in Section IV-E.

As already reported previously [18], on some systems, Linux is not able to access the PCI extended configuration space. The channel region definitions are within an address range that is always accessible. But for the ranks, the registers may not be accessible. We experienced this issue on an Intel Haswell system that is equipped with 6 ranks per channel. A workaround is to use a modified Linux kernel [18]. On our two Broadwell systems, complete configuration space was accessible.

Hillenbrand [18] only describes the registers of Broadwell and Haswell based systems. For Skylake and its successors, the register layout changed and is poorly documented. We cannot read the registers to find the regions for channels or banks. On those newer generation systems, our approach only works for a balanced power of two DIMM population.

V. RESULTS

We reverse engineered the address mapping on four different systems and confirm that the obtained addressing functions are correct.

A. Hardware

Table I lists the basic facts of the server systems. In the following, the servers will be referenced by their name, which is in the leftmost column of Table I.

All of the systems are NUMA aware but do not use on-chip NUMA. This means that the OS sees the different processors with their own explicitly addressable memory but cannot see the different memory controllers within one processor. All of the systems are equipped with DDR4 RAM. DDR4 RAM always has four bank groups, each with four banks. Every system uses only a single type of DIMM. For Arcturus, Rigel, and Spica, the configuration is the same on all sockets. On Comet, the memory setup differs for socket 0 and socket 1. On socket 0, only three out of four channels are active due to a hardware defect. On socket 1, all of the four channels are active. Rigel is a Skylake system, which supports up to six memory channels. Our system is equipped with DIMMs on four of the six channels. The rightmost column ranks in Table I is the number of ranks per channel.

All of the experiments were executed on machines running Ubuntu 18.04, and the benchmarks were compiled with gcc 7.4. We configured our tool to use a 20GB address space that is allocated using 2MB pages. We execute 2000 accesses per test and capture a total of 400 address samples.
Table I: The hardware we used for testing the reverse engineering method.

<table>
<thead>
<tr>
<th>Name</th>
<th>Architecture</th>
<th>CPUs</th>
<th>Board</th>
<th>DRAM Speed</th>
<th>DIMMs</th>
<th>Number of Channels</th>
<th>Number of Ranks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arcturus</td>
<td>Broadwell</td>
<td>2x E5-2699v4</td>
<td>Supermicro X10DQGQ</td>
<td>2400Mhz</td>
<td>Micron 36ASF4G72LZ-2G3B1</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Comet</td>
<td>Haswell</td>
<td>2x E5-2699v3</td>
<td>Dell 0NCJW</td>
<td>1867Mhz</td>
<td>SK Hynix HMA84GL7/MMR34N-TF</td>
<td>3 and 4</td>
<td>6</td>
</tr>
<tr>
<td>Rigel</td>
<td>Skylake</td>
<td>2x Xeon 8176</td>
<td>Supermicro X11DPG-QT</td>
<td>2667Mhz</td>
<td>Samsung M386A8K40BM2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Spica</td>
<td>Broadwell</td>
<td>4x E7-8890v4</td>
<td>Supermicro X10QBL-4</td>
<td>1600Mhz</td>
<td>Samsung M386A8K40BM1</td>
<td>4</td>
<td>8</td>
</tr>
</tbody>
</table>

B. Addressing Functions

We use a 0 based numbering for address bits. I. e. the bit number 0 is the least significant bit of an address followed by bit 1 and so on. The bank addressing can either be interpreted as 16 different banks, which need 4 bits for addressing. Or it can be seen as four bank groups, each with four banks. Thus the bank group addressing bits are the same as two of the bank addressing bits. There are separate performance counters for the 16 banks and the four bank groups. The reverse engineering is done separately and we report the results for all 16 banks and the four bank groups individually.

Table II shows the addressing function of Arcturus. Channels and banks use xor hashing. The pattern of the mapping is similar to what is reported by Pessl et al. [9]. However, the individual bits used for addressing are different. It highlights the need to study the mapping for every system individually.

Because Comet is equipped with 6 ranks per channel, a region based mapping is used for the ranks. The rank configuration registers are not accessible using a standard Linux kernel. Thus we cannot resolve the rank addressing and subsequently cannot resolve the bank addressing. On socket 0, there is a region based mapping due to the use of 3 channels. It is shown in Table IV. In the first address region, we did not record any memory accesses. We suspect that it is a reserved hardware area that is not mapped to the DRAM. The second region uses interleaving between the two controllers. Within the first memory controller, the channels are interleaved. The second controller needs no further interleaving because only one channel is available. The third region only uses the first controller and interleaves it’s two channels. The used bits are different from the second region. To the best of our knowledge, this is the first time a region based mapping was successfully reverse engineered.

Spica and Rigel do not use xor hashing. Instead, only single bits are used as shown in Table V and Table VI. With such a configuration, a performance decreasing imbalanced use of channels, ranks, or banks can easily occur if strided memory accesses happen to all fall into the same channel, rank, or bank.

If we equip Rigel with enough DIMMs for all six channels, a region based mapping will be used. Because we do not know the configuration registers for this architecture, reverse engineering is not possible.

C. Speed of Reverse Engineering

In addition to the reliability of the measurements, timing based approaches also have the disadvantage of long processing time. We compared the time required for reverse engineering is not possible.
engineering of our approach and the timing based approach of Pessl et al. [9], even though it did not report the correct result. We did the experiment on Spica. It has a large DRAM size of 512GB per socket, and it has 512 addressable sets (16 banks × 8 ranks × 4 channels). Our tool finds the correct addressing for all sets in ten out of ten tests in an average time of 1:04 minutes. In contrast, the timing based approach needs over 51 hours for the complete reverse engineering process. We assume the optimal case when the mapping is found in the first try, which often does not work due to inaccuracies in the timing measurement.

VI. Usage of Addressing Functions

To confirm that the recovered address mappings are correct, we implement micro benchmarks that reproduce known performance effects of bank and channel usage.

Based on addressing functions shown in Section V-B, we implement a benchmark that can access specific memory components. The benchmark first allocates a large array. Then it calculates the array indexes that are on the specified channels, ranks, and banks. Finally, it accesses the calculated array indexes in a parallel for loop. The number of data accesses stays constant, regardless of the configured channels, ranks, banks, and threads.

A. Channels

Figure 6 shows the measured bandwidth on the four different channels of Spica. In this experiment, the benchmark accesses only one memory channel. This can be clearly seen in the diagram. We measure a bandwidth of about 10GB/s on one of the channels but almost no activity on the other channels. Figure 7 shows the bandwidth on Comet with two out of four channels in use. Figure 8 shows the speedup over the sequential version when only one, two, three, or all four memory channels are used. As expected, the speedup is limited by the number of available channels. This experiment was executed on Spica. These experiments demonstrate that our determined addressing functions for the channels are correct.

B. Banks

It is known that co-running applications or multi-threaded programs, where concurrent threads access different address regions, interfere with each other, and cause increased row buffer conflicts [8], [20], [21]. We implemented a micro-benchmark that reproduces this phenomenon and an optimized version that uses a fixed thread-to-bank assignment. The benchmark reads an array using 16 threads. We limit the access on one channel and one rank so that there are 16 banks available to use. The original version simply accesses the array indexes in ascending order using a loop that is parallelized with OpenMP. Thus each thread accesses different array locations. In the optimized version, we use the same parallel for loop, but each thread accesses only a specific bank. E. g. thread 1 only accesses data stored on bank 1, while at the same time thread 2 only accesses bank 2. We measure the row buffer access status (hit, empty, conflict) as described in the Intel documentation [22].

Figure 9 shows the page hit, page empty, and page conflict ratios measured over time of the original version. We see that, in the original version, after the initialization phase, the page conflicts increase and the page hits decrease. Over 40% of the row accesses result in a page conflict. In contrast, in the optimized version, we can eliminate most of the conflicts and get a high page hit ratio by mapping each thread to a designated bank, as shown in Figure 10.
Figure 9: Row buffer hit, empty, and conflict ratio measured over time on Arcturus when using a standard multi-threaded array access.

Figure 10: Row buffer hit, empty, and conflict ratio measured over time on Arcturus when using a fixed thread-to-bank assignment.

VII. CONCLUSION

We present a new automatic method to reverse engineer Intel’s undocumented DRAM addressing. With the use of performance counters, we achieve reliable DRAM component attribution. Our evaluation shows that we can resolve the correct address mapping on recent Intel machines with a diverse set of memory configurations. In the future, we want to apply this method to other types of processors.

REFERENCES

A NUMA-aware NVM File System Design for Manycore Server Applications

June-Hyung Kim, Youngjae Kim, Safdar Jamil, and Sungyong Park
Department of Computer Science and Engineering
Sogang University, Seoul, South Korea
{junehyung, youkim, safdar, parksy}@sogang.ac.kr

Abstract—NOVA, a state-of-the-art NVM-based file system, is known to have scalability bottlenecks when multiple I/O threads read/write data simultaneously. Recent studies have identified the cause as the coarse-grained lock adopted by NOVA to provide consistency, and proposed fine-grained range-based locks to improve the scalability of NOVA. However, these variants of NOVA only scale on Uniform Memory Access (UMA) architecture and do not scale on Non-Uniform Memory Access (NUMA) architecture. This is because NOVA has no NUMA-aware memory allocation policy and still uses non-scalable file data structures. In this paper, we propose a NUMA-aware NOVA file system which virtualizes the NVM devices located across NUMA nodes so that they can be used as a single address space. The proposed file system adopts a local-first placement policy where file data and metadata are placed preferentially on the local NVM device to reduce the remote access problem. In addition, the lock-free per-core data structures proposed in this file system allow data to be updated concurrently while mitigating the remote memory access. Extensive evaluations show that our NUMA-aware NOVA for parallel writing is scalable with respect to the increased core count and outperforms vanilla NOVA by 2.56-19.18 times.

I. INTRODUCTION

In recent few years, several file systems have been proposed for Non-Volatile Memory (NVM) devices, such as 3D-Xpoint [1]–[4]. Among them, NOVA [2], a state-of-the-art NVM file system, ensures higher throughput and lower read-write latency than block-based file systems. NOVA also ensures consistency of file data and metadata through its log-structured design. NOVA adopts per-inode logging which logs metadata for every write operation. However, NOVA does not provide any degree of scalability in terms of I/O throughput when concurrent shared file I/Os are performed [5], [6]. This is mainly due to the coarse-grained locks on inodes to guarantee consistency of its per-inode logs, which negates the benefits of concurrent nature of NOVA and high-performance NVM devices.

To solve the scalability issue due to low concurrency caused by coarse-grained inode lock in NOVA, it has been suggested to use a fine-grained range based Readers-Writer (RW) lock, referred as range lock instead of the coarse-grained mutex locks for inodes [5], [7], [8]. The range lock-based NOVA succeeds in scaling performance for shared file I/Os. However, this fine-grained lock solution applies only to the Uniform Memory Access (UMA) architecture and does not scale on Non-Uniform Memory Access (NUMA) architecture. This is because NOVA is not designed for NUMA environment. For example, NOVA places all file data and metadata on a single NUMA node [9]. Therefore, threads running on other nodes must access the file via remote memory access, which leads to huge performance loss.

In this paper, we propose a NUMA-aware NOVA to show the scalability on the NUMA-based manycore servers. Our work has the following contributions:

- **Virtualizing NVM Devices:** In order to store files across multiple NUMA nodes, the non-contiguous physical address space of NVMs located at multiple nodes is virtualized into one logical address space as shown in Figure 1. We also propose a local first write policy to place file data and metadata preferentially on the NVM device allocated to the CPU where the thread is executing.

- **Lock-Free Per-Core Data Structures:** We suggest lock-free per-core data structures such as per-inode log using Global Log and Local Log to ensure the scalability by allowing multiple threads to perform write operations concurrently.

- **Evaluation of Intel Optane DC Manycore Servers:** We implemented our proposed ideas in Linux environment. Our extensive evaluations with an Intel Optane DC manycore server show that our proposed NUMA-aware NOVA for parallel writing is scalable with respect to the increased number of cores and outperforms vanilla NOVA by 2.56-19.18 times. In terms of parallel read, the NUMA-aware NOVA is scalable up to 56 cores and outperforms vanilla NOVA by 2.54 times.

II. BACKGROUND AND MOTIVATION

A. NOVA File System

NOVA is a state-of-the-art NVM file system. It adopts log-structure design to guarantee consistency and records the
NOVA uses per-inode locks to avoid inconsistent cases between multiple threads when they write data on the same file at different offsets or they update the log at the same time. For instance, suppose that thread $T_1$ and $T_2$ write on the same file at the same time and there is no per-inode lock. If $T_1$ is writing a log entry (step 2 of Figure 2) but $T_2$ appends a log entry before $T_1$ updates the tail pointer (step 3 of Figure 2), the log entry written by $T_1$ can be overwritten by $T_2$’s log entry. As a result, this will lead to the lose of $T_1$’s data. In our recent work [5], we identified that the NOVA’s per-inode lock becomes the bottleneck for parallel I/O as it serializes threads that try to read or write the same file. Then we proposed a fine-grained range-based RW lock. The proposed range lock first checks the range of the write request; if write ranges do not overlap, the lock can be acquired and the user data (step 1 of Figure 2) can be written concurrently. While the fine-grained range-based RW lock ensures file system scalability in the UMA architecture, it does not help in scaling performance in the NUMA architecture.

III. NUMA-AWARE PER-CPU LOG STRUCTURE DESIGN

A. NOVA file system on Manycore servers

Currently, manycore servers are mostly based on NUMA architecture, and the system applications being designed for manycore machines are required to be aware of their internal architectures to fully exploit their performance. However, NOVA is an NVM based file system designed to reside on a single NVM space. Figure 1 illustrates the non-contiguous address space configuration of multiple NVM devices installed on a NUMA-based manycore machine. Since only NVM space on a single NUMA node can be used to place data and log pages in current NOVA file system, threads executing on other NUMA nodes will cause remote memory access. This becomes the scalability bottleneck. In what follows, we explain how we mitigate these challenges and how we design the NUMA-aware file system.

B. A Unification of NVM Devices

The NUMA-aware file system can reduce remote memory accesses by distributing files over multiple NUMA nodes. In order to design the NUMA-aware NOVA, we have virtualized the non-contiguous NVM address space from multiple NUMA nodes to a single contiguous virtual address space. For this, we import the NVM devices’ information from all NUMA nodes to the superblock when mounting the file system. As a result, the physical addresses of NVM devices are linearly mapped, which allows NUMA-aware NOVA to easily place user data and metadata in multiple NVM devices.

To mitigate the remote memory accesses, a memory allocation policy is required. In our proposed NUMA-aware NOVA, we introduced a local first write policy, where threads give preference to write files to a local NVM device. Figure 3 shows the proposed NUMA-aware memory allocation policy with two NUMA nodes. In this case, threads running on CPU-0 and CPU-1 prefer to write files in physical address space ranging from 0x0040 to 0x0140, while CPU-2 and CPU-3 write over the range from 0x0180 to 0x0280. Also, NUMA-aware NOVA...
divides the entire logical address space based on the size of NVM devices and creates per NUMA node partitions. Further, the partitions are divided into the number of cores in each node and are managed by the per-CPU memory allocator in NOVA. Data and log pages for each thread are allocated from the region allocated to the core on which the thread is running. This allows each I/O thread to allocate data or log pages from the local NVM space first.

C. Lock-Free Per-CPU log structure

File data structures shared among various threads become a major cause of scalability bottleneck for the file system in a NUMA-based system. If files are shared among multiple threads running on different NUMA nodes, it leads to huge performance loss due to frequent remote memory access. Furthermore, since the shared resources are protected by locks to ensure consistency, concurrent update operations are serialized. Therefore, we propose per-CPU data structures for scalable NOVA and extend the NOVA’s per-inode log structure to be a lock-free per-CPU log.

Figure 4 shows the proposed lock-free per-CPU log structure for NOVA, where we introduce two new persistent data structures: Global Log and Local Log. The Global Log is a pointer array for each inode that indexes each CPU and corresponding Local Log pointer as shown in Figure 4. The Local Log is a per-CPU private data structure with head and tail pointers for per-CPU logs. These two data structures allow multiple threads from different NUMA nodes to perform file operations concurrently and improve the scalability.

The new write flow in the NUMA-aware NOVA can be as follows. Suppose that three threads want to write data to a shared file on CPU cores 0, 1, and 2. Then, each thread allocates a data page or a log page using its core’s memory allocator based on the local first allocation policy from the local NVM device. As a result, threads can update their data and log pages concurrently through local memory access. Additionally, they can append a log entry to the log page after each thread updates the tail pointer of Local Log with reference to Global Log. Finally, the index tree of DRAM is updated to point to the new log entries similar to the existing NOVA, and the parallel write operation of NUMA-aware NOVA completes.

IV. EXPERIMENTAL RESULTS

A. Intel Optane DC PM Server

To evaluate the scalability of NOVA, we conducted experiments on a Manycore server equipped with Intel Optane DC Persistent Memory (PM) modules. The detailed specifications of the Testbed are shown in Table I. Each socket is equipped with 6 Intel Optane DC PM modules. The hardware setup used in the experiments is as follows. The memory space of the PM modules within a single socket is provided by the operating system as a contiguous physical address space. In this setup, the access to the PM is interleaved across 6 PM modules in a single socket. However, the server does not allow to aggregate the PM modules across sockets.

To understand the performance of Intel Optane DC PM modules, we measured the bandwidth of each memory module of our Testbed by increasing the number of cores with read-only and write-only workloads. For the experiments, we used the Memory Latency Checker (MLC) tool by Intel [10]. The MLC tool measures the peak bandwidth for workloads, in which multiple threads sequentially read or write data from or to memory devices. In particular, for accurate memory latency measurements, the MLC tool disables the hardware pre-fetcher.

Figure 5(a) shows the measured write bandwidth of PM. In the figure, Optane DC(src, dst) indicates the cores in the socket number src performing I/O operations when they access the memory buffer allocated to Optane DC PM in socket number dst. Optane DC(0,0) reaches the peak bandwidth of 12.3 GB/s on 4 cores but it gradually degrades with the increasing number of cores. On the other hand, the maximum bandwidth of Optane DC(0,1) reaches at the peak of 6.5 GB/s on 8 cores, but it also continues to decrease with the increasing

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel(R) Xeon(R) Platinum 8280M v2 2.70GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Nodes (#):</td>
<td>2, Cores per Node (#): 28</td>
</tr>
<tr>
<td>Memory</td>
<td>DRAMs per Node (#): 6, DDR4, 64 GB * 12 (=768GB)</td>
</tr>
<tr>
<td>PM</td>
<td>Intel Optane DC Persistent Memory</td>
</tr>
<tr>
<td>PMs per Node (#):</td>
<td>6, 128 GB * 12 (=1.5TB)</td>
</tr>
<tr>
<td>OS</td>
<td>Linux kernel 4.13.0</td>
</tr>
</tbody>
</table>
number of cores. It is assumed that the write bandwidth is reduced by the hardware limitation of the Optane DC PM server as described in [11]. Figure 5(b) shows the measured read bandwidth of PM. Optane DC(0,0) bandwidth increases linearly as the number of cores increases, resulting in a maximum of 40.04 GB/s on 28 cores. The bandwidth of Optane DC(0,1) increases as the number of cores increases, resulting in a maximum of 15 GB/s on 8 cores. However, the bandwidth drops after that due to the same hardware limitations.

In short, we have the following observations from Figure 5. First, the read bandwidth of PM is higher than the write bandwidth. The read bandwidths of Optane DC(0,0) and Optane DC(0,1) are 3.24x and 2.31x higher than those of the corresponding write bandwidths, respectively. Second, the degree of the NUMA effect in PM is high. In terms of write, Optane DC(0,0) shows 1.86x higher write bandwidth on 8 cores than Optane DC(0,1), and 16.5x higher write bandwidth on 28 cores. In terms of reading, the read bandwidth of Optane DC(0,0) on 8 core is 1.98x higher than that of Optane DC(0,1) and 35.7x higher on the 28 core.

**B. Evaluation of NUMA-aware NOVA**

For the evaluation, we implemented three versions of NOVA:

- **NOVA(V)** (S1): Vanilla NOVA.
- **NOVA(RL)** (S2): NOVA using a range lock.
- **NOVA(RL+NUMA)** (S3): NOVA using a range lock with NUMA-aware design.

The FxMark benchmark [6] that can generate various workload patterns is used for the evaluations. Specifically, we used three parallel I/O workloads, DWOM, DWOL, and DRBL. The DWOM workload performs multi-threaded shared file writes where multiple threads write to a shared file (N-to-1 write). The DWOL workload performs multi-threaded private file writes where multiple threads write to their private files (N-to-N write). The DRBL workload performs multi-threaded private file reads where multiple threads read the files after writing the files (N-to-N read).

Figure 6(a) verifies the scalability of NOVA(RL) using the DWOM workload in the UMA architecture, while NOVA(V) does not provide any scalability due to the per-inode lock. NOVA(V) shows the peak bandwidth of 0.8 GB/s on 1 core, and it decreases slightly as the number of cores increases. On the other hand, NOVA(RL) shows its maximum throughput of slightly higher than 2 GB/s on 8 cores and maintains the throughput up to 28 cores. This is because the coarse-grained lock is replaced by the fine-grained range lock and thus shared files can be updated in parallel. However, the throughput does not scale after 8 cores for NOVA(RL) as the file log cannot be updated concurrently.

Figure 6(b) shows the results using the DWOM workload in the NUMA architecture. NOVA(V) and NOVA(RL) show the same results as those in the UMA environment. Since these are not NUMA-aware, there is no performance improvement in the NUMA-based Manycore servers. On the other hand, NOVA(RL+NUMA) shows the scalable performance up to 48 cores due to the per-CPU log and local first write policy. The throughput of NOVA(RL+NUMA) increases up to 17.2 GB/s on 48 cores, which exceeds the Testbed’s NUMA boundary of 28 cores. In addition, NOVA(RL+NUMA) shows high parallelism than NOVA(RL) despite NUMA-effect due to the lock-free data structure. For instance, the throughput of NOVA(RL+NUMA) on 16 cores is 31% higher than that of NOVA(RL).

Figure 6(c) presents the results using the DWOL workload in the NUMA architecture. As a range lock is basically designed to improve the DWOM performance, the performance of NOVA(RL) using the DWOL workload is not much different from that of NOVA(V). It scales up to 16 cores, but after that, it slightly decreases as the number of cores increases. We suspect that the hardware bottleneck of the Optane DC PM server shown in Figure 5 causes this problem. Besides, the performance after 28 cores continues to decrease due to the file arrangement without NUMA-awareness. However, NOVA(RL+NUMA) scales up to 48 cores with the maximum throughput of 17.69 GB/s similar to the experiments using the DWOM workload.

Figure 6(d) compares the results using the DRBL workload in the NUMA architecture. In terms of parallel read, all versions of the NOVA scale up to 28 cores. However, NOVA(V) and NOVA(RL) do not scale after 28 cores. In the case
of NOVA(V), the per-inode lock degrades the performance due to the single reader counter problem [12]. On the other hand, NOVA(RL) does not perform well due to the NUMA-effect because all files are placed only at node 0. Finally, NOVA(RL+NUMA) places each thread’s file in the local NVM with local write first policy. This allows threads to read files with local memory access. As a result, NOVA(RL+NUMA) scales up to 56 cores and shows the peak bandwidth of 49.32 GB/s on 56 cores.

V. CONCLUSION

In this paper, we developed a NUMA-aware NOVA file system to support file system scalability on Manycore machines. Specifically, we first virtualized the NVM modules of several NUMA nodes and reduced the number of remote accesses that can occur when performing parallel I/O through the local write first policy. Second, we extended the NOVA’s per-inode log to the lock-free per-CPU log. Extensive evaluations have shown that NUMA-aware NOVA is scalable for parallel writing as the number of cores increases on Intel Optane DC Manycore machines, and shows 19.18 times higher write throughput than vanilla NOVA.

ACKNOWLEDGMENTS

This work was supported by Institute of Information & Communications Technology Planning Evaluation (IITP) grant funded by the Korea government(MSIT) (No. 2014-0-00035, Research on High Performance and Scalable Manycore Operating System). Sungyong Park is the corresponding author.

REFERENCES

New Centrality Measures in Networks and their Applications to the International Trade and Migration Networks

Fuad Aleskerov
National Research University Higher School of Economics
V.A. Trapeznikov Institute of Control Sciences of Russian Academy of Sciences
Moscow, Russia
alesk@hse.ru

Anna Rezyapova
National Research University Higher School of Economics
V.A. Trapeznikov Institute of Control Sciences of Russian Academy of Sciences
Moscow, Russia
arezyapova@hse.ru

Alina Roman
National Research University Higher School of Economics
Moscow, Russia
aaroman@edu.hse.ru

Vyacheslav Yakuba
V.A. Trapeznikov Institute of Control Sciences of Russian Academy of Sciences
National Research University Higher School of Economics
Moscow, Russia
yakuba@ipu.ru

Abstract—New centrality indices in networks are introduced in which parameters of a node and a group influence of nodes to a node are taken into account. The Bundle Influence and Pivotal Influence as well as Copeland in-degree indices are evaluated using the data on international trade and international migration. The influence of the countries is evaluated for four years, 2001, 2006, 2011, and 2016. The Total Influence as a linear combination of the three indices is calculated.

Keywords—influence in networks, Bundle index, Pivotal index, Copeland in-degree index, international trade network, international migration network

I. INTRODUCTION

Centrality analysis in networks attracted a lot attention of the researchers, (see e.g. [1]). Since classic centrality measures disregard important issues in network analysis, parameters of nodes and group influence are not taken into account, the new centrality indices are introduced. If, say, in international migration problem, there is an inflow of migrants to some large country, it might not be an issue, but if the country is small, it might induce problems with public services, employment, etc., so the parameter of the country, such population, should be taken into account. The group influence of several nodes in network on a node should also be considered. For example, in international trade, the situation in which the import to the country from one other country is reduced, differs from the case when import is simultaneously reduced from several other countries.

In this paper the Bundle Influence and the Pivotal Influence indices are introduced to evaluate the group influence of the countries in international trade and international migration networks. The Copeland centrality index is also calculated for the countries, and the total influences are evaluated. The analysis is performed for four years 2001, 2006, 2011, and 2016.

II. CENTRALITY INDICES

Let us define a graph $G = (V, W)$ as a weighted directed graph, with a set of vertices $V$, $|V| = n$, a vertex $v_i$, and a set of edges $W$ with weights $w_{ij}$ for the arc from vertex $i$ to vertex $j$. Let us also define a quota $q_i$ for each vertex $i \in V$. The parameters of the vertices $v_i$ can be set explicitly, as, for example, the total population of the country $i$ in international migration network, or implicitly, as, for example, the total sum of import to the country $i$ in the international trade network, which is equal to the sum of weights $w_{ji}$ of all inbound arcs to the vertex $i$. The quota $q_i$ is defined for each of the vertices $i \in V$, to take into account the parameters of the vertices. The quota $q_i$ can be defined as a share of the value of the parameter of the vertex $v_i$, say, a share of the population of the country, or as a share of the sum of weights $w_{ji}$ of all inbound arcs to the vertex $i$. We define the maximum number of vertices which can simultaneously influence a node as $k$, i.e., the group of nodes influencing the node should not exceed $k$ nodes. These sets are called critical groups or critical sets.

A. Bundle Influence Index

We define the Bundle Influence index to reveal the group influence of vertices to one vertex. Then $BI = (BI_i)$ index is constructed as follows. For each set of vertices $S \subseteq V \setminus \{i\}$, $|S| \leq k$, $\forall j \in S, w_{ji} \neq 0$,

$$S \subseteq V \setminus \{i\}, |S| \leq k, \forall j \in S, w_{ji} \neq 0, \quad (1)$$

the sum of weights $\sum_{j \in S} w_{ji}$ of incoming edges from node $j \in S$ to node $i$ is compared to the quota $q_i$, and the value $BI_i(S)$ is calculated as

$$BI_i(S) = \begin{cases} 1, & \text{if } \sum_{j \in S} w_{ji} \geq q_i \; , \\ 0, & \text{else} \end{cases} \quad (2)$$

Here, the $BI_i(S)$ is equal to 1, if the sum of weights $w_{ji}$ is more or equal to the quota $q_i$, and it is equal to 0, otherwise.
The BI(i) index for the vertex i is defined as the sum of the BI1(S) on all considered subsets S, i.e.

\[ BI(i) = \sum S BI1(S). \]  

(3)

Thus, for each node i ∈ V, the value of the Bundle Influence index BI(i) is equal to the number of the subsets of incoming edges from not more than k vertices, with the sum of weights more or equal to the quota qj.

### B. Pivotal Influence Index

The Pivotal Influence index PI = (Pl1) is defined in the following way. The node jP ∈ S is called pivotal for the node i ∈ V in the set S ⊆ V\{i} with the quota qi, if

\[ \sum j \in S w_{ji} \geq q_i, \text{ but } \sum j \in S \setminus \{jP\} w_{ji} < q_i, \]  

(4)

i.e. the sum of the weights of the incoming edges to node i ∈ V from the nodes of the set S ⊆ V\{i} is greater or equal to the quota qi, but upon excluding the node jP from the set S, the sum of the weights becomes less than the quota.

Then the PI index is constructed as follows. For each of the subsets of the nodes S ⊆ V\{i}, |S| ≤ k, ∀j ∈ S, w_{ji} ≠ 0, the number of the pivotal nodes PI1(S) is calculated. The PI(i) index is defined as the sum PI1(S) on all considered subsets S of cardinality not more than k of the vertices of the incoming edges to the node i, i.e.

\[ PI(i) = \sum S PI1(S). \]  

(5)

As an alternative variant of the index, the sum can be calculated with the multiplier, which is equal to the size of the subset S.

\[ PI'(i) = \sum_S|S| \cdot PI1(S). \]  

(6)

We also use the known Copeland in-degree index CI = (CI1) for each vertex i, which is defined as the sum of weights w_{ji} of the incoming edges from connected vertices j, i.e.

\[ CI(i) = \sum j w_{ji}. \]

All these indices are normalized, i.e., are re-evaluated as a share of the value of the index of the particular vertex i with respect to the sum of the values of all vertices.

The Total Influence index is defined here as the linear combination of the BI, CI, and PI indices for each vertex, i.e., as a weighted sum of these indices.

### C. Indices Example

Consider the example of the network with the following adjacency matrix.

<table>
<thead>
<tr>
<th>Adjacency Matrix for the Example Network</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
</tr>
<tr>
<td>v1</td>
</tr>
<tr>
<td>v2</td>
</tr>
<tr>
<td>v3</td>
</tr>
<tr>
<td>v4</td>
</tr>
<tr>
<td>v5</td>
</tr>
</tbody>
</table>

Note, that pairs of vertices (v2, v4), and (v3, v5) have two-directed arcs. Let be the quotas defined as follows, q1 = q3 = q5 = 4, q2 = 3, and q4 = 5. For each of the vertices consider critical sets with the sum of weights of the arcs to the vertex being not less than quota, and their pivotal vertices. For the vertex v1, the critical sets (or coalitions) are {v2, v3}, {v2, v4}, {v3, v4} with both pivotal vertices in each, and the coalition {v2, v3, v4} with no pivotal vertices. For the vertex v2, the critical set is {v3, v5}, both vertices are pivotal. For the vertex v3 there are no critical sets, since the only inbound arc is from the set {v5}, of one vertex, with the weight of arc 3, while the quota is 4. For the vertex v4 there is a critical set {v1, v5}, both vertices are pivotal. And for the vertex v5 there is one critical set {v2}, consisting of the only pivotal vertex, v2.

To calculate the Copeland values, one needs just to take the sums of the columns of the adjacency matrix. The Total Influence is calculated as the mean of CI, BI, and PI indices.

To sum up, we present the values and the shares of all indices for each of the vertices (numbers are rounded).

<table>
<thead>
<tr>
<th>TABLE II. BI, PI, CI, AND TI INDICES FOR THE EXAMPLE NETWORK</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
</tr>
<tr>
<td>CI</td>
</tr>
<tr>
<td>BI</td>
</tr>
<tr>
<td>PI</td>
</tr>
<tr>
<td>TI</td>
</tr>
</tbody>
</table>

The more detailed discussion and examples of the abovementioned indices are presented in [5]. One can consider the indirect influences of nodes to nodes, taking into account the paths of fixed lengths connecting the vertices. Below we will not consider these cases, however, they are studied in [5].

### III. THE APPLICATIONS

The Bundle, Pivotal, Copeland in-degree, and the Total Influence indices are evaluated for all countries for two networks, the international trade network and the international migration network. The influence of the countries is evaluated over time for four periods of time, 2001, 2006, 2011, and 2016 years.

The data for the international trade network is taken from UN Comtrade and WITS International Trade Statistics database [11, 12]. For the migration network the OECD.Stat and Eurostat International Migration Database is used [9, 10]. In the migration network only those migrants’ flows are taken into account that exceed 1 thousand migrants per year.

The maximum number of the countries, which can simultaneously influence the country, or the number of countries in the critical set, for both applications is defined as k = 5. As for the quotas q_i, for the international trade network the quota is taken as 0.1% of the total import to the importing country for the considered year. For the migration network the...
quota is set as 0.01% of the population of destination country. So, the quota for the trade network is set as a share of import to the country, while for the migration network, the quota is set as the share of the parameter of a node—the share of the population of the country. This allows us to highlight the difference in nature of the two networks, the influence on the international trade activity for the trade network, and the influence on the population of the country in the migration network. As for the size of the critical set, \( k = 5 \), and the fact that the quota is defined independently as the share of all import, or population in the second application, this allows to pay more attention to small-sized critical sets, which are intuitively might be coordinated more easily, and, on the other hand, the influence of these sets are intrinsically relate to each other and to the total activity of the country via the common quota.

IV. INFLUENCE IN THE INTERNATIONAL TRADE NETWORK

As international trade is one of the most essential components of modern economies, the development of the international trade requires the implementation of evidence-based policymaking under high uncertainty. This induces the decision making based policies based on the use of data and data analysis. For the international trade network, the discrepancies of the data are resolved using the methods introduced in [3,4]. The discrepancies originated from differences in the reports of same import flow, as reported by two trade partner countries, the importer and the exporter. The Bundle, Pivotal, Copeland indices are evaluated for the networks. As for the size of the critical set, \( k = 5 \), and the fact that the quota is defined independently as the share of all import, or population in the second application, this allows to pay more attention to small-sized critical sets, which are intuitively might be coordinated more easily, and, on the other hand, the influence of these sets are intrinsically relate to each other and to the total activity of the country via the common quota.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>France</td>
<td>0.027</td>
<td>0.021</td>
<td>0.020</td>
<td>0.021</td>
</tr>
<tr>
<td>Spain</td>
<td>0.025</td>
<td>0.019</td>
<td>0.021</td>
<td>0.019</td>
</tr>
<tr>
<td>South Africa</td>
<td>0.023</td>
<td>0.020</td>
<td>0.018</td>
<td>0.018</td>
</tr>
<tr>
<td>Poland</td>
<td>0.007</td>
<td>0.020</td>
<td>0.018</td>
<td>0.018</td>
</tr>
<tr>
<td>Netherlands</td>
<td>0.019</td>
<td>0.019</td>
<td>0.019</td>
<td>0.018</td>
</tr>
<tr>
<td>Thailand</td>
<td>0.019</td>
<td>0.017</td>
<td>0.016</td>
<td>0.018</td>
</tr>
<tr>
<td>Germany</td>
<td>0.027</td>
<td>0.021</td>
<td>0.018</td>
<td>0.018</td>
</tr>
<tr>
<td>Korea, Rep.</td>
<td>0.019</td>
<td>0.019</td>
<td>0.015</td>
<td>0.017</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>0.024</td>
<td>0.021</td>
<td>0.017</td>
<td>0.017</td>
</tr>
<tr>
<td>United States</td>
<td>0.024</td>
<td>0.021</td>
<td>0.018</td>
<td>0.017</td>
</tr>
<tr>
<td>Belgium</td>
<td>0.015</td>
<td>0.013</td>
<td>0.015</td>
<td>0.018</td>
</tr>
<tr>
<td>Russian Federation</td>
<td>0.014</td>
<td>0.013</td>
<td>0.013</td>
<td>0.016</td>
</tr>
<tr>
<td>Canada</td>
<td>0.020</td>
<td>0.017</td>
<td>0.016</td>
<td>0.016</td>
</tr>
<tr>
<td>Czech Republic</td>
<td>0.017</td>
<td>0.017</td>
<td>0.016</td>
<td>0.016</td>
</tr>
<tr>
<td>Singapore</td>
<td>0.015</td>
<td>0.005</td>
<td>0.008</td>
<td>0.016</td>
</tr>
<tr>
<td>New Zealand</td>
<td>0.007</td>
<td>0.014</td>
<td>0.012</td>
<td>0.015</td>
</tr>
<tr>
<td>Pakistan</td>
<td>0.007</td>
<td>0.013</td>
<td>0.012</td>
<td>0.015</td>
</tr>
<tr>
<td>India</td>
<td>0.011</td>
<td>0.017</td>
<td>0.015</td>
<td>0.015</td>
</tr>
<tr>
<td>Turkey</td>
<td>0.011</td>
<td>0.014</td>
<td>0.015</td>
<td>0.014</td>
</tr>
<tr>
<td>Slovenia</td>
<td>0.015</td>
<td>0.015</td>
<td>0.014</td>
<td>0.014</td>
</tr>
<tr>
<td>Italy</td>
<td>0.019</td>
<td>0.019</td>
<td>0.015</td>
<td>0.014</td>
</tr>
<tr>
<td>Slovak Republic</td>
<td>0.010</td>
<td>0.013</td>
<td>0.012</td>
<td>0.014</td>
</tr>
<tr>
<td>Iceland</td>
<td>0.016</td>
<td>0.013</td>
<td>0.014</td>
<td>0.014</td>
</tr>
<tr>
<td>China</td>
<td>0.011</td>
<td>0.013</td>
<td>0.013</td>
<td>0.014</td>
</tr>
<tr>
<td>Sweden</td>
<td>0.013</td>
<td>0.013</td>
<td>0.013</td>
<td>0.014</td>
</tr>
<tr>
<td>Mexico</td>
<td>0.011</td>
<td>0.017</td>
<td>0.014</td>
<td>0.014</td>
</tr>
<tr>
<td>Australia</td>
<td>0.015</td>
<td>0.014</td>
<td>0.014</td>
<td>0.013</td>
</tr>
<tr>
<td>Finland</td>
<td>0.011</td>
<td>0.011</td>
<td>0.005</td>
<td>0.013</td>
</tr>
<tr>
<td>Brazil</td>
<td>0.013</td>
<td>0.013</td>
<td>0.014</td>
<td>0.013</td>
</tr>
<tr>
<td>Denmark</td>
<td>0.016</td>
<td>0.014</td>
<td>0.013</td>
<td>0.013</td>
</tr>
</tbody>
</table>

One can see that the France, Spain, South Africa, Poland, Netherlands, Thailand, Germany, Korea, United Kingdom, and United States are among top of the countries according to the BI index. It may be surprising that Poland and Thailand are among the top countries, but the reasons are the following.

First, BI index depends on the number of trading partners. If a country has a large number of trading partners, then a larger number of trading partners can form a coalition for the country, and if a country has a small number of trading partners, then a smaller number of trading partners can form a coalition for the country. Indeed, if we take a closer look at the total number of trade partners, we see that Thailand is the second country in the world by the number of trading partners, Poland is the fourth. For comparison, China is the thirty-fourth country by the number of trading partners.

Second, BI index depends on the values of trade flows. If a country’s trade flows are evenly distributed, the risks of trade partners’ group influence are lower; if trade flows are unevenly distributed, the risks of forming a coalition against a given country are higher. Indeed, if we take a closer look at the values of trade flows, we see that top-5 exporters of Thailand account for 57% of Thai imports, and top-5 exporters of Poland account for 52% of Polish imports. For comparison, top-5 exporters of China accounted for 34% of Chinese imports.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>France</td>
<td>0.027</td>
<td>0.021</td>
<td>0.019</td>
<td>0.022</td>
</tr>
<tr>
<td>Thailand</td>
<td>0.022</td>
<td>0.018</td>
<td>0.021</td>
<td>0.022</td>
</tr>
<tr>
<td>Slovak Republic</td>
<td>0.013</td>
<td>0.019</td>
<td>0.017</td>
<td>0.020</td>
</tr>
<tr>
<td>Korea, Rep.</td>
<td>0.021</td>
<td>0.022</td>
<td>0.017</td>
<td>0.020</td>
</tr>
<tr>
<td>Poland</td>
<td>0.006</td>
<td>0.024</td>
<td>0.022</td>
<td>0.019</td>
</tr>
<tr>
<td>Singapore</td>
<td>0.018</td>
<td>0.004</td>
<td>0.009</td>
<td>0.019</td>
</tr>
<tr>
<td>Mexico</td>
<td>0.015</td>
<td>0.022</td>
<td>0.017</td>
<td>0.019</td>
</tr>
<tr>
<td>Czech Republic</td>
<td>0.021</td>
<td>0.022</td>
<td>0.021</td>
<td>0.019</td>
</tr>
<tr>
<td>New Zealand</td>
<td>0.008</td>
<td>0.017</td>
<td>0.015</td>
<td>0.018</td>
</tr>
<tr>
<td>Canada</td>
<td>0.028</td>
<td>0.020</td>
<td>0.017</td>
<td>0.018</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>0.025</td>
<td>0.019</td>
<td>0.016</td>
<td>0.018</td>
</tr>
<tr>
<td>Germany</td>
<td>0.028</td>
<td>0.022</td>
<td>0.016</td>
<td>0.017</td>
</tr>
<tr>
<td>Slovenia</td>
<td>0.019</td>
<td>0.017</td>
<td>0.014</td>
<td>0.017</td>
</tr>
<tr>
<td>South Africa</td>
<td>0.025</td>
<td>0.019</td>
<td>0.018</td>
<td>0.017</td>
</tr>
<tr>
<td>Pakistan</td>
<td>0.006</td>
<td>0.012</td>
<td>0.013</td>
<td>0.017</td>
</tr>
<tr>
<td>Belgium</td>
<td>0.014</td>
<td>0.012</td>
<td>0.015</td>
<td>0.017</td>
</tr>
<tr>
<td>Spain</td>
<td>0.023</td>
<td>0.015</td>
<td>0.017</td>
<td>0.017</td>
</tr>
<tr>
<td>Ireland</td>
<td>0.022</td>
<td>0.017</td>
<td>0.018</td>
<td>0.017</td>
</tr>
<tr>
<td>United States</td>
<td>0.025</td>
<td>0.019</td>
<td>0.016</td>
<td>0.016</td>
</tr>
<tr>
<td>Netherlands</td>
<td>0.019</td>
<td>0.019</td>
<td>0.018</td>
<td>0.016</td>
</tr>
<tr>
<td>Sweden</td>
<td>0.015</td>
<td>0.015</td>
<td>0.013</td>
<td>0.016</td>
</tr>
<tr>
<td>Indonesia</td>
<td>0.018</td>
<td>0.014</td>
<td>0.015</td>
<td>0.016</td>
</tr>
<tr>
<td>Australia</td>
<td>0.019</td>
<td>0.018</td>
<td>0.016</td>
<td>0.015</td>
</tr>
<tr>
<td>Bahrain</td>
<td>0.001</td>
<td>0.001</td>
<td>0.005</td>
<td>0.014</td>
</tr>
<tr>
<td>Finland</td>
<td>0.011</td>
<td>0.011</td>
<td>0.005</td>
<td>0.014</td>
</tr>
<tr>
<td>Russian Federation</td>
<td>0.012</td>
<td>0.012</td>
<td>0.011</td>
<td>0.014</td>
</tr>
<tr>
<td>Austria</td>
<td>0.022</td>
<td>0.018</td>
<td>0.018</td>
<td>0.014</td>
</tr>
<tr>
<td>Japan</td>
<td>0.022</td>
<td>0.018</td>
<td>0.014</td>
<td>0.013</td>
</tr>
<tr>
<td>Brazil</td>
<td>0.012</td>
<td>0.012</td>
<td>0.013</td>
<td>0.013</td>
</tr>
<tr>
<td>Denmark</td>
<td>0.016</td>
<td>0.014</td>
<td>0.013</td>
<td>0.013</td>
</tr>
</tbody>
</table>

Countries like France, Thailand, Slovak Republic, Korea, Poland, Singapore, Mexico, Czech Republic, New Zealand, Canada, United Kingdom, Germany have high values according to PI index. Again, it may be surprising that Thailand, Slovak Republic, Poland, Singapore are among the
top countries. The reasons of such distribution of countries are similar to those ones mentioned for BI index.

First, PI index takes into account the total number of trade partners. The more trading partners a country has, the greater the risks of having a key country, and vice versa. Since Thailand, Poland and Slovakia have one of the largest number of trading partners worldwide, this risk is reflected in the PI index.

Second, PI index also depends on the values of trade flows. The higher the value of the trade flow, the more likely the trading partner is to be a pivotal one. If we look at the statistics, we see that Thailand, Slovak Republic, Poland, Singapore have highly concentrated imports from other countries.

### Table VI: Copeland In-Degree Influence, Trade, Top 30 Countries

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>United States</td>
<td>0.181</td>
<td>0.153</td>
<td>0.120</td>
<td>0.137</td>
</tr>
<tr>
<td>China</td>
<td>0.036</td>
<td>0.055</td>
<td>0.086</td>
<td>0.088</td>
</tr>
<tr>
<td>Germany</td>
<td>0.079</td>
<td>0.078</td>
<td>0.070</td>
<td>0.068</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>0.058</td>
<td>0.049</td>
<td>0.040</td>
<td>0.043</td>
</tr>
<tr>
<td>France</td>
<td>0.053</td>
<td>0.049</td>
<td>0.042</td>
<td>0.038</td>
</tr>
<tr>
<td>Hong Kong, China</td>
<td>0.025</td>
<td>0.028</td>
<td>0.031</td>
<td>0.037</td>
</tr>
<tr>
<td>Japan</td>
<td>0.051</td>
<td>0.043</td>
<td>0.043</td>
<td>0.035</td>
</tr>
<tr>
<td>Netherlands</td>
<td>0.036</td>
<td>0.038</td>
<td>0.037</td>
<td>0.033</td>
</tr>
<tr>
<td>Italy</td>
<td>0.039</td>
<td>0.038</td>
<td>0.032</td>
<td>0.027</td>
</tr>
<tr>
<td>Korea, Rep.</td>
<td>0.023</td>
<td>0.025</td>
<td>0.028</td>
<td>0.026</td>
</tr>
<tr>
<td>Canada</td>
<td>0.036</td>
<td>0.029</td>
<td>0.024</td>
<td>0.025</td>
</tr>
<tr>
<td>Belgium</td>
<td>0.030</td>
<td>0.030</td>
<td>0.026</td>
<td>0.024</td>
</tr>
<tr>
<td>India</td>
<td>0.006</td>
<td>0.014</td>
<td>0.025</td>
<td>0.023</td>
</tr>
<tr>
<td>Mexico</td>
<td>0.022</td>
<td>0.017</td>
<td>0.016</td>
<td>0.020</td>
</tr>
<tr>
<td>Spain</td>
<td>0.027</td>
<td>0.028</td>
<td>0.020</td>
<td>0.020</td>
</tr>
<tr>
<td>Switzerland</td>
<td>0.015</td>
<td>0.015</td>
<td>0.017</td>
<td>0.017</td>
</tr>
<tr>
<td>Singapore</td>
<td>0.018</td>
<td>0.017</td>
<td>0.018</td>
<td>0.017</td>
</tr>
<tr>
<td>United Arab Emirates</td>
<td>0.006</td>
<td>0.009</td>
<td>0.011</td>
<td>0.014</td>
</tr>
<tr>
<td>Poland</td>
<td>0.008</td>
<td>0.011</td>
<td>0.013</td>
<td>0.014</td>
</tr>
<tr>
<td>Russian Federation</td>
<td>0.008</td>
<td>0.014</td>
<td>0.017</td>
<td>0.012</td>
</tr>
<tr>
<td>Turkey</td>
<td>0.007</td>
<td>0.011</td>
<td>0.013</td>
<td>0.012</td>
</tr>
<tr>
<td>Australia</td>
<td>0.010</td>
<td>0.011</td>
<td>0.012</td>
<td>0.012</td>
</tr>
<tr>
<td>Vietnam</td>
<td>0.002</td>
<td>0.004</td>
<td>0.006</td>
<td>0.012</td>
</tr>
<tr>
<td>Thailand</td>
<td>0.009</td>
<td>0.010</td>
<td>0.011</td>
<td>0.012</td>
</tr>
<tr>
<td>Malaysia</td>
<td>0.012</td>
<td>0.011</td>
<td>0.011</td>
<td>0.011</td>
</tr>
<tr>
<td>Austria</td>
<td>0.012</td>
<td>0.012</td>
<td>0.011</td>
<td>0.010</td>
</tr>
<tr>
<td>Brazil</td>
<td>0.010</td>
<td>0.008</td>
<td>0.013</td>
<td>0.009</td>
</tr>
<tr>
<td>Sweden</td>
<td>0.011</td>
<td>0.011</td>
<td>0.010</td>
<td>0.009</td>
</tr>
<tr>
<td>Czech Republic</td>
<td>0.006</td>
<td>0.008</td>
<td>0.008</td>
<td>0.009</td>
</tr>
<tr>
<td>Indonesia</td>
<td>0.006</td>
<td>0.006</td>
<td>0.009</td>
<td>0.009</td>
</tr>
</tbody>
</table>

For the CI index, the ordering of the top countries in 2016 is different: United States, China, Germany, United Kingdom, France, Hong Kong (China), Japan, Netherlands, Italy, Korea. This distribution of countries is directly related to the volume of countries \(^1\) imports: the more a country imported, the more it was exposed to external influence, and vice versa.

The Total influence index for the top 30 countries is as follows. Let us notice that this index allows us to reveal new characteristics of cross-country interaction that have not been considered yet in the network analysis.

### Table VII: Total Influence, Trade, Top 30 Countries

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>United States</td>
<td>0.040</td>
<td>0.033</td>
<td>0.028</td>
<td>0.029</td>
</tr>
<tr>
<td>France</td>
<td>0.030</td>
<td>0.024</td>
<td>0.022</td>
<td>0.023</td>
</tr>
<tr>
<td>Germany</td>
<td>0.032</td>
<td>0.027</td>
<td>0.022</td>
<td>0.022</td>
</tr>
<tr>
<td>China</td>
<td>0.014</td>
<td>0.017</td>
<td>0.019</td>
<td>0.020</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>0.028</td>
<td>0.023</td>
<td>0.019</td>
<td>0.020</td>
</tr>
</tbody>
</table>

Now we present the charts for the top 10 countries in dynamics, for all four considered periods.

For the BI index, Poland has big increase from 2001 to 2006 years, among top countries.

![Fig. 1. BI index over the years, trade, top 10 countries](image1)

![Fig. 2. PI index over the years, trade, top 10 countries](image2)
For the PI index Poland, New Zealand, Mexico, and Slovakia have increasing values from 2001 to 2006 years, while line for Singapore is U-shaped.

![Graph](image)

Fig. 3. Copeland index over the years, trade, top 10 countries

For the CI index, on the other hand, the United States has far higher values than other countries, with Germany and the increasing China on second places.

![Graph](image)

Fig. 4. Total Influence index over the years, trade, top 10 countries

The TI index shows United States, France, Germany, and China among tops of the countries.

V. INFLUENCE IN THE MIGRATION NETWORK

International migration is one of the aspects of the globalization that needs to perform the analysis in integrated way. Network analysis, as comparatively new method in the migration studies, originates from social science, namely, from an analysis of social interactions of individuals. Network is presented as a graph, where nodes are agents and edges are relationships between them. The migration process is modelled as a weighted-directed graph, where nodes are countries and edges are migration flows between them. Countries, then, are connected through flows of migrants. Several studies have analyzed the process of the international migration by countries, various network properties and clusters in the network [6,7] and studied migration network between the OECD countries [8].

We evaluated the influence of countries based on the dataset on international migration from OECD and Eurostat [9,10]. In this dataset the migration is defined as a flow of persons leaving from their country of origin for another one. Origin country in this case is defined by citizenship, place of birth or residence depending on the regulations in the country.

In this work the new model of influence in the network is applied to the of international migration problem. The model deals with the novel centrality indices developed in [5] to the international migration process between the pairs of the countries of the world. One of the particular directions in the exploratory network analysis of the international migration is the analysis of network’s characteristics on the nodes’ level, in our case – countries.

The proposed centrality indices also take into account the individual characteristics of the nodes of the network, namely, the demographic characteristic – total population. The countries’ total population is important in terms of international migration process. For example, if there is a migration flow of 50 thousand migrants arriving to the country A with the population 10 million people and the same migration flow arriving to the country B with the population of 500 thousand people. The flow would account for 0.05% of the country A population, however, it would be as high as 10% of the country B population, which might be treated as crucial for the country B population and almost negligible for the population of country A. Thus, the individual parameters of the nodes, population in this case, are very important in the evaluation of the centrality of the countries in the network, specifically, in the international migration process.

The proposed indices, The BI and PI indices, take into consideration group influence in the network. Consider the following example in this context. Let the migration flow from a single country in the Middle East to a country in Europe, for instance, may not be crucial for the population of the receiving country. On the other hand, if the information about the living conditions in the destination country would spread among other countries in the Middle East, more migrants would consider the possibility of moving there. As the destination country becomes more attractive in view of potential migrants, the migration flow might increase from most of the countries of the region, that might result that migration issues start playing a significant role for the receiving country society. Therefore, the group influence is important to account for in the evaluation of countries’ centrality in the network.

Generally, the influence in international migration in this work is modelled the following way. The new centrality indices evaluate the group influence of countries from different points of view. The Bundle Influence corresponds to group influence of critically large sets of countries which have the migration flow to the destination country. The Pivotal Influence estimates the influence of a pivotal country inside each of such groups. The Copeland in-degree index refers to migrant inflow to the country. The Total Influence is estimated as a linear combination of three indices. The individual country characteristics are accounted by the quota that refers to the percentage of the receiving country population.

In the migration network the arcs are only present for the number of migrants not less than 1 thousand people. The quota is taken as 0.01% of population of destination country.

The results of the evaluation of the indices for top countries are presented on the tables.
TABLE VIII. BUNDLE INFLUENCE, MIGRATION, TOP 20 COUNTRIES

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Germany</td>
<td>0.140</td>
<td>0.091</td>
<td>0.121</td>
<td>0.238</td>
</tr>
<tr>
<td>United States</td>
<td>0.193</td>
<td>0.193</td>
<td>0.173</td>
<td>0.176</td>
</tr>
<tr>
<td>Japan</td>
<td>0.053</td>
<td>0.045</td>
<td>0.037</td>
<td>0.059</td>
</tr>
<tr>
<td>Republic of Korea</td>
<td>0.026</td>
<td>0.046</td>
<td>0.050</td>
<td>0.059</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>0.064</td>
<td>0.067</td>
<td>0.084</td>
<td>0.056</td>
</tr>
<tr>
<td>Spain</td>
<td>0.072</td>
<td>0.106</td>
<td>0.055</td>
<td>0.051</td>
</tr>
<tr>
<td>Italy</td>
<td>0.035</td>
<td>0.044</td>
<td>0.057</td>
<td>0.041</td>
</tr>
<tr>
<td>Canada</td>
<td>0.040</td>
<td>0.035</td>
<td>0.035</td>
<td>0.040</td>
</tr>
<tr>
<td>France</td>
<td>0.027</td>
<td>0.031</td>
<td>0.031</td>
<td>0.033</td>
</tr>
<tr>
<td>Australia</td>
<td>0.054</td>
<td>0.053</td>
<td>0.035</td>
<td>0.029</td>
</tr>
<tr>
<td>Poland</td>
<td>0.014</td>
<td>0.019</td>
<td>0.027</td>
<td>0.021</td>
</tr>
<tr>
<td>Netherlands</td>
<td>0.016</td>
<td>0.010</td>
<td>0.015</td>
<td>0.021</td>
</tr>
<tr>
<td>Chile</td>
<td>0.004</td>
<td>0.006</td>
<td>0.011</td>
<td>0.019</td>
</tr>
<tr>
<td>Austria</td>
<td>0.012</td>
<td>0.010</td>
<td>0.010</td>
<td>0.019</td>
</tr>
<tr>
<td>Sweden</td>
<td>0.007</td>
<td>0.010</td>
<td>0.010</td>
<td>0.017</td>
</tr>
<tr>
<td>Switzerland</td>
<td>0.013</td>
<td>0.011</td>
<td>0.017</td>
<td>0.017</td>
</tr>
<tr>
<td>Belgium</td>
<td>0.008</td>
<td>0.008</td>
<td>0.017</td>
<td>0.012</td>
</tr>
<tr>
<td>New Zealand</td>
<td>0.013</td>
<td>0.012</td>
<td>0.010</td>
<td>0.012</td>
</tr>
<tr>
<td>Romania</td>
<td>0.005</td>
<td>0.005</td>
<td>0.019</td>
<td>0.012</td>
</tr>
<tr>
<td>Portugal</td>
<td>0.023</td>
<td>0.006</td>
<td>0.009</td>
<td>0.007</td>
</tr>
</tbody>
</table>

For the BI index, the values for Germany is much bigger, than for other countries in 2016 year, though it was less than that of the United States in 2001.

TABLE IX. TOTAL INFLUENCE, MIGRATION, TOP 5 COUNTRIES

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>United States</td>
<td>0.887</td>
<td>0.970</td>
<td>0.968</td>
<td>0.963</td>
</tr>
<tr>
<td>Germany</td>
<td>0.110</td>
<td>0.028</td>
<td>0.019</td>
<td>0.029</td>
</tr>
<tr>
<td>France</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
<td>0.005</td>
</tr>
<tr>
<td>Japan</td>
<td>0.000</td>
<td>0.000</td>
<td>0.000</td>
<td>0.001</td>
</tr>
<tr>
<td>Italy</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
<td>0.001</td>
</tr>
</tbody>
</table>

The PI index shows that the influence if the United States accounts for 89-97% of the all countries, while Germany and France on the second places.

TABLE X. COPPELND IN-DEGREE INFLUENCE, MIGRATION, TOP 20 COUNTRIES

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Germany</td>
<td>0.140</td>
<td>0.091</td>
<td>0.121</td>
<td>0.238</td>
</tr>
<tr>
<td>United States</td>
<td>0.193</td>
<td>0.193</td>
<td>0.173</td>
<td>0.176</td>
</tr>
<tr>
<td>Japan</td>
<td>0.053</td>
<td>0.045</td>
<td>0.037</td>
<td>0.059</td>
</tr>
<tr>
<td>Republic of Korea</td>
<td>0.026</td>
<td>0.046</td>
<td>0.050</td>
<td>0.059</td>
</tr>
<tr>
<td>United Kingdom</td>
<td>0.064</td>
<td>0.067</td>
<td>0.084</td>
<td>0.056</td>
</tr>
<tr>
<td>Spain</td>
<td>0.072</td>
<td>0.106</td>
<td>0.055</td>
<td>0.051</td>
</tr>
<tr>
<td>Italy</td>
<td>0.035</td>
<td>0.044</td>
<td>0.057</td>
<td>0.041</td>
</tr>
<tr>
<td>Canada</td>
<td>0.040</td>
<td>0.035</td>
<td>0.035</td>
<td>0.040</td>
</tr>
<tr>
<td>France</td>
<td>0.027</td>
<td>0.031</td>
<td>0.031</td>
<td>0.033</td>
</tr>
<tr>
<td>Australia</td>
<td>0.054</td>
<td>0.053</td>
<td>0.035</td>
<td>0.029</td>
</tr>
<tr>
<td>Poland</td>
<td>0.014</td>
<td>0.019</td>
<td>0.027</td>
<td>0.021</td>
</tr>
<tr>
<td>Netherlands</td>
<td>0.016</td>
<td>0.010</td>
<td>0.015</td>
<td>0.021</td>
</tr>
<tr>
<td>Chile</td>
<td>0.004</td>
<td>0.006</td>
<td>0.011</td>
<td>0.019</td>
</tr>
<tr>
<td>Austria</td>
<td>0.012</td>
<td>0.010</td>
<td>0.010</td>
<td>0.019</td>
</tr>
<tr>
<td>Sweden</td>
<td>0.007</td>
<td>0.010</td>
<td>0.010</td>
<td>0.017</td>
</tr>
<tr>
<td>Switzerland</td>
<td>0.013</td>
<td>0.011</td>
<td>0.017</td>
<td>0.017</td>
</tr>
<tr>
<td>Belgium</td>
<td>0.008</td>
<td>0.008</td>
<td>0.017</td>
<td>0.012</td>
</tr>
<tr>
<td>New Zealand</td>
<td>0.013</td>
<td>0.012</td>
<td>0.010</td>
<td>0.012</td>
</tr>
<tr>
<td>Romania</td>
<td>0.005</td>
<td>0.005</td>
<td>0.019</td>
<td>0.012</td>
</tr>
<tr>
<td>Portugal</td>
<td>0.023</td>
<td>0.006</td>
<td>0.009</td>
<td>0.007</td>
</tr>
</tbody>
</table>

For the Copeland index, Germany, United States, Japan, Korea, United Kingdom, Spain, Italy, Canada, France, and Australia are among the pot countries.

The Total influence of the United States, Germany, Spain, France in sum takes around 93% of all countries.

Let us present the charts of the influence of the countries in the migration network for four periods of time.

Fig. 5. BI index over the years, migration, top 10 countries

Two countries, the United States, Germany are most influential according to the BI index, with the highest values of the the United States. The chart shows, that the dynamics of these countries is somewhat opposite to each other.

Fig. 6. PI index over the years, migration, top 10 countries
The same can be seen and for the PI index, but the United States has almost all influence in the network.

Fig. 7. Copeland index over the years, migration, top 10 countries

The Copeland index, to the contrary, assigns more influence to Germany in 2016, while the influence of the the United States is still very high. The UK and Spain have influence on the third places.

Fig. 8. Total Influence index over the years, migration, top 10 countries

The TI index ranks the United States top, above Germany, and with th UK the third country in 2011 year.

In general, according to the Copeland index, Germany, the United States, the United Kingdom and South Korea are countries with the highest inflow of migrants in 2016. The Bundle index places European countries, like Spain, France and Italy high in the ranking. Pivotal index ranks Japan and Poland higher comparing to the other indices. The results show that apart from direct influence it is important to consider group influence. Group influence shows the countries that are important on the community level, most of European countries are outlined as influential in the network. Pivotal Influence shows the countries that are crucial actors in the international migration in their region, like Poland, as well, as Japan, which are countries with both high emigration and immigration flows.

VI. CONCLUSION

New centrality indices, Bundle and Pivotal Influence, as well as the Copeland in-degree index, and the Total Influence index, as the combination of them, have been evaluated for all countries and for four periods of time of total length of 15 years for two networks, the international trade network, and the international migration network. The result shows that the most influential countries for the trade network are France, Spain, the United States, and the United Kingdom, and for the migration network are the United State, Germany, United Kingdom, and South Korea. The further modeling with different definitions of the quota, and using other parameters of the countries, as, for example, GDP or unemployment rate would be interesting. One of the possible extensions of this analysis is the evaluation of influence on the level of flows meaning that in addition to the country ranking the ranking of migration or trade flows can be provided as well. It may help to explore the influence in the network in a new perspective and bring implications for the migration and trade policy of the countries highly involved in the process. However, in the case of migration, this analysis is limited due to the interpretation of the data. The data (OECD, Eurostat) are collected from the country statistics and do not refer to the chain migration meaning that people moving between the countries are different. Therefore, the ranking of migration flows might be used only on the first level. For example, it is possible to analyze the largest migration flows to or from the most influential countries.

ACKNOWLEDGMENTS

The paper has been prepared within the framework of the HSE University Basic Research Program and funded by the Russian Academic Excellence Project '5-100'. We thank the International Center of Decision Choice and Analysis (DeCan Center) of the National Research University Higher School of Economics. The work is partially supported by RFBR grant #18-01-00804a. We are grateful to the participants of the EURO-2019 Conference (Dublin, Ireland) for helpful comments. We are also grateful to the anonymous referee for valuable comments.

REFERENCES


Optimum Checkpoints for Time and Energy
Erol Gelenbe, Paweł Boryszko, Miltiadis Siavvas and Joanna Domanska

Abstract—We study programs which operate in the presence of possible failures and which must be restarted from the beginning after each failure. In such systems checkpoinst are introduced to reduce the large costs of program restarts when failures occur. Here we suggest that checkpoints should be introduced in a manner which assures effective reliability, while reducing both the computational overhead as much as possible, but also to save energy. We compute the total average program execution time in the presence of checkpoints so as to limit the re-execution time of the program from the most recent checkpoint. We also study the total energy consumption of the program under the same conditions, and formulate an optimization problem to minimize a weighted sum of both average computation time and energy. This approach is placed in the context of Application Level Checkpointing and Restart (ALCR). We then focus on checkpoints placed at the beginning of a loop, and derive the optimum placement of checkpoints to minimize a weighted combination of the program’s execution time and energy consumption. Numerical results are presented to illustrate the analysis. Finally we describe a software tool with a graphical interface that has been designed to assist a system designer in choosing the optimum checkpoint for a given program as a function of different failure rates and other parameters.

Index Terms—Optimum checkpoints; Program loops; Software reliability; Energy saving; Application Level Checkpoints and Restart (ALCR); Time optimization; Toolbox; User interface

I. INTRODUCTION

Reliability has long been important to many applications, including for long-running software that performs computationally expensive or critical tasks [1]. Indeed, a single failure may lead to the need to re-execute a large number of instructions, resulting in significant overhead, reduction in effective performance performance and increase in energy consumption. The lack of reliability can also have dire consequences for time-critical software programs that have to provide their results within a specific time frame [2]–[4], as well as for smaller applications that have to ensure energy efficiency of their operation [5]. For all such applications, fault tolerance mechanisms that avoid the complete program re-execution in case of failures have long been studied [6], [7].

In particular, the Checkpoint and Restart (CR) technique [8] is widely used to keep a sequence of safe copies (or checkpoints) of the program execution state so that the program may be restarted from a recent checkpoint so as to avoid re-executing the whole program in case of failures. Checkpointing schemes are also popular in high performance computing systems [9]–[12], and have also been implemented in operating systems such as Unix or Linux [13]–[15], and to insure the consistency of distributed systems [16]–[18]. Multiple level checkpointing introduced in [19], [20] was also recently studied in [21].

The Application-level Checkpoint and Restart (ALCR) technique [22], [23] is an efficient approach to checkpointing which requires a relatively small memory footprint which is limited to a single application. However, implementing checkpoints requires programming skills, as well as expertise in selecting the locations in the application where additional checkpointing code will be inserted, and a judicious choice of the checkpoint intervals. Existing ALCR tools and libraries facilitate the programming aspects by helping to insert checkpoints within ongoing loops [24], [25], but typically do not offer guidelines to determine the interval between checkpoints. Nevertheless checkpoint optimization applications using ALCR have been recently studied [26], [27].

While placing checkpoints increases the overhead for both execution time and energy for program execution, it can help save both time and power consumption during the recovery from when failures. This trade-off has been studied in several papers over the years using analytic models for transaction processing systems [28]–[30] and techniques that analyze a system’s robustness under failures or attacks [31]–[34].

The checkpoint interval has often been used to maximize a system’s availability for transaction processing systems or databases [35]–[39]. In addition, energy consumption minimization for computer systems is now recognized as a valid issue [40]–[44] with regard to the sustainability of information technology [45], [46]. However despite the importance of energy consumption in systems for reasons of economic cost and sustainability [47], [48], less work has been undertaken on judicious checkpointing to reduce the energy that is consumed by a system [49].

In this paper we take a common approach to checkpointing so as to save both the energy consumed by a program and its effective execution time. Using first principles, we construct a probability model that predicts the total average execution plus energy consumption of a program. Turning specifically to the case of checkpointing with ALCR, we compute the corresponding quantities for a program built around a long repetitive loop. The analysis offers a way to compute the optimum checkpoint interval that minimizes the overall average cost, which we illustrate with several examples. Finally we present a software tool to implement these results, allowing the end user to select checkpoint intervals for ALCR equipped programs, without having to delve deeply into the underlying theory.
II. A SINGLE LOOP PROGRAM WITH CHECKPOINTS

Suppose that some program runs \( y_n \) instructions starting from its \((n-1)\)th until its \(n\)th checkpoint. Here \( y_n \) does not include the repetitions of instruction execution due to failures, or the additional code executed for recovering from failures. At the instant \( t_n > 0 \) when the program creates its \( n \)-th checkpoint let \( Y_n \) be the total number of instructions that the program up to \( t_n \) and \( Y_n \) does not include the instructions that were repeatedly executed due to checkpoints and failures, so that \( Y_n = \sum_{i=1}^{n} y_i \).

If \( B^r(Y_n) \) is the time during needed for the creation of the \( n \)-th checkpoint, which depends on memory occupied the program occupies, and can also depend on \( Y_n \) due to data generated by the program, then \( B^r(Y_n) = B_0^r + B_1^r Y_n \) with constants \( B_0^r > 0 \) and \( B_1^r \geq 0 \). When a failure occurs after the program has completed \( y \) instructions following a checkpoint, \( b^r(Y_n, y) \) is the time required to start the program from the most recent checkpoint.

Thus the program successfully executes \( y \leq Y_n+1 - Y_n \) instructions after the most recent checkpoint and before the \((n+1)\)th checkpoint, where \( b^r(Y_n, y) = b_0^r + b_1^r y \) with constants \( b_0^r > 0 \) and \( b_1^r \geq 0 \), depending on the total number of instructions executed since the last checkpoint, i.e.

- The time \( B^c(Y_n) \) that is needed for establishing the \( n \)-th checkpoint will depend on the total number of instructions \( Y_n \) executed since the program was launched, where \( B^c(Y_n) = B_0^c + B_1^c Y_n \),
- While the duration \( b^c(Y_n, y) \) that is needed for failure recovery prior to the \( n \)th checkpoint, including related reloading the state of the system after a failure, depends on \( y \leq Y_n+1 - Y_n \), where \( b^c(Y_n, y) = b_0^c+b_1^c y \).

If the energy consumption for creating the \( n \)th checkpoint is \( B^r(Y_n) \), while \( b^r(y) \) is the consumed energy for failure recovery when the total executed instructions are \( Y_n + y \), we have \( B^c(Y_n) = B_0^c + B_1^c Y_n \) where \( b^c(y) = b_0^c + b_1^c y \) and \( B_0^c > 0 \), \( B_1^c \geq 0 \), \( b_0^c > 0 \), \( b_1^c \geq 0 \).

Calling \( \alpha, \beta > 0 \) the positive constants representing the relative cost of computation and energy, we characterize the joint total cost of time and energy through the quantities:

\[
\begin{align*}
B_0(Y) &= \alpha B_0^c(Y) + \beta B_0^r(Y), \\
B_1(Y) &= \alpha B_1^c(Y) + \beta B_1^r(Y), \\
b_0 &= \alpha b_0^c + \beta b_0^r, \\
b_1 &= \alpha b_1^c + \beta b_1^r,
\end{align*}
\]

\( c = \alpha c^c + \beta c^r \).

III. FIXED CHECKPOINT INTERVALS

Age dependent checkpoints [39] have been shown to reduce the overall cost of checkpointing and failure recovery, when the failure probability can increase with time, practical checkpointing generally makes the simple choice of periodically carrying out a checkpoint each time the program successfully executes a fixed number of instructions \( y_n = y \). We will now proceed in this manner, and assume that checkpoints are installed after \( Y_1 = y, Y_2 = 2y, \ldots, Y_n = ny \) instructions are executed. We then need to select the optimum \( y \), or the optimum \( n \), that minimizes the composite cost that combines time and energy. If the program ends when \( Y = Ny \) instructions are successfully executed, we do not need the \((N+1)\)th checkpoint, and the first checkpoint is installed prior to executing the first instruction.

We formulate our problem for a program that executes \( Y \) successfully, and we select \( y \) or \( N \), so that the total overhead in computation and energy consumption due to failure recovery and checkpoints is minimized.

\( C^c(y) \), the total expected execution time including all restarts due to failures, starting from the most recent checkpoint, for an average execution time per instruction \( c \) and failure probability per instruction \((1-a)\), is then:

\[
C^c(y) = c y^a + (b_0 + C^c(y))(1-a^y),
\]

\[
+ (c^e + b_1^e) \sum_{x=1}^{y} x a^{x-1} (1-a).
\]

Indeed, with probability \( a^y \) a failure does not occur during the \( y \) instructions, leading to an execution time of \( c^e \) time units. However, with probability \( (1-a^y) \) one failure or more will occur in the \( y \) instructions. The first failure that occurs provokes the restart of the program which takes time \( b_0^c \), and we add to it the time \( C^c(y) \) which includes the needed to restart and re-execute the program to execute \( y \) instructions after possible future failures.

Including the execution time and additional work needed per executed instruction until occurrence of the next failure, introduces the term \( (c^e + b_1^e) \) multiplied by \( x \) and the probability that the failure occurs at instruction \( x \), i.e. \( a^{x-1} a \), the whole being summed over \( x \) between 1 to \( y \). Using

\[
\sum_{x=0}^{y} a^x = \frac{1-a^{y+1}}{1-a},
\]

and

\[
d\frac{1-a^{y+1}}{1-a} = \frac{1-ya^y(1-a)-a^y}{(1-a)^2},
\]

results in:

\[
C^c(y) = b_0^c y^{a-1} + c^e + b_1^e y^{a-1} - b_1^e y. \tag{2}
\]

The total expected energy consumption \( C^e(y) \) for executing a number of instructions \( y \) after the most recent checkpoint, is also obtained in the same manner:

\[
C^e(y) = b_0^e y^{a-1} + c^e + b_1^e y^{a-1} - b_1^e y, \tag{3}
\]

where \( c^e \) is the average energy consumption per executed instruction. Therefore:

\[
C(y) = \alpha C^c(y) + \beta C^e(y),
\]

\[
= b_0[a^{y-1}] + \frac{c + b_1}{1-a} [a^{-y} - 1] - b_1 y. \tag{4}
\]

IV. THE TOTAL COST

When we include both the time and energy needed to create each checkpoint, and assuming a fixed number of instructions \( y \) executed between successive checkpoints, we can obtain
the total cost of the program up to and including the last instruction executed at \( Y = yN \) as:

\[
K_N(y) = NB_0 + \sum_{i=1}^{N} iyB_1 + NC(y),
\]

\[
= N[B_0 + C(y)] + \frac{N(N + 1)}{2}yB_1.
\]

The optimum checkpoint interval \( y^* \) is then the value of \( y \) that minimises \( \kappa_N(y) \), the cost per unit of work that is accomplished, i.e. \( K_N(y) \) divided by \( Y = Ny \) which is the total number of useful instructions executed for this total cost:

\[
\kappa_N(y) = \frac{K_N(y)}{Ny},
\]

\[
= \frac{B_0 + C(y)}{y} + \frac{(N + 1)B_1}{2},
\]

\[
= \frac{B_0 + C(y) + \frac{B_1y}{2}}{y} + \frac{B_1}{2},
\]

so that to examine the sign of \( C''(y^*) \) we obtain:

\[
C''(y) = a^{-\gamma}[\ln a]^2[b_0 + \frac{c + b_1}{1 - a}] - a^{-\gamma} \ln a \cdot \frac{c + b_1}{(1 - a)^2}. \tag{15}
\]

We see that \( C''(y) > 0 \), so that \( \kappa_N(y) \) is minimized for \( y = y^* \).

V. PROGRAMS WITH A SINGLE OUTER LOOP

When software that manages a set of sensors and actuators, it will process data from all sensors, update some variables, and provide data to actuators and/or forward the data to some cloud server. This sequence may repeat indefinitely in this program with one outer loop. Thus the theoretical results we obtain can be applied to a program with an outer loop with \( L \) instructions, which is executed \( T \) times with \( Y = LT \). Depending on \( y^* \), we may be constrained to place checkpoints at the start of a loop or at every few starts of a loop, or several checkpoints may be inserted in each loop.

Thus we will first apply the results of Section III to compute the optimum number of instructions executed between successive checkpoints \( y^* \) that minimize a weighted sum of execution time and energy consumption, as indicated in the expression (12) where we write:

\[
B = B_0 + B_1 L T, \quad A = b_0 + \frac{c + \frac{b_1}{1 - a}}{10^4}. \tag{16}
\]

Let \( I(x) \) be the integer that is closest to the real number \( x \). Then we obtain the related value:

\[
n = I\left(\frac{L}{y} \right), \quad i\left(\frac{y}{L} \right) \geq 1, \tag{17}
\]

\[
= I\left(\frac{L}{y} \right), \quad i\left(\frac{y}{L} \right) > 1.
\]

We will present examples in order to show the effect of \( Y \) and hence \( n \) on the expected execution time and the total energy consumption of a program that runs in the presence of failures.

To differentiate the computation time from energy consumption, \( n = n^t \) will represent the value that minimizes the total computation time, while \( n = n^e \) will refer to the value minimizing energy consumption. Thus in (17), we have \( n^o \) for \( \alpha = 1, \beta = 0 \), while \( n^+ \) corresponds to \( \alpha = 0, \beta = 1 \).

We consider a program with a single loop with checkpoints placed at the beginning (or at the end) of the loop. The parameters used are:

\[
B_0^0 = 500, \quad B_1^0 = 0, \quad B_0^c = 10^7, \quad B_1^c = 0, \quad c^c = 1
\]

\[
c^e = 10^{-5}, \quad b_0^0 = 100, b_1^0 = 10, \quad b_0^c = 100, b_1^c = 10
\]

\[
g = 5 \times 10^{-6}, \quad L = 100, \quad N = 10^{-4}.
\]

In Figure 1, a short program is taken with \( Y = 10^4 \) is considered. Its expected execution time is shown as a function of \( n \).
Fig. 1. Variation of the expected execution time with the number of checkpoints that are placed in a program with the total number of instructions executed being $Y = 10^4$.

We define the *Gain* as the ratio of the expected execution time to the number of checkpoints that are placed in a program with the total number of instructions executed being $Y = 10^4$.

In Figure 2 shows the expected *Gain* in terms of expected execution time for different values of $n$. The values that correspond to the optimum checkpoint interval $n^*$ are marked within a rectangle. It illustrates the fact that the optimum checkpoint interval $n^*$ minimizes the overall execution time of the application and maximizes the overall expected Gain.

The relationship between $n^*$ and $n^+$, the optimum checkpoint interval that minimizes computation time and energy, are investigated in Figure ??$. In Figure ??$ we see how the execution time changes when we the use optimal checkpoint interval calculated for energy consumption. Similarly Figure ??$ shows how energy consumption changes when we use the checkpoint interval that optimizes execution time.

VI. THE OPTIMUM CHECKPOINTING TOOL

In addition to the theoretical work of solving the checkpointing problem analytically, it has proved very useful to build a software tool that can help system designers, who are not performance modeling experts, to install checkpoints in a way that minimize energy consumption and/or program execution time, by a judicious choice of the checkpoint interval.

Thus we have developed software consisting of a Dashboard front-end and back-end, that provides analysis, predictions and recommendations, in the form of micro-services written in Java and built into Docker Containers [50]. The recommendations can be triggered by the user, or retrieved from prior examples in a database built using MongoDB [51].

A graphical interface, which we call the “Dashboard”, presents itself to the user and displays the analysis in the form of several display pages. This interface is based on the JavaScript library React [52] with the free user interface KIT [53] for building responsive websites and applications [53].

One of these components is the Dependability Toolbox which consists of three key features. Two of them: (i) Quantitative Security Assessment and (ii) Vulnerability Prediction, are responsible for maintaining the security of the applications. The third is responsible for maintaining the reliability of applications and is based on the assessing (iii) The optimal checkpoint interval between successive checkpoints. The calculations and predictions are triggered by the user, or can be retrieved from the database built on top of MongoDB.

One of the pages presents the results for the optimal checkpoint interval for a given program in terms of execution time and energy consumption. It is done by identifying the most demanding loop in the program and recommending the optimal interval between checkpoints to provide the lowest possible execution time or energy consumption. Calculations can also be done concerning both of them at the same time with user specified by relative importance.

The user of platform will first login to her account and choose the project from the projects page. Next, she can go to the proper page presenting the optimal checkpoint interval for her application as presented in Figure 3. It is possible for the user to see the name of the program and to perform the analysis using the proper button if there have been changes in the program. The date of the last analysis is also shown on the dash page.
After the data is taken from the new analysis or the database, the user can see the results divided into two sections. One of them presents the outcome for execution time in Figure 4, and the other for energy consumption in Figure 5. Both of them share a similar template. It presents the relevant data to the user the in the form of the plot, from which the user can see how the specified metric, the average computation time and the average energy consumption of the program varies concerning the number of loop repetitions between checkpoints. This is powered by the Plotly library and is located on the left of the section. To the right, there is shown to us two smaller sections. They present the minimum value from the set and also the corresponding value of the distance between checkpoints in the form of loop repetitions. Below we can also see the detailed table of the values if the user prefers to choose differently than optimal value.

To perform the analysis we first need provide the back-end of the dependability toolbox with specific parameters in the form of the JSON shown in Listing 1 through the available API.

Listing 1. Request body example

```javascript
{
   "ProgramType": "OptimalCheckpoints",
   "g": 0.000005,
   "B0e": 0.00000059,
   "B0c": 0.00000347,
   "L": 2826.0,
   "ce": 0.000000000445,
   "cc": 0.000000000074231,
   "b0c": 0.00000000077,
   "b1c": 0.000000000007,
   "b0e": 0.0000000367,
   "ble": 0.0000000000367,
   "N": 200.0,
   "alfa": 0.0,
   "beta": 1.0,
   "Ble": 0.0,
   "Blc": 0.0,
   "Y": 19782.0,
   "history_data": "1.0",
   "project_name": "Neurasmus",
   "username": "userName"
}
```

This consists mainly of parameters needed to perform the analysis for selecting the checkpoint interval of the program, and parameters that distinguish this particular program which may be selected from data stored in the database. After retrieving the data through the back-end the new analysis is performed. The outcome of the calculations is then transferred to the database to store the newest data, and exhibited on the particular dashboard page that is dedicated to optimal checkpoint interval analysis. Again the data exchanged between back-end and front-end are sent using the JSON format as presented in the Listing 2.

Listing 2. Response body example

```javascript
{
   "id": {
      "oid": "5f0ea3d06018001080cbcc"
   },
   "energyConsumptionTable": {
      "columns": [
         {
            "field": "y",
            "label": "Energy consumption"
         }
```  

VII. CONCLUSIONS

Checkpoints are widely used in databases, operating systems, high performance computing, and in long-running programs for real-time applications. They allow a system or a program to recover from failures without having to restart execution from scratch each time a failure occurs. However, checkpointing has costs both in additional time and energy, even when no failures occur. Indeed, the placement of checkpoints themselves consumes system resources.

Thus, this paper has analyzed the choice of optimum checkpoint intervals both from the perspective of energy costs and in terms of execution time, innovating with respect to previous work that has focused principally on execution time. Starting from first principles we have derived the optimum checkpoint for long running programs. We have also detailed the analysis for programs with a long running outer loop, which is common to many applications which deal with real-time systems. Explicit analytic results have been derived with closed form expressions and have been illustrated with numerical examples. We have also presented the design and operation of a tool that incorporates these results.

Future work will consider nested program structures, linking checkpointing to program structure in a useful and intuitive manner, similar to what was done in this paper for programs where a single loop is iterated a large number of times.

REFERENCES

Fig. 3. Accessing the Optimal Checkpoint Dashpage


Fig. 4. Part of page for execution time
Fig. 5. Part of page for energy consumption


TrimTuner: Efficient Optimization of Machine Learning Jobs in the Cloud via Sub-Sampling

Pedro Mendes¹, Maria Casimiro¹,², Paolo Romano¹, David Garlan²

¹INESC-ID and Instituto Superior Técnico, Universidade de Lisboa
²Institute for Software Research, Carnegie Mellon University

Abstract—This work introduces TrimTuner, the first system for optimizing machine learning jobs in the cloud to exploit sub-sampling techniques to reduce the cost of the optimization process, while keeping into account user-specified constraints. TrimTuner jointly optimizes the cloud and application-specific parameters and, unlike state of the art, eschews the need to train the model with the full training set every time a new configuration is sampled. Indeed, by leveraging sub-sampling techniques and data-sets that are up to $60 \times$ smaller than the original one, we show that TrimTuner can reduce the cost of the optimization process by up to $50 \times$.

Further, TrimTuner speeds-up the recommendation process by $65 \times$ with respect to state of the art techniques for hyper-parameter optimization that use sub-sampling techniques. The reasons for this improvement are twofold: i) a novel domain specific heuristic that reduces the number of configurations for which the acquisition function has to be evaluated; ii) the adoption of an ensemble of decision trees that enables boosting the speed of the recommendation process by one additional order of magnitude.

Index Terms—Machine Learning, Cloud optimization, Sub-sampling, Bayesian Optimization

I. INTRODUCTION

Training machine learning (ML) jobs on the cloud represents the de facto standard approach today for larges models. Existing ML models are in fact increasingly complex [1] and their training procedure sometimes involves an enormous amount of computational resources, which the cloud can provision in an elastic and on-demand fashion, sparing users and enterprises from colossal upfront capital investments. Further, by taking advantage of economies of scale, cloud providers can drastically reduce their internal operational costs. This ultimately reflects into cost savings for the end users.

Nonetheless, a key problem that users face is that modern cloud providers offer a large spectrum of heterogeneous virtual machine (VM) types, optimized for different types of resources and with different costs. For instance, at the time of writing, Amazon Web Services (AWS) EC2 offers approximately 300 different VM types [2]. The problem is further exacerbated by the fact that the (distributed) training process of ML jobs exposes several hyper-parameters — such as the batch size considered in each training iteration or the frequency of synchronization among workers. The optimal configuration for these parameters can be substantially affected by the choice of the type and number of provisioned cloud resources [3].

As such, end users who wish to train their ML models in cloud infrastructures are faced with a complex constrained optimization problem: determining which type/amount of cloud resources and model hyper-parameters to use in order to maximize the model’s accuracy while enforcing constraints on the maximum cost and/or time of the training process.

Given the complexity of modelling the dynamics of modern ML and cloud platforms via white-box methods, a common approach in the literature is to rely on black-box modelling and Bayesian Optimization (BO) techniques [3]–[5]. These techniques have the key advantage of requiring no prior knowledge of the target ML model to be optimized, and as such require the target ML model to be deployed and trained in several (cloud/hyper-parameter) configurations. The corresponding measurements of accuracy and execution成本/time are then used to build black-box models, e.g., Gaussian Processes (GPs). These models guide the optimization process by recommending which configurations to test next, balancing exploration (of unknown regions of the configuration space) and exploitation (of the models’ current knowledge) via different model-driven heuristics, a.k.a., acquisition functions [6].

Unfortunately, these systems suffer from a severe limitation: each time a configuration is tested, the target ML model needs to be trained on its entire data-set. As such, the optimization process can become prohibitively expensive (and slow) if the target model is meant to be trained on massive data-sets, as it is increasingly the case in practice [1].

In this work, we tackle this problem by presenting TrimTuner, the first system for optimizing the training of ML jobs in the cloud that exploits data sub-sampling techniques to enhance the efficiency of the optimization process.

TrimTuner considers the problem of identifying the cloud and hyper-parameter configuration that maximizes the accuracy of a ML model, while ensuring that user-defined constraints on the efficiency (e.g., cost or execution time) of the training process are complied with.

TrimTuner deploys the target job using sub-sampled data-sets (up to $60 \times$ smaller than the original one in our experiments) and constructs predictive models that keep into account how shifts of the data-set size affect both the quality (i.e., accuracy) and training efficiency (i.e., cost or execution time) of the target model. These models are then used within a novel acquisition function that aims at estimating the advantage of

This research was partially supported by FCT (POCI-01-0247-FEDER-045915 and UIDB/50021/2020) and NSA (Award No. H9823018D0008). The extensive evaluation presented was made possible due to the grant received from the AWS Cloud Credits for Research Program.
testing a new configuration $x$ using a data-set that is $s \times$ smaller than the original one ($s \in [0,1]$) by weighing in two main factors: (i) the expected information gain [7] that testing $(x,s)$ will yield on the configuration that maximizes accuracy when using the full data-set ($s=1$), and (ii) the likelihood that the latter configuration $(x,s=1)$ meets the user-specified constraints.

Overall, this paper makes contributions of both a methodological and practical nature. From a methodological perspective, TrimTuner builds on recent systems for hyper-parameter tuning of ML models [8]–[10], which have first investigated the use of sub-sampling techniques, and extends them in a number of ways: (i) by supporting the enforcement of additional user-specified constraints; (ii) by jointly optimizing the model’s hyper-parameters and the selection of the cloud configuration (number and type of virtual machines); (iii) by introducing a novel domain-specific heuristic, named “Constrained Expected Accuracy”, that accelerates the recommendation process by a factor of up to $2 \times$ when compared to state of the art approaches [11], [12]; (iv) by proposing the adoption of an ensemble [13] of Decision Trees (DTs) [14] as a light-weight alternative to Gaussian Processes (the de facto standard modelling approach for BO), which enables boosting the speed of the recommendation process by one additional order of magnitude.

From a practical perspective, when compared to state of the art BO-based systems for the (constrained) optimization of ML jobs, such as Lynceus [3] or Cherrypick [4], TrimTuner reduces the cost of the exploration process by up to $50 \times$ thanks to the use of sub-sampling techniques. Another practical contribution of this work is making available to the community the data-sets obtained for the evaluation of TrimTuner, which consider the training of three neural networks via TensorFlow on AWS EC2 over a search space (encompassing both model hyper-parameters and cloud-related parameters) composed of approximately 1400 configurations, whose collection incurred a cost of approximately $1200 and took about two months.

The remainder of this paper is structured as follows: in § II we describe related work; § III presents TrimTuner; § IV evaluates our contributions and finally § V concludes the paper.

II. RELATED WORK

This section discusses related works in the areas of cloud optimization and Bayesian optimization.

Cloud Optimization Approaches. Recent works to find the optimal configuration to deploy jobs in the cloud, such as CherryPick [4], Ernest [15], Paris [16] and Arrow [5], focus solely on optimizing cloud related parameters and disregard the possibility of learning from related tasks. Lynceus [3] was the first work to highlight the relevance of jointly optimizing the cloud configuration and the hyper-parameters affecting the distributed training process of ML models.

The key difference between TrimTuner and these systems is its reliance on sub-sampling techniques to reduce the cost of testing configurations during the optimization process. As we discuss in § IV, this allows TrimTuner to reduce the cost and duration of the optimization process by up to a factor of 50 and 65, respectively.

Another key aspect of TrimTuner, which differentiates it from systems like Paris [16], Quasar [17], or Arrow [5] is that it does not rely on a priori knowledge of similar types of jobs — whose representativeness constitutes a key assumption on which the accuracy of the optimization process hinges. Conversely, TrimTuner (analogously, e.g., to CherryPick and Lynceus) operates in a fully on-line fashion.

Bayesian Optimization. TrimTuner, similarly to other recent systems for the optimization of cloud jobs [3], [4], relies on a generic optimization methodology, known as (model-based) Bayesian Optimization (BO). BO has been adopted in a wide range of application domains including self-tuning of transactional memory systems [18], [19], databases [20] and of hyper-parameters of ML models [8], [10], [21].

BO aims to identify the optimum $x^*$ of an unknown function $f : \mathbb{X} \rightarrow \mathbb{R}$ and operates as follows [22]: (i) $f$ is evaluated (i.e., tested or sampled) over $N$ initial configurations, $x_i$, selected at random so as to build an initial training set $S$ composed of pairs $(x_i, f(x_i))$; (ii) $S$ is used to train a black-box model (typically a Gaussian Process [23]) that serves as a predictor/estimator of the unknown function $f$; (iii) an acquisition function, noted $\alpha$, is used to exploit the model’s knowledge (and related uncertainty) to determine which configuration to evaluate next by balancing exploitation of model’s knowledge — recommending configurations that the model deems to be optimal — and exploratory behaviours — recommending configurations whose knowledge can reduce the model’s uncertainty and enhance its accuracy; (iv) the process is iteratively repeated until a stopping condition is met, e.g., after a fixed budget is consumed or if the gains from further sampling are predicted to be marginal by the model (e.g., below a fixed threshold).

The definition of the acquisition function is arguably one of the most crucial components of BO methods and in the literature there exists a number of proposals. Expected Improvement (EI) is probably the most well-known acquisition function. As the name suggests, EI (Eq. (1)) uses the probability distribution of observing a value $f(x)$ at $x$, predicted by the model trained on data-set $S$, in order to measure the expected amount by which evaluating $f$ at $x$ can improve over the current best value or incumbent $\eta$:

$$\alpha_{EI}(x) = \int \max(0, f(x) - \eta)p(f(x)|S)df(x)$$  \hspace{1cm} (1)

Entropy Search (ES) [24] is an alternative acquisition function that chooses which configurations to evaluate by predicting the corresponding information gain on the optimum, rather than aiming to evaluate near the optimum (as in EI). ES (Eq. (2)) is based on the probability distribution $p_{opt}(x|S)$, namely the likelihood that a configuration $x$ belongs to the set of optimal configurations for $f$, given the current observations in $S$. The information gain deriving from testing $x$ is computed using the expected Kullback-Leibler divergence (relative entropy) between $p_{opt}(\cdot|S \cup \{x, y\})$ and the uniform
distribution $u(x)$, with expectations taken over the model-predicted probability of obtaining measurement $y$ at $x$:

$$ \alpha_{ES}(x) = \mathbb{E}_{p(y|x,s)} \left[ \int \frac{p_{opt}(x^\star | S \cup \{x, y\})}{p_{opt}(x^\star | S \cup \{x, y\})} \log \frac{p_{opt}(x^\star | S \cup \{x, y\})}{u(x^\star)} \, dx^\star \right] $$

(2)

Despite being numerically much more complex to compute than EI, ES allows for quantifying to what extent testing a configuration $x$ will give the model knowledge about the optimum $x^\star$, where generally $x \neq x^\star$. In the context of hyper-parameter tuning of ML models, this property of ES has been exploited by MTBO [21] and FABOLAS [8] to trade off the information gain and the cost (i.e., execution time) of training a ML model in a configuration $(x, s)$, where $x$ denotes a hyper-parameter’s configuration and $s \in [0, 1]$ the sub-sampling rate applied to the original/full model’s training data-set.

More precisely, in FABOLAS, the acquisition function (Eq. (3)) for $(x, s)$ is defined as the ratio between the information gain on the configuration that maximizes accuracy for the full data-set ($s=1$) and the (predicted) cost of training the model with a sub-sampling rate $s$ and hyper-parameters $x$:

$$ \alpha_F(x, s) = \mathbb{E}_{p(y|x,s)} \left[ \int \frac{p^{opt}_{x=1}(x^\star | S \cup \{x, s, y\})}{p^{opt}_{x=1}(x^\star | S \cup \{x, s, y\})} \log \frac{p^{opt}_{x=1}(x^\star | S \cup \{x, s, y\})}{u(x^\star)} \, dx^\star \right] \cdot \frac{1}{C(x, s)} $$

(3)

TrimTuner builds on these approaches and extends them in several ways. First, TrimTuner supports the definition of additional (independent) constraints, e.g., on cloud cost and/or execution times of training/querying the model. This is achieved by extending the acquisition function to factor in the probability that the new incumbent will comply with the constraints, which will be discovered after updating the model with the observation of a (possibly sub-sampled) configuration $(x, s)$. Unlike existing constrained versions of ES-based acquisition functions, such as Predictive Entropy Search with Constraints (PESC) [25] and constrained Max-value Entropy Search (cMES) [26], the proposed acquisition function does not make use of Bochner’s theorem for a spectral approximation, which allows TrimTuner to use GPs with non-stationary kernels (as in FABOLAS) or lightweight (ensembles of) decision trees. Further, TrimTuner jointly optimizes the hyper-parameters of the training process and the cloud configuration — which, as already mentioned, is crucial to maximize the cost efficiency of the recommended configuration [3].

Finally, the numerical computation of the ES (and of any acquisition function based on ES, like FABOLAS’ and TrimTuner’s) is onerous. TrimTuner introduces two mechanisms to accelerate the recommendation process: (i) a novel domain-specific heuristic, that is used to estimate which configurations are most likely to yield the highest values of the acquisition function — thus restricting the number of configurations for which the acquisition function is evaluated; (ii) differently from FABOLAS and MBTO, which rely on GPs to estimate the probability distribution of the accuracy and cost of unknown configurations, TrimTuner relies on an ensemble of decision trees that, as we will show in § IV, achieves comparable accuracy while enabling speed-ups of up to 14×.

III. TrimTuner

TrimTuner is a Bayesian Optimization based approach that jointly optimizes the configuration of the cloud platform over which a ML model is trained as well as the hyper-parameters of the training process. More formally, TrimTuner considers the following constrained optimization problem:

$$ \begin{align*}
    \text{maximize}_{x \in \mathbb{X}} & \quad A(x, s = 1) \\
    \text{subject to} & \quad q_1(x, s = 1) \geq 0, \ldots, q_m(x, s = 1) \geq 0
\end{align*} $$

(4)

where $x \in \mathbb{X}$ denotes the vector encoding both the cloud’s and hyper-parameters’ configuration, $s \in [0, 1]$ is the sub-sampling rate (relative to the full data-set), $A$ is the accuracy and $q_1 \ldots q_m$ is a set of Quality of Service (QoS) constraints on the target model (e.g., on the maximum duration/cost of its training process or on the latency for querying the resulting ML model). We assume that the accuracy and constraint functions are unknown, independent, and can only be observed via point-wise and noisy evaluations by training the ML model in the corresponding configuration. Note that both the objective function (i.e., model’s accuracy) and the constraints are expressed for configurations using the full data-set ($s=1$). However, in order to enhance the efficiency of the optimization process, TrimTuner tests configurations using sub-sampled data-sets ($s < 1$) and leverages the knowledge gained via these cheaper evaluations to recommend configurations that use the full data-set.

Algorithm 1 TrimTuner’s pseudo-code

1: function TrimTuner($M, X, Q$) 
   ▷ $M$: Model to be trained and full data-set 
   ▷ $X$: Set of possible cloud and hyper-parameters configurations 
   ▷ $Q$: Set of QoS constraints 
2: $T = \{ (x, s_i) : x \in X, i \in [1, k] \}$ — Set of untested configs. 
3: Select at random a configuration $x \in X$ 
4: for $i = 1, \ldots, k$ do 
   ▷ Test $x$ with $k$ sub-sampling rates 
5: $\langle a, c, q \rangle \leftarrow \text{Train } M \text{ in configuration } (x, s_i)$ 
6: $S^A \leftarrow S^A \cup \{x, s_i, a\}$ — Add accuracy of $(x, s_i)$ to $S^A$ 
7: $S^C \leftarrow S^C \cup \{x, s_i, c\}$ — Add cost of $(x, s_i)$ to $S^C$ 
8: $S^Q \leftarrow S^Q \cup \{x, s_i, q\}$ — Add QoS const. of $(x, s_i)$ to $S^Q$ 
9: $T \leftarrow T \setminus \{x, s_i\}$ — Remove $(x, s_i)$ from untested configs 
10: Fit models $A(x, s), C(x, s), Q(x, s)$ using $S^A, S^C, S^Q$, resp. 
11: for $(i = 1, \ldots, \text{MaxIterations})$ do 
   ▷ Select the most promising configurations using CEA 
12: $D \leftarrow \{ \beta \text{ configs } (x, s) \in T \text{ with max. CEA} \}$ 
13: $\langle x', s' \rangle \leftarrow \text{argmax}_{(x, s) \in D} \alpha_F(x, s)$ — Eq. 5 
14: $\langle a, c, q \rangle \leftarrow \text{Train } M \text{ in configuration } (x', s')$ 
15: $S^A \leftarrow S^A \cup \{x', s', a\}$ — Add accuracy of $(x', s')$ to $S^A$ 
16: $S^C \leftarrow S^C \cup \{x', s', c\}$ — Add cost of $(x', s')$ to $S^C$ 
17: $S^Q \leftarrow S^Q \cup \{x', s', q\}$ — Add QoS of $(x', s')$ to $S^Q$ 
18: $T \leftarrow T \setminus \{x', s'\}$ — Remove $(x', s')$ from untested configs 
19: Fit $A(x, s), C(x, s), Q(x, s)$ using $S^A, S^C, S^Q$, resp. 
20: New incumbent $x'$: feasible config. with max accuracy for $s = 1$, as predicted by the models $A(.)$ and $Q(.)$. 


Algorithm 1 provides the pseudo-code of TrimTuner. TrimTuner receives as input: (i) the ML model \( M \) whose training process has to be optimized along with its (full) training set; (ii) the set \( \mathbb{X} \) of possible cloud and hyper-parameter configurations; (iii) a set of QoS constraints \( \mathcal{Q} \).

**Initialization phase.** As in typical BO approaches, since no *a priori* knowledge is assumed on the target job, TrimTuner bootstraps its knowledge base via a random sampling strategy. More precisely, TrimTuner randomly selects a configuration\(^1\) \( x \in \mathbb{X} \) and tests it (i.e., trains the model) using different sub-sampling levels \( s_1, \ldots, s_k \).

We choose these sub-sampling levels so that they are biased towards configurations with small data-sets (thus reducing the cost/duration of the bootstrapping phase), while still gathering representative information on how variations of the sub-sampling rate \( s \) affect the objective and the constraint functions. Specifically, in the MNIST data-set (which will be evaluated in § IV), we consider \( s \in \{1/60, 1/10, 1/4, 1/2 \} \).

Note that since we test the same cloud/hyper-parameter configuration and only vary the sampling rate, we can test all the configurations \( (x, s_i) \) \((i \in [1, k])\) via a single training instance by taking a snapshot of the accuracy and QoS constraints whenever the sub-sampling rate \( s_i \) is achieved — this yields a cost equivalent to testing a single configuration using 50% of the model’s data-set.

Whenever a configuration \( (x, s) \) is tested, the corresponding accuracy, training cost (due to the use of cloud resources) and QoS constraints values are stored in the \( \mathcal{S}^A, \mathcal{S}^C \) and \( \mathcal{S}^Q \) data-sets, respectively.

The initialization phase ends by building distinct black-box models (see § III-A) that predict, for an untested configuration \( (x, s_i) \): (i) the accuracy of the target ML model, noted \( A(x, s_i) \), using \( \mathcal{S}^A \) (ii) the (cloud) cost of its training process, noted \( C(x, s_i) \), using \( \mathcal{S}^C \) and (iii) the value of each QoS constraint \( q_1 \ldots q_m \in \mathcal{Q} \), noted \( Q(x, s_i) \), using \( \mathcal{S}^Q \).

**Main optimization loop.** The optimization loop of TrimTuner relies on a novel acquisition function (Eq. (5)) to determine which configuration to test next:

\[
\alpha_T(x, s) = \mathbb{E}_{p(a|x,s,S)} \left[ \prod_{i \in \mathcal{Q}} p(q_i(x^*, s=1) \geq 0) | S \cup \{x, s, q, A\} \right] \cdot \frac{1}{C(x, s)} \cdot \mathbb{E}_{p(a|x,s,S)} \left[ \int_{p_{opt}^{x}}^{p_{opt}^x} \frac{d\mathcal{S}^A \cup \{x, s, a\}}{u(x^*)} \right]
\]

(5)

Conceptually, this acquisition function extends the one of FABOLAS (Eq. (3)) by additionally accounting for the probability that the new incumbent, \( x^* \), predicted by the models after having acquired knowledge on configuration \( (x, s) \) will meet the QoS constraints. The key challenge here is that the new incumbent that the models will predict after testing \( (x, s) \) is unknown at this point, since we are still reasoning on whether to test \( (x, s) \) or not.

We tackle this problem via a simulation approach that exploits the current model’s knowledge. Intuitively, when computing the acquisition function for \( (x, s) \), the following steps are executed for all possible values of accuracy and QoS constraints, \( (a, q) \):

1. Extend the accuracy and constraints data-sets (\( \mathcal{S}^A \) and \( \mathcal{S}^Q \)) with \( \{x, s, a\} \) and \( \{x, s, q\} \), respectively.
2. Train the models with the extended data-sets.
3. Identify the new incumbent \( x^* \) predicted using the updated models. This is the configuration \( (x, s = 1) \) that, based on the predictions of the updated accuracy and constraint models, achieves the largest accuracy among the ones that comply with all constraints.
4. Compute the probability that all constraints are met by the incumbent determined in the previous step, as the product of the probabilities (predicted by the updated models) that each constraint \( q_i \in \mathcal{Q} \) is respected (recall that constraints are assumed independent).

Finally, the expectation over all possible values of accuracy and QoS constraints \( (a, q) \) has to be computed. To this end, the models (prior to being updated) can be used to predict the probability of configuration \( (x, s) \) yielding \( (a, q) \) for its accuracy and constraints, respectively. The above expectation can be numerically approximated, e.g., using the Gauss-Hermite quadrature [28], which, roughly speaking, approximates the unbounded integral associated with the expectation (that should be computed over all possible values \( (a, q) \)) by discretizing the \( (a, q) \) space over a small number of predetermined “root” points. To limit the computational complexity of computing \( \alpha_T \), in TrimTuner we use a coarser, but cheaper, approximation, which conceptually coincides with using a single root in the GH quadrature: we simulate the testing of \( (x, s) \) by computing the above steps considering only for \( (a, q) \) the accuracy and QoS constraints values predicted by the corresponding models in \( (x, s) \).

Note that \( \alpha_T \) (see Eq. (5)) extends the acquisition function of FABOLAS (see Eq. (3)) in a modular way, in the sense that the second and third lines of Eq. (5), which correspond to FABOLAS’ acquisition function, can be numerically computed as discussed in FABOLAS’ paper.

Note also that, for efficiency reasons, TrimTuner (see Alg. 1 line 12) does not evaluate the acquisition function on every untested configuration \( (x, s) \), but only on a smaller sub-set (denoted \( \mathcal{D} \) in the pseudo-code) that is determined via a novel filtering heuristic, which we called Constrained Expected Accuracy (CEA) and that we describe in § III-B.

Once having determined which configuration \( (x', s') \in \mathcal{D} \) maximizes the acquisition function \( \alpha_T \), that configuration is tested (by training \( M \)) and the observed accuracy, cost and constraints’ values are stored in the corresponding data-sets.

Next, the models are updated to incorporate the knowledge obtained by testing \( (x', s') \) and the new incumbent is recommended. As already mentioned, the incumbent is selected

\(^1\)Testing more than a configuration provided no benefit in all the tests we performed, but TrimTuner supports testing a larger number of initial configurations, selecting them using Latin Hyper-Cube Sampling [27].
using the accuracy and constraint models as the configuration with full data-set \((s=1)\) that is predicted to be feasible (i.e., meet the constraints with high probability\(^2\)) and that achieves maximum accuracy.

TrimTuner adopts a simple stop condition that terminates the optimization after a fixed number of cycles. However, it would be relatively straightforward to incorporate more sophisticated, adaptive stop-conditions [18], [22] that, e.g., interrupt the optimization if the new predicted incumbent does not improve significantly over the best known optimum.

A. Models

To estimate the probability distribution of accuracy \((A)\), cost \((C)\) and constraints \((Q)\) for unexplored configurations \((x,s)\), TrimTuner relies on two black-box modelling techniques, namely Gaussian Processes and ensembles of Decision Trees. GPs. GPs [23] represent the de facto standard modelling approach in BO, due to their analytical tractability and flexibility [8], [22]. Key to the tractability of GPs is that the prediction that they generate follows, by construction, a Gaussian Distribution with known parameters. It is the possibility to define specialized kernels that provides flexibility to GPs, by allowing to imbue the model with domain-specific knowledge.

TrimTuner, analogously to FABOLAS, relies on kernels designed to capture the expected impact on cost and accuracy deriving from the use of sub-sampling. Specifically, we use a kernel obtained by the inner product of a “general purpose” Matérn 5/2 kernel and two custom kernels that encode, respectively, the expectation that accuracy and cost of a ML model grow normally with larger data-set sizes (see [8] for details).

Training GPs is notoriously an expensive process [22]. Evaluating the acquisition function of Eq. (5) and Eq. (3), requires re-training the models several times. Thus, as we shall see in § IV, the recommendation process can be extremely slow if a computationally expensive modelling technique, such as GPs, is used. This led us to explore an alternative approach based on DTs (more precisely Extra Trees [29]).

Ensemble of DTs. DTs are known for their high efficiency, but they can not be directly used to replace GPs since, unlike GPs, DTs do not provide a measure of uncertainty of their prediction. We circumvent this problem by using an ensemble of DTs and injecting diversity among the various learners by generating their data-sets drawing with replacement from the same data-set. We then estimate the probability distribution for a prediction as a Gaussian with mean and standard deviation derived from the predictions of the ensemble.

B. \(CEA\)

As mentioned, the numerical computation of TrimTuner’s acquisition function (and in general of ES-based acquisition functions) is very expensive, especially if GP models are used. The problem is further exacerbated since TrimTuner, differently from BO-based optimizers like FABOLAS, considers a configuration space that encompasses not only the model’s hyper-parameters, but also the cloud configuration. This results in an exponential growth of the search space and, as such, of the set of untested configurations for which the acquisition function should be evaluated.

The common approach, in the BO literature, to cope with this computational challenge, is to rely on generic search heuristics. These heuristics range from simple random sampling to sophisticated black-box optimizers [11], [12] and aim at reducing the number of configurations for which the acquisition function is evaluated. TrimTuner instead introduces a novel, domain-specific heuristic called Constrained Expected Accuracy (CEA) (Eq. (6)):

\[
CEA(x, s) = A(x, s) \cdot \prod_{q_i \in Q} p(q_i(x, s) \geq 0|S) \tag{6}
\]

Defined as the product of the predicted accuracy for a (possibly sub-sampled) configuration \((x, s)\) by the probability that \((x, s)\) satisfies the constraints, CEA can be seen as a rough, but cheap, approximation of \(\alpha_T\). In fact, while CEA directly estimates the quality of a (possibly sub-sampled) configuration \((x, s)\), \(\alpha_T\) predicts how much information the test of \((x, s)\) will disclose about \((x', s = 1)\), namely the optimal feasible configuration using the full data-set.

Due to its simplicity, CEA can be efficiently evaluated for every untested configuration, and TrimTuner uses it to rank and filter configurations: only the \(\beta \in [0,1]\) configurations with largest CEA are evaluated using \(\alpha_T\). We evaluate the sensitivity of TrimTuner to the \(\beta\) parameter in § IV-B.

IV. Evaluation

This section aims at addressing two main questions:

- Which cost and time savings does TrimTuner achieve vs existing BO-based approaches (§ IV-A)?
- How expensive is the recommendation process in TrimTuner and how effective is the CEA heuristic in accelerating it (§ IV-B)?

Data-sets. The data-sets used to evaluate TrimTuner were obtained by training three different neural networks (NN) in the AWS cloud: Convolutional Neural Network (CNN), Multilayer Perceptron (MLP), and Recurrent Neural Network (RNN). The networks were implemented using the Tensorflow framework [30] and trained on the MNIST database [31].

Each configuration (see Table I) is composed of cloud resources (number, type, and size of the virtual machines), application-specific parameters (batch size, learning rate, and training mode), and the size of the data-set used to train the NN. This results in a search space of 1440 configurations (288 configurations for each data-set size). To reduce noise in the measurements, we executed the training of the NN in each configuration three times. Whenever we train a model in a configuration \((x, s)\) we measure the achieved accuracy, training time and cloud cost. Collecting these data-sets took more than 2 months and costed a total of about 1200 USD. We have made these data-sets publicly available\(^3\), along with the implementations of TrimTuner and of the considered baselines.

\(^2\)We set this probability to 90\% in TrimTuner.

\(^3\)https://github.com/pedrogbmendes/TrimTuner.git
limits the maximum training cost to be $0 within 5\%.

TABLE II: Number of feasible configurations with an accuracy highest accuracy for the full data-set.

<table>
<thead>
<tr>
<th>Neural Network</th>
<th>Feasible Configurations</th>
<th>Feasible Configurations with high accuracy</th>
</tr>
</thead>
<tbody>
<tr>
<td>RNN</td>
<td>178 (61.8%)</td>
<td>28 (9.72%)</td>
</tr>
<tr>
<td>MLP</td>
<td>161 (55.8%)</td>
<td>29 (10.07%)</td>
</tr>
<tr>
<td>CNN</td>
<td>111 (38.5%)</td>
<td>39 (13.54%)</td>
</tr>
</tbody>
</table>

Based on these data-sets, we define a QoS constraint that limits the maximum training cost to be $0.02, $0.06, and $0.1 for RNN, MLP, and CNN, respectively. Table II reports the number (percentage) of configurations that use the full data-set (s=1) and comply with the cost constraints, and the number of those whose accuracy is no more than 5\% lower than the accuracy of the feasible configuration with highest accuracy. Only around 10\% of the full data-set configurations are close to the optimum, which illustrates the non-triviality of the considered optimization problem.

Baselines. We compare TrimTuner using GPs and DTs against constrained Expected Improvement (Elc) and Elc/USD, two popular acquisition functions for constrained optimization problems that were used by two recent cloud optimizers (CherryPick [4] and Lynceus [3], resp.).

Elc extends EI (Eq. (1)) by factoring in the probability that the configuration being evaluated will meet the constraints. Elc/USD, in its turn, extends Elc by considering the trade off between the benefits stemming from an exploration (computed using Elc) and the exploration cost (estimated via a dedicated model). None of these techniques use sub-sampling. We use GPs as base models for both Elc and Elc/USD that were implemented using the George library [32] in Python. We include in the comparison also Fabolas [8], which uses sub-sampling but does not consider constraints, and a simple random approach. We used the publicly available standard implementation of Fabolas.

Experimental setup and evaluation. The reported results are the average of 10 independent runs. For all the compared systems we bootstrap the models using 4 initial samples. For TrimTuner and Fabolas, which use sub-sampling strategies, we select a cloud and hyper-parameter configuration uniformly at random, and test it over the considered 4 data-set sizes (Table I). For Elc and Elc/USD, which do not use sub-sampling, we sample 4 full data-set configurations using Latin Hypercube Sampling (LHS).

We set the maximum number of iterations to 44 for all optimizers. Unless otherwise stated, for both TrimTuner variants we use the CEA heuristic, setting the filtering rate $\beta$ to 10\%.

All the systems were implemented in Python3.6 and the simulations were deployed in a VM running Ubuntu 18.04 LTS with 32 cores and 8GB of memory, hosted in machines with an Intel Xeon Gold 6138 CPU and 64GB of memory.

Evaluation Metrics. To evaluate the systems, we use a metric which we named Constrained Accuracy (Accuracy$_C$), that penalizes recommended configurations that do not respect the cost constraint.

\[
\text{Accuracy}_C = \begin{cases} 
A(x, s), & \text{if } (x, s) \text{ is feasible} \\
A(x, s) \cdot \frac{C_{\max}(x, s)}{C(x, s)}, & \text{otherwise} 
\end{cases}
\]

It is easy to see that this metric imposes larger penalizations to configurations that violate the constraint by a larger extent.

A. Comparison with state of the art optimizers

Figure 1 evaluates the cost efficiency of the compared solutions by reporting the Accuracy$_C$ of the recommended incumbent as a function of the cost of the optimization process for the various networks. The plots clearly show that both TrimTuner variants achieve higher Accuracy$_C$ levels at a fraction of the cost of the other solutions. The reason underlying TrimTuner’s gains with respect to Elc and Elc/USD is the use of sub-sampling, whose benefits are clear both in the initialization stage (shown using dashed lines) and once the models are in use. The average cost of each exploration step with TrimTuner is approx. 10× and 2.4× smaller than with Elc and Elc/USD, respectively. This is explicable by considering that TrimTuner uses an average sub-sampling rate of approx. 1/4 whereas Elc and Elc/USD test using full data-sets.

As expected, FABOLAS is the worst performing solution in this constrained optimization problem (for which it is not designed): since FABOLAS does not keep into account constraints, although the configurations it recommended achieve high accuracy, they frequently violate the cost constraint. As for the cost efficiency of the optimization process for the two variants of TrimTuner, the plots do not highlight significant differences. DTs appear to be slightly more accurate than GPs in the considered data-sets, confirming that they can represent a solid (and as we will see shortly way more lightweight) modelling alternative to GPs.

Figure 2 provides another perspective to assess the gains achieved by TrimTuner w.r.t. the considered baselines by reporting the time (Figure 2a) and cost (Figure 2b) savings that TrimTuner (DT variant) achieves to identify a configuration whose accuracy is 90\% (or more) of the optimal (feasible) solution. We omit FABOLAS and Random from the plots, which perform poorly, to enhance visualization. Using Elc and Elc/USD, the optimization process lasts up to 65×/15× (resp.) and costs up to 50×/10× more.
the setting of the efficiency of this heuristic and the sensitivity of TrimTuner to

(rhs) see that the DT-based implementation of TrimTuner achieves

TrimTuner takes approximately 30% longer. It is interesting to

TrimTuner variants the CEA heuristic configured to select

considerations baselines through the use of sub-sampling.

obtaining significant cost and time reductions w.r.t. the con-

functional complexity of the considered solutions. As expected,

FABOLAS, which also relies on GP-based models, takes

approximately one order of magnitude longer to output a

recommendation (≈14 minutes) and the GP-based variant of

TrimTuner takes approximately 30% longer. It is interesting to

see that the DT-based implementation of TrimTuner achieves

a 13× speed up compared to the GP variant, attaining a

performance almost on par with Elic and Elic/USD.

To sum up, TrimTuner can recommend configurations that

achieve high accuracy and meet the cost constraint, while

obtaining significant cost and time reductions w.r.t. the con-

sidered baselines through the use of sub-sampling.

B. Sensitivity to Filtering Heuristic

The data reported in Table III was obtained by using in both

TrimTuner variants the CEA heuristic configured to select 10%

(β) of the untested configurations. This section investigates the

efficiency of this heuristic and the sensitivity of TrimTuner to

the setting of the β parameter.

We start by comparing, in Figure 3, the AccuracyC that

is achieved in RNN when using TrimTuner with GPs and

the following heuristics: CEA, two state of the art black-box

optimizers (used, e.g., in FABOLAS), namely Direct [12] and

CMAES [11] and a simple random policy. For all the heuristics

we set the filtering level (β) to 10% and treat the optimization
cost as the independent variable.

The plot confirms the cost-efficiency of the proposed heuristic:

considering the cost spent to recommend a feasible configura-
tion whose accuracy is 90% (or more) of the optimum,

CEA achieves a 3.62× and a 7× savings when compared to

CMAES and Direct, respectively.

Table IV allows to evaluate the computational efficiency of

CEA by comparing the average time to recommend the next

configuration using TrimTuner (both variants) with different

heuristics and filtering levels (including no filter). We start

by observing that when considering a 10% filtering level,

recommending configurations with CEA takes roughly as long

as with Random. Note that with a random policy, the time
taken to recommend a configuration is dominated by the

evaluation of the acquisition function (for the configurations

selected by the filtering heuristic). This data confirms that CEA

is, indeed, a lightweight filtering heuristic, that is actually more

computationally efficient than Direct and CMAES (by up to

approximately 2×).

Finally, in Figure 4, we report the results of a sensitivity

study on the impact of tuning the filtering level (β) with CEA.

As expected, the best results are achieved when the no filtering

heuristic is employed. However, this comes at a very high

computational cost (see Table IV): if no filtering heuristic is

used, it takes on average about two hours to recommend a

configuration for the GP variant of TrimTuner; with DTs, that
time reduces to approximately 4 minutes, but still remains

quite large. Overall, these experimental data confirm the

relevance of developing effective filtering heuristics.

As for the sensitivity to the tuning of β, clearly the smaller

the number of configurations that the heuristic can select, the

TABLE III: Average time to recommend a configuration

(average of the three data-sets).

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Avg. time to recommend a configuration [min]</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrimTuner (GPs)</td>
<td>18.65</td>
<td>2.31</td>
</tr>
<tr>
<td>TrimTuner (DTs)</td>
<td>1.36</td>
<td>0.28</td>
</tr>
<tr>
<td>Fabolas</td>
<td>13.96</td>
<td>1.88</td>
</tr>
<tr>
<td>Elic (or Elic/USD)</td>
<td>1.17</td>
<td>0.07</td>
</tr>
</tbody>
</table>

Table III reports the average time to recommend the next configu-
rational complexity of the considered solutions. As expected,

Elic and Elic/USD, which use the simplest acquisition func-
tions, although with GPs, have the lowest computational cost.

FABOLAS, which also relies on GP-based models, takes

approximately one order of magnitude longer to output a

recommendation (≈14 minutes) and the GP-based variant of

TrimTuner takes approximately 30% longer. It is interesting to

see that the DT-based implementation of TrimTuner achieves

a 13× speed up compared to the GP variant, attaining a

performance almost on par with Elic and Elic/USD.

To sum up, TrimTuner can recommend configurations that

achieve high accuracy and meet the cost constraint, while

obtaining significant cost and time reductions w.r.t. the con-

sidered baselines through the use of sub-sampling.

B. Sensitivity to Filtering Heuristic

The data reported in Table III was obtained by using in both

TrimTuner variants the CEA heuristic configured to select 10%

(β) of the untested configurations. This section investigates the

efficiency of this heuristic and the sensitivity of TrimTuner to

the setting of the β parameter.

We start by comparing, in Figure 3, the AccuracyC that

is achieved in RNN when using TrimTuner with GPs and

the following heuristics: CEA, two state of the art black-box

optimizers (used, e.g., in FABOLAS), namely Direct [12] and

CMAES [11] and a simple random policy. For all the heuristics

we set the filtering level (β) to 10% and treat the optimization
cost as the independent variable.

The plot confirms the cost-efficiency of the proposed heuristic:

considering the cost spent to recommend a feasible configura-
tion whose accuracy is 90% (or more) of the optimum,

CEA achieves a 3.62× and a 7× savings when compared to

CMAES and Direct, respectively.

Table IV allows to evaluate the computational efficiency of

CEA by comparing the average time to recommend the next

configuration using TrimTuner (both variants) with different

heuristics and filtering levels (including no filter). We start

by observing that when considering a 10% filtering level,

recommending configurations with CEA takes roughly as long

as with Random. Note that with a random policy, the time
taken to recommend a configuration is dominated by the

evaluation of the acquisition function (for the configurations

selected by the filtering heuristic). This data confirms that CEA

is, indeed, a lightweight filtering heuristic, that is actually more

computationally efficient than Direct and CMAES (by up to

approximately 2×).

Finally, in Figure 4, we report the results of a sensitivity

study on the impact of tuning the filtering level (β) with CEA.

As expected, the best results are achieved when the no filtering

heuristic is employed. However, this comes at a very high

computational cost (see Table IV): if no filtering heuristic is

used, it takes on average about two hours to recommend a

configuration for the GP variant of TrimTuner; with DTs, that
time reduces to approximately 4 minutes, but still remains

quite large. Overall, these experimental data confirm the

relevance of developing effective filtering heuristics.

As for the sensitivity to the tuning of β, clearly the smaller

the number of configurations that the heuristic can select, the

TABLE III: Average time to recommend a configuration

(average of the three data-sets).

<table>
<thead>
<tr>
<th>Optimizer</th>
<th>Avg. time to recommend a configuration [min]</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>TrimTuner (GPs)</td>
<td>18.65</td>
<td>2.31</td>
</tr>
<tr>
<td>TrimTuner (DTs)</td>
<td>1.36</td>
<td>0.28</td>
</tr>
<tr>
<td>Fabolas</td>
<td>13.96</td>
<td>1.88</td>
</tr>
<tr>
<td>Elic (or Elic/USD)</td>
<td>1.17</td>
<td>0.07</td>
</tr>
</tbody>
</table>
worse the performance. Yet, we do not observe a significant degradation for values of β as low as 10%, which motivates the setting employed in the study presented in Section IV-A.

V. CONCLUSION

We presented TrimTuner, a system for optimizing the training of ML jobs in the cloud that exploits data sub-sampling techniques to enhance the efficiency of the optimization process. TrimTuner builds upon recent systems for hyperparameter tuning, which it extends by supporting user-defined constraints, and jointly optimizing cloud and model’s hyperparameters. TrimTuner relies on two methods to accelerate the recommendation process: a new heuristic, called Constrained Expected Accuracy; and the adoption of an ensemble of decision trees to model the cost and accuracy.

Thanks to sub-sampling, TrimTuner reduces the cost and latency of the exploration process by up to 50× and 65×, respectively, whereas the joint use of CEA and decision trees accelerates the recommendation process by up to 117×.

In future work, we plan to extend TrimTuner to cope with alternative optimization problems, e.g., multi-objective optimization of both cost and accuracy, and to evaluate it in problems with multiple constraints.

REFERENCES


TABLE IV: Average time to recommend the next configuration with different heuristics and filtering levels (RNN, TrimTuner).

<table>
<thead>
<tr>
<th>Filtering Heuristic</th>
<th>Recommendation time TrimTuner (GPs) [min]</th>
<th>Recommendation time TrimTuner (DTs) [min]</th>
</tr>
</thead>
<tbody>
<tr>
<td>No filter</td>
<td>125.76</td>
<td>3.69</td>
</tr>
<tr>
<td>CEA (1%)</td>
<td>5.94</td>
<td>1.07</td>
</tr>
<tr>
<td>CEA (10%)</td>
<td>16.85</td>
<td>1.72</td>
</tr>
<tr>
<td>CEA (20%)</td>
<td>28.65</td>
<td>2.05</td>
</tr>
<tr>
<td>Direct (10%)</td>
<td>36.18</td>
<td>2.63</td>
</tr>
<tr>
<td>CMAES (10%)</td>
<td>30.87</td>
<td>2.26</td>
</tr>
<tr>
<td>Random (10%)</td>
<td>16.53</td>
<td>1.62</td>
</tr>
</tbody>
</table>

Fig. 4: Sensitivity study on the β parameter for CEA (RNN, DT). The initialization phase does not use CEA and is omitted.

TABLE IV: Average time to recommend the next configuration with different heuristics and filtering levels (RNN, TrimTuner).
Abstract—Mobile communication systems need to adapt to temporally and spatially changing mobile network traffic, due to dynamic characteristics of mobile users, in order to provide high quality of service. Since these changes are not purely random, one can extract the deterministic portion and patterns from the observed network traffic to predict the future network traffic status. Such prediction can be utilized for a series of proactive network management procedures including coordinated beam management, beam activation/deactivation and load balancing.

To this end, in this paper, an intelligent predictor using artificial neural networks is proposed and compared with a baseline scheme that uses linear prediction. It is shown that the neural network scheme outperforms the baseline scheme for relatively balanced data traffic between highly random and deterministic mobility patterns. For highly random or deterministic mobility patterns, the performance of the two considered schemes is similar to each other.

Index Terms—Network traffic forecasting, mobility prediction, network load balancing, artificial neural networks, joint beam configuration.

I. INTRODUCTION

With a tremendous increase of the number of mobile users and their data demands, providing a high quality of service in mobile wireless communication systems becomes more challenging and critical. An important aspect for ensuring a high-quality service is to manage network resources efficiently. Due to the nature of mobile communications, users’ traffic demands undergo temporal and spatial changes. Thus, networks should be able to adapt to the changing environment over time due to dynamic mobile users. To make this adaptation uninterrupted and smooth, the network should not only observe the changes and then react but also foresee such dynamic behavior. Considering that these changes are not purely random, one can exploit the partly deterministic evolution of the changing network traffic resulting from users’ mobility. Extracting regularities from users’ mobility patterns and utilizing them to predict the future of the traffic demand is beneficial for guaranteeing an efficient and timely service.

In this study, we design a novel predictor using artificial neural networks to predict the future mobile network traffic status. We first divide the geographical network service area into smaller areas, referred to as pixels, and represent the aggregated mobile network traffic in terms of pixel densities. Then, predictors are used to forecast the mobile network traffic per pixel based on observations of the recent network traffic patterns. The prediction outcome can be utilized for a series of network management procedures including coordinated beam management, beam activation/deactivation and load balancing between cells or beams. For example, a portion of the beams could be scheduled more often for mobile network service areas which are predicted to be densely occupied by the mobile users in order to realize proactive load balancing. This objective is closely related with topics in the 3GPP standardization where an aim was to use network data in 5G New Radio (NR) to realize Self-Organizing Networks (SON) features including cell and beam based mobility optimization, load sharing and balancing related optimization, and coverage and capacity optimization [1].

Although spatio-temporal network traffic prediction using deep learning techniques has been already studied, prior works consider predictions only with much longer look ahead windows, e.g. sizes of 10 minutes, 30 minutes, 1 hour or even larger [2-6]. However, the predictions considered in this work pertain to shorter time intervals, of an order of seconds, thus facilitating network load balancing within a single or multiple cells. The choice of the time scale alters the problem significantly since data characteristics and randomness in user mobility would be considerably different. Additionally, we highlight the effect of deterministic versus random user movements in forecasting the network data traffic.

The analysis in this work demonstrates that the designed neural network based predictor outperforms linear predictors for various simulation scenarios and has significant advantages in terms of required observation amounts. In particular, it is shown that the neural network based predictor outperforms the linear predictor for relatively balanced mobile network traffic between purely random and deterministic mobility patterns. For highly random or highly deterministic mobility patterns, the performance of the neural network based predictor is similar to that of the linear predictor.

II. SYSTEM MODEL

A. Data Set

We use artificially generated simulation data which includes both individual user data sets and group users data sets mimicking a real world scenario well. The data sets include the mobility history, i.e., the location coordinates sequences of mobile users.

1) Scenario Layout

Individual user mobility data is generated by a dynamic system level simulator where mobile users are placed on the Madrid Grid layout described in the METIS (Mobile
Fig. 1: An instance of Madrid Grid Layout with street users, shown by red crosses, and pedestrians, shown by blue circles.

Fig. 2: Two user groups, shown by white and pink rectangles, moving on a 6x6 grid shown in 6 consecutive snapshots.

Fig. 3: User locations shown by blue circles for 4 consecutive time instants in the mobile network service area divided into 6x6 grid.

Fig. 4: Pixel values assigned to the snapshots based on number of users hosted in each pixel shown for 4 consecutive time instants.

and wireless communications Enablers for the Twenty-twenty Information Society) project documentation [7]. There, the Madrid Grid layout is advocated as a realistic urban environment model and an example of a typical European city environment. A visualization of the layout is shown in Fig. 1 depicting an instance of mobile users’ locations. The green boxes represent buildings while the regions outside the green boxes are streets or open pedestrian areas.

2) Mobility Model

Mobile users moving on the streets, referred to as street users, are highlighted by red crosses while the users walking within pedestrian areas, referred to as pedestrian users, are shown by blue circles in Fig. 1. Individual users are randomly dropped onto the scenario layout. Street users move along a randomly chosen direction until they arrive to a street connection point or an edge where they randomly choose a new direction/street out of all possible choices, while pedestrian users are simulated with a random walk model. The randomness of the individual movements is non-negligibly high, especially for longer observation windows, which makes accurate predictions difficult. It is noted that this mobility model is considered in the METIS project [7] and also in other more recent studies [8], and is furthermore based on the urban Manhattan mobility model considered in the ETSI (European Telecommunications Standards Institute) technical report [9].

The group user mobility data represents scenarios where the users move together as groups and thus reflects more deterministic movement patterns. Examples of such patterns occur when a group of people (e.g. tourists) move together to make a tour around attractive places or mobile users located within means of public transportation move thereby having a deterministic route. As simulation data, several groups of users are generated and assigned deterministic routes. A visualization of two user groups, depicted by white and pink rectangles, moving within a 6x6 grid shown for consecutive snapshots is provided in Fig. 2. The two data sets, i.e., the individual mobility data including relatively high randomness and the group user mobility data with highly deterministic nature can be combined to reflect a realistic scenario where
both kinds of mobility models are encountered.

B. Data Pre-processing

The proposed prediction method observes mobile network traffic patterns and exploits them to predict the future network load concentration in the network service area. Therefore, it is emphasized that in this work, the evolution of the aggregate per pixel network traffic load is of interest, rather than tracking individual users. The network service area is divided into pixels within a grid and each pixel is assigned a value representing the number of users it hosts.

An example for this process is shown in Figs. 3-4. In Fig. 3, the users located within the network service area are represented by blue circles for 4 consecutive time instants and the area of interest is divided into 6x6 pixels. Then, the total number of users within each pixel is counted and assigned to the pixel’s value as in Fig. 4. Afterwards, the group user data is added on top of the individual user data to obtain the final data set to be used for the prediction task. Following this pre-processing, the user concentration in each time instant is represented by a matrix representing “snapshots” (or “images”) of the mobile network traffic.

It should be noted that the total data rate consumed (by the users located) in each pixel can also be assigned as pixel values to represent the mobile network traffic. The proposed pre-processing and prediction algorithm can be applied to both representations, i.e., is applicable to both the data rate of the mobile users and the number of the mobile users. Furthermore, for the case of an equal data rate for all mobile users, the two representations would result in essentially the same pixel values, ignoring a constant factor.

C. Prediction Task

The objective of this study is to design a predictor which forecasts mobile network traffic based on observations capturing the mobile network traffic history. The prediction operation can be represented as

\[ \hat{X}_n = f(X_{n-1}, X_{n-2}, \ldots, X_{n-c}), \]

where \( f \) is the function realized by the predictor estimating the mobile network traffic matrix, \( X_n \), at time \( n \) based on the most recent \( c \) observations of the mobile network traffic. Our objective is to find a function \( f \), i.e., a predictor, which predicts the mobile network traffic as accurately as possible. For the sake of comparison, we evaluate the performance of both a linear predictor and the proposed neural network predictor.

1) Linear Predictor

Linear predictors are the most commonly used predictor type which we use as a benchmark. We represent the random process \( X_n \) for the design of a linear predictor as

\[ X_n = \sum_{i=1}^{P} \phi_i X_{n-i} + \epsilon_n, \]

where \( X_n \) denotes the mobile network traffic matrix at time \( n \), and \( \epsilon \) stands for an error matrix. The matrices \( \phi_i \) are the optimum coefficient matrices of the linear predictor and are estimated by using a least-square estimator.

2) Artificial Neural Network Predictors

We utilize three classes of artificial neural networks (ANNs), namely Multilayer Perceptrons (MLPs), i.e., fully connected feed-forward neural networks [10], Convolutional Neural Networks (CNNs) [11] and Long-Short Term Memory (LSTM) Neural Networks [12] for the design of the proposed ANN predictors. Here, CNNs are useful to exploit spatial local correlations especially in image-like data such as in the considered prediction task. On the other hand, LSTMs are commonly used for the prediction of time series of data, which is at hand for the described prediction task. In Section IV, the performance of the three adopted ANN predictors is evaluated.

The ANN predictors used in this work are structured as follows. 2-D snapshots of the mobile network traffic are organized into vectors for the training of the MLP and LSTM predictors, respectively. To train the considered MLP predictor, the snapshot vectors of an observed snapshot sequence are concatenated into a single vector in an ordered manner. Therefore, the size of the input layer of the MLP predictor depends on the number of snapshots observed prior to the next prediction output, referred to as the observation window length. For the CNN predictor training, each snapshot matrix in an observed snapshot sequence is fed to the CNN predictor as an input channel. The last convolutional layer is followed by a fully connected layer to produce the output. For the considered LSTM predictor, the length of the input sequence is set to the observation window length. The last LSTM layer is followed by a fully connected layer to generate the output. The neurons in the output layer of all three predictors deliver the pixel values of the predicted mobile network traffic snapshot in the next time instant.

Input and output size parameters of the ANN predictors are given in Table I, where \( P \) and \( ObsWinLen \) denote the pixel matrix dimension (e.g. 8 for an 8x8 image) of the snapshots and the observation window length, respectively. An example for the predictor parameters is given in Table II for

<table>
<thead>
<tr>
<th>Layer</th>
<th>MLP</th>
<th>CNN</th>
<th>LSTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of Input Layer</td>
<td>ObsWinLen x P^2</td>
<td>P^2</td>
<td>P^2</td>
</tr>
<tr>
<td>Size of Output Layer</td>
<td>P^2</td>
<td>P^2</td>
<td>P^2</td>
</tr>
<tr>
<td>Number of Input Channels</td>
<td>-</td>
<td>ObsWinLen</td>
<td>-</td>
</tr>
<tr>
<td>Input Sequence Length</td>
<td>-</td>
<td>-</td>
<td>ObsWinLen</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Layer</th>
<th>MLP</th>
<th>CNN</th>
<th>LSTM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of Input Layer</td>
<td>192</td>
<td>8x8</td>
<td>64</td>
</tr>
<tr>
<td>Size of Output Layer</td>
<td>64</td>
<td>64</td>
<td>64</td>
</tr>
<tr>
<td>Number of Input Channels</td>
<td>-</td>
<td>3</td>
<td>-</td>
</tr>
<tr>
<td>Input Sequence Length</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
</tbody>
</table>
TABLE III: Hyper-parameter optimization results of the MLP predictor for \( P = 8 \) and \( \text{ObsWinLen} = 3 \).

<table>
<thead>
<tr>
<th>No. Hidden Layers</th>
<th>No. Neurons in Hidden Layers</th>
<th>MSE</th>
<th>Proportion of Wrongly Predicted Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>128</td>
<td>0.09</td>
<td>0.22</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>0.06</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>0.06</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>1024</td>
<td>0.06</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>2048</td>
<td>0.08</td>
<td>0.20</td>
</tr>
<tr>
<td>1</td>
<td>64</td>
<td>0.11</td>
<td>0.23</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0.07</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>0.06</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>0.06</td>
<td>0.18</td>
</tr>
<tr>
<td>2</td>
<td>64</td>
<td>0.13</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0.08</td>
<td>0.20</td>
</tr>
</tbody>
</table>

TABLE IV: Hyper-parameter optimization results of the CNN predictor for \( P = 8 \) and \( \text{ObsWinLen} = 3 \).

<table>
<thead>
<tr>
<th>Kernel Size</th>
<th>No. Output Channels</th>
<th>Stride</th>
<th>Pooling Size</th>
<th>MSE</th>
<th>Proportion of Wrongly Predicted Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>0.06</td>
<td>0.17</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>0.05</td>
<td>0.16</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.09</td>
<td>0.22</td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>0.06</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.05</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.05</td>
<td>0.17</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>1</td>
<td>1</td>
<td>0.05</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0.05</td>
<td>0.17</td>
</tr>
</tbody>
</table>

TABLE V: Hyper-parameter optimization results of the LSTM predictor for \( P = 8 \) and \( \text{ObsWinLen} = 3 \).

<table>
<thead>
<tr>
<th>No. Hidden Layers</th>
<th>No. Neurons in Hidden Layers</th>
<th>MSE</th>
<th>Proportion of Wrongly Predicted Users</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>0.12</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>0.06</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0.06</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>0.07</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>0.07</td>
<td>0.19</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>0.11</td>
<td>0.25</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>0.06</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>0.07</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>256</td>
<td>0.08</td>
<td>0.19</td>
</tr>
<tr>
<td></td>
<td>512</td>
<td>0.08</td>
<td>0.20</td>
</tr>
</tbody>
</table>

The simulation setting with 8x8 snapshots and an observation window length of 3.

III. RESULTS

In this section, we give simulation results in order to compare the predicted and the true mobile network traffic for an evaluation of the performance of the linear predictors and the ANN predictors. Each item in the simulation data set consists of a set of consecutively observed normalized mobile network traffic snapshots (number of observed snapshots being determined by the observation window length) as input and the normalized traffic snapshot in the next time instant as reference (for the training phase) or target (during the test phase). The normalization is done by extracting the mean and dividing by the standard deviation. The test data set is different from the training data set to test the predictors with unseen data.

Two main evaluation metrics are considered. One evaluation metric is the mean-squared error (MSE) between the normalized predicted and the true network traffic snapshots. The second evaluation metric is the ratio of the number of wrongly predicted users (i.e., users whose location (host pixel) is wrongly predicted) to the total number of users. Simulations are conducted with a time step size of 2 seconds between the consecutively observed mobile network traffic snapshots. The network service area is divided into 8x8 pixels resulting in physical pixel dimensions of 52x72 meters. Note that the time step size and number of pixels used for subdivision can be modified by the designer depending on use cases and requirements.

A. ANN Hyper-parameter Optimization

The neural networks of the ANN predictors are trained by using the Adam optimizer [13]. An adaptive learning rate which decreases when learning converges is used. The MSE (mean-squared error) and ReLU (rectified linear unit) functions are chosen as the objective function and activation function in the hidden neurons, respectively, which are both well suited for the considered regression problem. Hyper-parameters related to the neural network structure are optimized using a grid search. While the proposed method can be applied to any selection of the pixel matrix dimension and the observation window length, the ANN hyper-parameter optimization should be done for each selection of those parameters since each selection leads to a different problem dimension, and therefore, a different optimal ANN structure.

In order to exemplify the ANN predictor optimization process, we select the pixel matrix dimension as 8, i.e., 8x8 matrices are considered, and the observation window length as 3, and perform a grid search to find the optimal hyper-parameter set for each predictor. Note that, due to space limitations, hyper-parameter optimization is presented only for the mentioned selection, and only the hyper-parameter sets which deliver relatively accurate predictions are displayed. The results of the grid search are given in Tables III, IV and V for MLP, CNN and LSTM predictors, respectively. The optimal hyper-parameter sets delivering the most accurate prediction, i.e., minimizing the MSE and the proportion of wrongly predicted users, are indicated by underlined bold symbols. It is observed for all three ANN predictors that the neural network structures having one hidden layer deliver the most accurate predictions, i.e., a further increase in the number of hidden layers did not improve the prediction accuracy.
Observation Window Length of Prediction

Predictors estimate the network traffic in the next instant based on observations of the recent traffic snapshots of the mobile network. A parameter which is worth to be investigated is the observation window length, i.e., the number of snapshots observed by the predictor prior to the next prediction instant. The observation window length does not relate to the time step length between consecutive snapshots but refers to the number of snapshots utilized by the predictor for each prediction output.

While a short observation window length can result in inaccurate predictions, longer observation windows imply larger required input data per prediction, as well as observations into the distant past and a problem of larger size. In Fig. 5, the prediction MSE is shown as a function of the observation window length. It can be deduced from Fig. 5 that the increase in prediction accuracy, i.e., decrease in MSE, becomes almost negligible beyond a sufficiently large number of observations. Considering the accuracy-complexity trade-off, it is appropriate to select an individual moderate observation window length for a given scenario. Furthermore, the CNN predictor yields the lowest prediction error, i.e., it delivers the most accurate predictions. Therefore, we select the CNN predictor and will refer to it as the proposed ANN predictor in the following.

Unlike fully connected neural networks, CNNs take advantage of proximity relations among image pixels, i.e., nearby pixels being more likely to be related to a given pixel than distant pixels. Since, each convolutional neuron processes data only from its receptive field, or field of view, CNNs benefit from a more compact representation, i.e., a smaller number of required parameters. Therefore, CNNs need fewer samples for training and are characterized by a reduced computational load. As a result, CNNs show an improved performance in image related applications, cf. Fig. 5.

Figure 6 shows the proportion of wrongly predicted users as a function of the observation window length. The main observation from Fig. 6 is that the ANN predictor clearly outperforms the linear predictor and reduces the number of wrongly predicted users. In addition, an important result is that the performance of the linear predictor gets close to that of the ANN predictor as the observation window length becomes larger. However, the ANN predictor can accurately predict the mobile network traffic using a much shorter observation window. The opportunity of using less input information can be of significant benefit in real scenarios. Considering that, in real world applications, data might be partially unavailable, transferring data can increase the signaling overhead; also if computational complexity is of interest, using ANN predictors would be favorable to employing linear predictors.

Group User Percentage

The ratio of the number of group users to the number of all users determines the considered scenario and the level of randomness/determinism associated to the movement patterns. It should be noted that while individual users are subject to a highly random mobility model, group users show a more deterministic behavior as described in Section II. The randomness/determinism level can be adjusted by weighting each of both cases accordingly in order to represent various realistic scenarios.

Figure 7 shows the prediction accuracy of the considered predictors in terms of the proportion of wrongly predicted users, i.e., the ratio of the number of users whose host pixel is wrongly predicted to the number of all users, as a function of percentage of group users, i.e., the ratio of the number of group users to the number of all users on a percentage basis. To highlight the comparison of the performance of the predictors, Fig. 8 illustrates the performance gain of the ANN predictor over the linear predictor which is measured by the ratio of the linear predictor’s MSE, $MSE_{LP}$, to that of the ANN predictor, $MSE_{ANN}$ as a function of the percentage of group users in the scenario.

In the most important scenario of interest, there is a balanced combination of the individual users and the group users.
Fig. 7: Proportion of users predicted in a wrong pixel vs. group user percentage (the ratio of the number of group users to the number of all users on a percentage basis). A time step size of 2 seconds between consecutive network traffic snapshots composed of 8x8 pixels is used.

Fig. 8: Performance (MSE) gain of the ANN predictor compared to the linear predictor vs. group user percentage (the ratio of the number of group users to the number of all users on a percentage basis). A time step size of 2 seconds between consecutive network traffic snapshots composed of 8x8 pixels is used.

users, reflecting the situation in most practical applications. In this case, the effects of both highly random individual user movements and highly deterministic group user movements are present in the data set. This scenario is represented by moderate group user percentage values (e.g. $20 - 40 - 60\%$) on the horizontal axis of Figs. 7-8.

A clear performance gain of the ANN predictor compared to the linear predictor can be observed from Fig. 8 for the mentioned range for the group user percentage. Therefore, using the ANN predictor results in more accurate predictions compared to the linear predictor. Another observation, as partially mentioned in Section III-B, is that the observation window length has an effect on the performance gain. That is, a shorter observation window length results in a higher performance gain of the ANN predictor over the linear predictor. Additionally, from Figs. 7-8, it can be seen that the linear predictor performs poorly when the observation window length is smaller than or equal to three.

On the other hand, the performance of the ANN predictor and the linear predictor are similar for scenarios with extreme domination of either individual users or group users. In other words, in highly random or deterministic scenarios, the ANN predictor yields a performance similar to that of the linear predictor. It can be observed from Fig. 8 that the performance gain values are close to one at the two extreme ends of the horizontal axis. For the scenario highly dominated by individual users, where the horizontal axis value in Figs. 7-8 is around 0%, the data set includes a high randomness where there are not much patterns to be exploits for the prediction by nature of the data. This leads to the conclusion that even using a more sophisticated predictor does not yield a performance gain over the linear predictor. This difficulty in prediction can be observed also in Fig. 7 where for group user percentages close to zero, prediction errors have relatively higher values. For the scenario highly dominated by group users, where the horizontal axis value in Figs. 7-8 is around $80 - 100\%$, the data set reflects high determinism and thus results in a relatively simple prediction problem to solve. Consequently, even a simple linear predictor can deliver accurate predictions without the need for further intelligence. The simplicity of the problem can be seen in Fig. 7 according to which both predictors have indeed very high accuracy with negligible performance differences for the mentioned range.

It is worth to be mentioned here that the optimum neural network structure and the results presented depend on the adopted mobility model. The mobility model employed in this paper is widely used in standardization and large-scale research projects as explained in Section II and, thereby, can be expected to reflect real world scenarios sufficiently well.

IV. CONCLUSIONS

A predictor employing artificial neural networks for prediction of network traffic in mobile networks has been proposed and studied. Such predictions can be utilized for a series of proactive network management procedures including coordinated beam management, beam activation/deactivation and load balancing, in order to improve the quality of service. The mobile network service area was divided into pixels to enable prediction of fine spatial granularity at intra-cell level.

ANN predictors were found to outperform linear predictors for scenarios where both individual and group users are present which is in line with real world scenarios. Additionally, ANN predictors require shorter observation window lengths than linear predictors to forecast network traffic status accurately. The findings show that ANN predictors do not need to observe the distant past and require less input information, which is very useful considering possible network traffic data unavailability or signaling overheads related to the collection or transfer of the network traffic status data.

REFERENCES

[1] 3GPP RP-181456. Study on RAN-centric Data Collection and Utilization for NR, 2018


Towards a common environment for learning scheduling algorithms

Renato Luiz de Freitas Cunha*†
*IBM Research
São Paulo, Brazil
†Programa de Pós-Graduação em Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil

Luiz Chaimowicz†
†Programa de Pós-Graduação em Ciência da Computação
Universidade Federal de Minas Gerais
Belo Horizonte, Brazil

Abstract—We propose a way to model and integrate HPC scheduling simulators into a popular Reinforcement Learning toolkit. We show experimentally that such an approach not only aids researchers being able to iterate faster by means of software reuse, but also to achieve state-of-the-art performance with 10x less interactions with the environment. We validate the simulation model’s correctness by using unit tests, assertions and experimental comparisons. We also share an open source implementation of the model that will benefit researchers in resource management tasks assisted by Machine Learning.

Index Terms—Reinforcement Learning, Simulation, High-Performance Computing, Standard Workload Format

I. INTRODUCTION

In recent years, we have seen the application of Machine Learning (ML) techniques to diverse areas, such as resource management [4], user-support tools [21], animation [18], image editing, computer-aided art, and games. Particularly for games, Reinforcement Learning (RL) has had compelling, super-human success stories both in board games, such as Backgammon [26], Chess [24], and Go [24], and in digital ones, such as in games from the Atari system [1, 9].

Games play an important role in the development and testing of novel ML algorithms because they have well-defined rules, which tend to be easy to implement, and to simulate in a computer environment. Moreover, since games are developed for human consumption, they are also easy to debug. Especially for digital games, we can attribute the recent success stories to the existence of standardized interfaces to learning environments, which enabled researchers to compare and contrast algorithms and models, alleviating the need to define, test and debug environments, allowing them to focus on research, and to clearly separate between what is code related to an agent and what is code related to environments.

Still, for a long time, the development and testing of agents was done with custom-made software, which made it hard for researchers to reproduce results and to separate algorithm improvements from slight changes in the environment [8]. The introduction of toolkits, such as OpenAI Gym [2] or the Arcade Learning Environment (ALE) [1], for developing and comparing RL algorithms helps solve the aforementioned problems.

Although there are plenty of environments for developing gaming agents and for developing agents focused on solving specific aspects of robotics problems, there is still a lack of environments that could benefit systems research [15], an area that could benefit greatly from RL, since systems problems tend to be sequential in nature and, therefore, a good fit for RL algorithms.

In this paper we present a simulator and corresponding interface with the OpenAI Gym toolkit as an alternative for the development of job scheduling RL agents. We validate our implementation by comparing with results from the literature, and propose extensions to algorithms from the literature to make learning faster and safer.

From a high-level perspective, one might argue why use a toolkit for RL research, when the agent-environment interface (exemplified in Fig. 1 and formally defined in § III) is simple. This is a valid concern, but although the interface itself might be simple, environments themselves are not, making it easy to introduce bugs that hinder learning. Moreover, deep RL has faced reproducibility issues in the past, in part due to the high variance of the methods, and in part due to different metrics being used for reporting results by different researchers [8, 13].

Additionally, by using an existing toolkit, researchers get helper functionality for free. For example, Gym has functionality for representing state and action spaces, functionality for modifying inputs and outputs, recording and monitoring interactions between agent and environment, and libraries of peer-reviewed baseline algorithm implementations that could speed up research. Therefore, with this paper, we hope we’ll help the community by providing a tested, configurable environment for doing RL research in scheduling systems.

In summary, our contributions are:

- An open-source RL environment for learning and evaluating scheduling policies for HPC clusters;
- An experimental analysis of the proposed environment and how algorithms from the literature behave in it;
- Evaluation of state-of-the-art RL algorithms in the environment, providing baseline performance metrics for these algorithms.

The rest of this paper is organized as follows: we present and discuss the related work in § II, while § III presents the background needed for better understanding of the topics discussed in this paper. § IV presents a clear description of the problem we set to solve in this paper. In § V we discuss the methods followed to solve the problem. Experimentation and results are discussed in § VI, and we present our concluding remarks in § VII.
II. RELATED WORK

The application of RL to resource management has been studied in recent years, such as memory controllers, job scheduling, and energy efficiency. For example, Gawłowicz and Zubow [7] integrate the network simulator 3 with Gym for providing environments for optimization of network simulations while Vázquez-Canteli et al. [29] present an environment for optimizing demand response for electricity with RL. Still, few studies use RL frameworks for performing research. Park [15] proposes a set of environments for studying resource management issues, but it does so by proposing a new framework with similar, but not identical APIs to Gym. Jay et al. [10] follow an approach similar to ours, providing environments for RL research integrated with Gym, but focusing on internet congestion control algorithms [31]. Finally, Zhang et al. [32] present an agent that schedules jobs using RL and, although they use a Gym environment, its use seems to be limited to their agents.

The inspiration for this work comes from our previous research on job placement in hybrid clouds [4] and from algorithms such as DeepRM [14]. As presented in this paper, the development of a HPC scheduling simulator integrated with a popular RL toolkit provides an important tool to investigate the application of RL algorithms for job scheduling.

III. BACKGROUND

A. Reinforcement Learning

In a RL problem, an agent interacts with an unknown environment in which it attempts to optimize a reward signal by sequentially observing the environment’s state and taking actions according to its perception. For each action, the agent receives a reward. Thus, in the end, we want to find the sequence of actions that maximizes the total reward, as will be detailed in the next paragraphs. This problem is formalized as a Markov Decision Process (MDP) represented by a 4-tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T})$. At each discrete time step $t$ the agent is in state $s_t \in \mathcal{S}$. From $s_t$, the agent takes an action $a_t \in \mathcal{A}$, receives reward $r_{t+1} \in \mathcal{R}$ and ends up in state $s_{t+1} \in \mathcal{S}$. The transition from $s_t$ to $s_{t+1}$ follows the probability distribution defined by $\mathcal{T}(s_t, a_t)$. In Fig. 1 we show the agent-environment interaction loop. In the figure, the environment represents the kind of problem we consider in this paper: rigid job scheduling in parallel machines.

The agent tries to maximize the expected discounted return

$$R_t = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{T-1} r_T = \sum_{k=0}^{T} \gamma^k r_{t+k+1}, \quad (1)$$

where $\gamma$ is a parameter $0 \leq \gamma \leq 1$, called the discount rate. The discount rate models the present value of future rewards. For example, a reward received $k$ steps in the future is worth only $\gamma^k$ now. This discount factor is added due to the uncertainty in receiving rewards. Since the environment is stochastic, there is no guarantee a future reward will be received.

B. Policies and approximators

RL agents follow a policy $\pi$, where $\pi$ is a mapping from state $s \in \mathcal{S}$ and action $a \in \mathcal{A}$ to the probability $\pi(a|s)$ of taking action $a$ when in state $s$. We are interested in learning $\pi$ directly. In practical problems, however, we are unable to do so, since these problems tend to have very large state spaces. Because of this, we turn to function approximators, which are parameterized functions that allow us to approximate $\pi(a|s)$ by a function $\pi_\theta(a|s)$ with tunable parameters $\theta$. Approximation not only allows us to represent large spaces with a relatively small number of parameters, but it also enables generalization of models, at the cost of using a sub-optimal objective function. Popular function approximators include linear combinations of features [12] and neural networks [24, 26].

C. Policy gradients

When $\pi_\theta(a|s)$ is differentiable, policies can be learned directly based on the gradient of scalar performance measure $J(\theta)$. If $J(\theta)$ increases as an agent’s performance increases, then we want to do gradient ascent in $J$, so that $\theta_{t+1} \leftarrow \theta_t + \alpha \nabla J(\theta)$ gives us increasing performance, with $J(\theta)$ being a stochastic estimate of $J(\theta)$ and $\alpha$ being a scaling factor to the step taken in the direction of the gradient of $\widehat{J}(\theta)$. A corollary of the Policy Gradient theorem establishes that $\nabla J(\theta) = \mathbb{E}_\pi [R_t \nabla \log \pi_\theta(a|s)]$. Combining this with the previous gradient ascent update, we get the REINFORCE update, which is a Monte Carlo update [25]:

$$\theta_{t+1} = \theta_t + \alpha R_t \nabla \log \pi_\theta(a|s), \quad (2)$$

with $R_t$ as defined in equation (1). Being (2) a Monte Carlo update, it might suffer from high variance [25]. One way
The idea of the baseline is key to practical policy gradients. Job 7 appears twice to indicate the difference between the two algorithms. In the queue fits in the system, it is allocated the resources. In this context, time proceeds in a discrete fashion. Therefore, jobs can only run for integral time as well. In a pure FCFS algorithm, job 7 would be scheduled at time unit 9, since it was the last submitted job to the system. In a Backfilling algorithm, job 7 would be scheduled at time 2, since its scheduling would not delay any other jobs. The scenario depicted in this figure was also used to write a test case to assess the correctness of the implementation of the simulator. The bottom part of the figure shows statistics about the jobs, with slowdown computed after completion of all jobs.

![Fig. 2. Expected scheduling behavior of FCFS and Conservative backfilling algorithms for a set of seven jobs submitted to a system with three processors. Job 7 appears twice to indicate the difference between the two algorithms.](image)

<table>
<thead>
<tr>
<th>Job</th>
<th>Submission time</th>
<th>Duration</th>
<th>Processors</th>
<th>Slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FCFS</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>1.3</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>4</td>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>8</td>
</tr>
</tbody>
</table>

To reduce variance is to introduce a state-dependent baseline function in the update, yielding

$$ \theta_{t+1} = \theta_t + \alpha(R_t - b(s))\nabla \log \pi_\theta(a|s).$$  \hspace{1cm} (3)$$

If the baseline does not depend on the action a, then it has zero bias, because it is zero in expectation:

$$\mathbb{E}_a[b(s)\nabla \log \pi_\theta(a|s)] = b(s)\sum_a \pi_\theta(a|s)\nabla \log \pi_\theta(a|s) = b(s)\sum_a \nabla \pi_\theta(a|s) = b(s)\nabla \sum_a \pi_\theta(a|s) = b(s)\nabla 1 = 0.$$  

The idea of the baseline is key to practical policy gradients, algorithms, due to its variance reduction properties. Baselines tend to be learned from interaction with the environment, together with the policy approximation.

D. Job schedulers

Job schedulers are used to manage the job queue and coordinate execution of jobs in supercomputers and High Performance Computing (HPC) clusters. They guarantee jobs execute when requested resources are available and usually guarantee there won’t be over subscription of resources.

A commonly used performance metric for evaluating scheduling systems is the slowdown. Slowdown of job j is defined as slowdown(j) = (t_f(j) – t_s(j))/t_e(j), where t_s(j) is the time job j was submitted, t_e(j) is the time it took to execute job j, and t_f(j) is the finish time of job j.

The basic scheduling algorithm in use is FCFS, also known as First In First Out (FIFO). In this algorithm, jobs are kept in a queue sorted by order of arrival. Whenever the first job in the queue fits in the system, it is allocated the resources it needs. The problem with this approach is that whenever a large job is at the head of the queue, smaller jobs are delayed, since scheduling them would violate the FCFS nature of the queue. Due to that, backfilling algorithms were implemented extensively. Backfilling makes reservations for all skipped jobs. EASY backfilling makes a reservation for that job. This reservation is an upper bound on the time the job would start: if all preceding jobs run to completion, this new job will start running at the reserved time. If earlier jobs fail or finish early, this job may start earlier. After this reservation is made, the scheduler keeps scanning the queue. All jobs that fit the system and that would finish prior to the reservation can be scheduled without delaying the reserved job. EASY backfilling makes a reservation for the first “big” job only, while conservative backfilling makes reservations for all skipped jobs.

For the backfilling procedure to work, algorithms need an estimate of how long a job will execute. Therefore, backfilling systems tend to ask users for run time estimates of their jobs. If users under-estimate job run times, execution of later jobs advance, but if users over-estimate job run times, the system tends to kill such jobs, otherwise reservation times would be violated. Therefore, users tend to estimate their job run times to avoid having their jobs terminated. There is work in the literature that analyses user estimates and their correctness [11]. In this paper, we won’t be making any adjustments and will assume estimates are correct.

IV. PROBLEM DESCRIPTION

In the past, HPC clusters were dominated by bag-of-tasks applications and tightly-coupled message-passing applications. In recent years, we have seen this scenario changing, as Artificial Intelligence (AI) applications gain popularity in HPC centers. Proof of this statement comes from the increasing number of clusters in the top-500 with specialized accelerators (most notably GPUs) for processing AI workloads. Hence, HPC clusters now handle high-throughput, data-intensive, stream-processing applications [20], in addition to traditional applications.

Developing learning algorithms for these new workloads may be slow, difficult and error prone if the community keeps on building custom simulators and custom logic for evaluation. We take a step in the direction of leveraging algorithms and software from the RL community by contributing a simulator that is well integrated with an existing toolkit. For this reason,
in this paper, we are considering rigid jobs in parallel machines, leaving increasing complexity in the simulation for future work. Rigid jobs use a fixed number of processors from start to finish. Therefore, each job is a rectangle in the space of processors per time. Considering these types of jobs allows us to validate our approach with similar approaches from the literature. Also, such models allow us to study execution environments for scientific computing applications, and neural network training procedures that use distributed neural network training frameworks, such as Horovod [23] and DDL [3].

Therefore, the problem we attempt to solve in this paper is that of proposing an approach for using RL for developing novel scheduling algorithms while making use of standard toolkits which enable reuse of proven technologies and approaches from the literature.

V. METHODOLOGY

In this section we describe the methods we used to implement and evaluate our simulator and corresponding RL environment. We begin by describing and validating the simulation core. Then, we describe the development of the RL environment and its validation. Finally, we present novel results based on this environment. Specifically, we present procedures for faster convergence of the learning algorithms, and then describe how state-of-the-art RL algorithms can be applied to the environment, potentially improving performance on the scheduling task described here and other similar tasks.

A. Job Scheduling Simulation

We started by implementing a discrete-event simulator for simulating job submission and execution. We could have started from an existing simulator used in previous research [4], but after careful analysis, we came to the conclusion that the amount of work needed to refactor a simulator to interface with an RL framework would be better spent writing a new one from scratch with extensibility in mind and ensuring its correctness.

The design and development of the core components of the simulator followed a Test-Driven Development (TDD) approach, in which the design of the components followed from the specification of tests of how the components should and should not behave. This approach enabled us to have confidence about the behavior of the simulation system. One example of test of correctness is shown in Fig. 2. In the top part of the figure, we show the expected schedule when seven jobs are submitted for two different scheduling systems. In the bottom part of the figure we show the sequence of jobs that generated that schedule. Therefore, in one test we encoded the sequence of job submissions and their characteristics (number of requested processors and duration for this example) and specified the start, wait, and finish times of each job. Ensuring tests such as this passed gave us confidence the implementation was sound. Since tests alone cannot prove correctness, we also inserted assertions to ensure invariants of algorithms were maintained.

To simulate job execution, one needs to generate load for the simulated system. We added two main methods for simulating job submission. Jobs can be sampled from a distribution, or they can be generated from workload traces. Traces are supported in our simulator by using the Standard Workload Format (SWF). SWF files are text files that contain, on each line, characteristics of jobs, such as submission time, wait time, requested execution time, processors requested, memory requested, actual run time, and so on. The Parallel Workloads Archive [6] contains workload files from supercomputers recorded over a period of almost three decades and, by using those files, we can simulate the loads of real supercomputing centers.

The generative process for job submission is defined as follows, considering $U\{a, b\}$ is the discrete uniform distribution: on each time step, a job can be created with probability $p$. Jobs can be either short or long. A job will be a long job with probability $l$. Actual job run times are sampled from uniform distributions, with long jobs being sampled from $U\{l_1, l_2\}$ and small jobs being sampled from $U\{s_1, s_2\}$. With regards to resource usage, a job can be either processor-dominant or memory-dominant. The dominant resource is chosen with probability $\frac{1}{2}$. The usage of the dominant resource is sampled from $U\{d_1, d_2\}$, and resource usage for the non-dominant resource is sampled from $U\{n_1, n_2\}$, where $r$ is the quantity of resources of the cluster. In this paper, $r$ was set to be the same for both processors and memory.

B. Gym RL environment

With a working simulator, we now had to define an environment that would work with the OpenAI Gym. We decided not to try to devise a new environment of our own, but rather to use a successful, peer-reviewed one. For this reason, we drew inspiration for our environment from DeepRM [14]. DeepRM is an algorithm and corresponding MDP definition whose purpose is to learn to schedule rigid, parallel jobs. The following discussion documents how we implemented the DeepRM MDP and integrated it with OpenAI Gym. A more detailed description of the MDP is presented by Mao et al. [14].

Currently, Gym only supports environments interfaced with it in the Python language, so the environment was naturally implemented in Python as well. To interface with Gym, environments are required to inherit from the gym.Env class and implement the following five methods: __init__, for initialization; step, for implementing the MDP itself via the $T(s, a)$ function, the set of states $S$, the set of actions $A$ and the reward signal from $R$; reset, for resetting the environment state; render, for rendering the environment (both for human visualization or consumption by other programs); and close for closing the environment and freeing any resources.

Once an environment is registered with Gym (with a call to gym.envs.registration.register), it can be instantiated with a call to the gym.make function with the environment name passed as string. This will instantiate a new environment and call the __init__ method of our environment. This function takes an optional argument that is passed to the initializer. We use this function to be able to configure the environment. One such configuration option is how the
workload is going to be generated, which uses the job generation facilities described in the previous section. Jobs that enter the system will either be displayed in the waiting queue, or, when it is full, will be added to a backlog. Jobs in the backlog have their characteristics concealed. All the agent knows is that such position is occupied in the backlog. Fig. 3 shows a subset of the state representation of the system (as generated by a call to the render method) for three timesteps (with an intermediate timestep not shown). As shown in the figure, a job is scheduled between \( s_t \) and \( s_{t+1} \). From the first timestep to the next, the agent had decided to schedule the first waiting job, changing the state of the cluster processors, of the waiting queue, and of the backlog. In the simulation, time passes only when the agent is done scheduling jobs (or tries to schedule a job that won’t fit the cluster.) Note that the state representation doesn’t look ahead far into the future. In fact, the agent can only see a limited number of steps into the future. In the figure, time horizon was set to 5 at environment initialization time. The logic of job arrivals and changes to the state representation of the cluster and the queues, describes the transition function \( \mathcal{T}(s_t, a_t) \) of our MDP.

The set of actions \( \mathcal{A} \) the agent can take corresponds to positions in the waiting queue, plus an additional action that indicates it doesn’t want to select any jobs. The size of the waiting queue is fixed at environment initialization time. Hence, if the waiting queue has five positions, the agent will have six actions available. Although limited in the environment, the simulator that backs it will maintain an unbounded queue. Jobs that do not fit in the queue shown to the agent are added to the “backlog”. After each action is taken, a reward signal is returned by the MDP. Ideally, we would like to use the raw slowdown of jobs as reward signal, but due to the dependence on \( \ell_j(j) \), we can only know the actual slowdown of job \( j \) once it finishes executing. Therefore, using it as reward signal would have introduced credit assignment issues to the agent’s actions. Luckily, there is an incremental formulation of the slowdown: return \( 1/\ell(j) \) for each time step \( j \) is in the system until it finishes execution. The key insight is that the sum of all time steps a job is not scheduled equals the wait time \( \ell(j) \), therefore, slowdown\( (j) = \frac{1}{\ell(j)} \left( \sum_{i=1}^{t_u(j)} 1 + \sum_{i=1}^{t_a(j)} 1 \right) \). Since the agent maximizes its reward, we set the reward signal to \(- \sum_j 1/\ell(j) \) when there are unfinished jobs in the system, and zero otherwise.

Whenever the agent schedules a job successfully, it receives zero reward, which is the highest reward it can get. To complete the description of the MDP, the set of states \( \mathcal{S} \) comprises all possible jobs that may arrive in the system and all their possible scheduling assignments. This set needs not be precomputed: elements are generated by applying the \( \mathcal{T}(s_t, a_t) \) function.

The code used to implement this environment is publicly available at https://github.com/renato1fc/sched-rl-gym.

C. Policy network and learning procedure

In this work we used a Multi-Layer Perceptron (MLP), a fully-connected feed-forward neural network, with a single hidden layer of 20 units for function approximation. The neural network used a Rectified Linear Unit (ReLU) as activation function and works similarly to a classifier: given an input, the neural network generates probabilities for each class. The number of classes is equal to the number of positions in the waiting queue in the state representation plus one. In the minimalistic example of Fig. 3, there are three classes, corresponding to the first waiting job, the second, or no job.

Algorithm 1: REINFORCE with average baselines

<table>
<thead>
<tr>
<th>Result: Learned weights ( \theta ) for policy ( \pi_{\theta} \approx \pi_{\star} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input : Differentiable policy ( \pi_{\theta}(a</td>
</tr>
<tr>
<td>Input : Parameters ( \alpha &gt; 0, E, T, W )</td>
</tr>
<tr>
<td>for each worker ( w ) ← 0 to ( W - 1 ) do</td>
</tr>
<tr>
<td>[ \Delta \theta_w \leftarrow 0 ]</td>
</tr>
<tr>
<td>for ( i ) ← 0 to ( E - 1 ) // Sampling phase do</td>
</tr>
<tr>
<td>[ e_i \leftarrow { s_0^w, a_0^w, r_1^w, \ldots, s^w_{T-1}, a^w_{T-1}, r^w_T } \sim \pi_{\theta}(\cdot</td>
</tr>
<tr>
<td>foreach ( t</td>
</tr>
<tr>
<td>[ R_i^w \leftarrow \sum_{k=0}^{T} \gamma^{k-t-1} r_i^w ]</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>for ( t ) ← 0 to ( T - 1 ) // Learning phase do</td>
</tr>
<tr>
<td>[ b_t \leftarrow 1/</td>
</tr>
<tr>
<td>for ( i ) ← 0 to ( E - 1 ) do</td>
</tr>
<tr>
<td>[ \Delta \theta_w \leftarrow \Delta \theta_w + \alpha \nabla \log \pi_{\theta}(a_i</td>
</tr>
<tr>
<td>end</td>
</tr>
<tr>
<td>foreach ( \Delta \theta_w ) do ( \theta \leftarrow \theta + \Delta \theta_w )</td>
</tr>
</tbody>
</table>

We used the REINFORCE learning procedure to learn the weights \( \theta \) of our neural network. REINFORCE is a Monte Carlo algorithm that works in two phases: a sampling phase and a learning phase that updates the weights. In the sampling phase, the agents use fixed weights to interact with the environment, obtaining trajectories, sequences \( \{s_0, a_0, r_1, s_1, a_1, \ldots, s_T, a_T, r_T\} \) of states, actions and rewards, where \( T \) is the maximum length of an episode, configured at environment creation time (§ V-B). Algorithm 1 shows one iteration of the learning procedure followed in this paper. The algorithm takes as input a step size \( \alpha > 0 \), a number of workers \( W \) to be executed in parallel, a number of episodes \( E \) to be sampled by each worker, and a maximum length of episodes \( T \).

Since adjustment of weights only occurs in a specific phase, it is possible to perform data collection in parallel. We’ve implemented two distinct strategies: (i) each worker computed weight updates locally and propagated them to all other workers (in HOGWild! [19] style) and (ii) a worker received trajectories from all other workers, performed learning locally and broadcast new weights to all other workers. As shown in the algorithm, baselines are computed as the average reward at each timestep. In variant (i) of the algorithm, baselines are computed locally and are not shared, while in variant (ii), baselines are shared between all episodes, since they are computed by a central worker. Algorithm 1 computes baselines locally, implementing variant (i) above. As we shall see, there seems to be no difference in algorithm performance between both methods of computing baselines.
D. Applying state-of-the-art algorithms

In DeepRM [14] and in Algorithm 1, baselines are the total returns received from time step $t$ averaged over all trajectories. This means that, since simulation and environment clocks aren’t synchronized, trajectories will have slightly different lengths based on the different decisions (and jobs) submitted to each worker. Therefore, trajectories might need to be padded, or truncated when implemented in code. A better approach would be to learn the baseline, as performed by actor-critic algorithms such as Advantage Actor-Critic (A2C) [16] and Proximal Policy Optimization (PPO) [22].

We’ve used publicly available, reviewed implementations [5] of the A2C and PPO algorithms to optimize our MDP. These algorithms learn the baseline function with a distinct set of parameters, $v_{\phi}(s)$, used to learn the policy. They also use the entropy of the policy as regularization to prevent early convergence to a local minimum. The full update step for A2C is

$$
\Delta \theta \leftarrow \Delta \theta + \nabla \log \pi(a_t|s_t)(R_t - v_{\phi}(s_t) + \beta \nabla H(\pi_{\theta}(s_t))),
$$

where $\beta$ is a hyperparameter and $H(X) = -\sum_{x=1}^{n} P(x) \log P(x)$ is the entropy function.

We are interested in learning whether such algorithms are able to converge faster and whether they’d be able to outperform an algorithm custom-built for this environment and, if so, by how much.

VI. EVALUATION

Our main objective is to assess whether our environment has been implemented correctly and to understand whether existing RL algorithms can learn in such an environment. A secondary objective is to show that we can apply state-of-the-art algorithms to this environment, resulting in faster convergence and, possibly, better performance.

A. Scheduling simulation validation

To assess the correctness of the simulator, we performed verification and consistency tests. We performed automated unit tests every time a commit was pushed into the repository. When measuring the coverage of tests (the percentage of lines of code hit by testing routines), we obtained 79% total coverage for the complete codebase. After removing code related to the RL environment from analysis, we get 95% test coverage. Thorough testing allowed us to catch difficult bugs early on, giving us confidence the code implemented the “contracts” defined in the unit tests.

We also compared the output of our simulator with the work of Tsafrir and Feitelson [28]. Basically, we simulated execution of the San Diego Supercomputer Center (SDSC) SP2 log from the Parallel Workloads Archive [6] using an EASY scheduler and compared the average bounded slowdown with theirs. We obtained similar results, with slowdown curve following the same pattern and in the same order of magnitude. Also, when comparing wait times between the simulated and the real system, we obtained $\approx 0.6\%$ difference in mean wait time, and median difference of wait times of $6s$.

B. Validation of the environment

Next, we evaluate whether an algorithm from the literature is able to learn in our proposed environment. To do so, we instantiated an environment with the following configuration: episodes lasted for 50 simulation time steps; time horizon was set to 20; backlog had 60 entries; the cluster was configured to have 2 different resource types (processors and units of memory) with 10 units of each resource; new job probability was set to 0.3; long job probability $l$ was set to 0.2; range of duration for long jobs was set to $l_1 = 10, l_2 = 15$, duration for short jobs was set to $s_1 = 1, s_2 = 3$; dominant resource usage

![Fig. 3. Snapshot of three frames of the DeepRM environment implemented in this work when rendered via the Gym rendering function, along with annotations for long jobs was set tol1 = 10, l2 = 15, duration for short jobs was set to s1 = 1, s2 = 3; dominant resource usage](image-url)
was set to $d_1 = 5, d_2 = 10$, non-dominant resource usage was set to $n_1 = 1, n_2 = 2$.

To implement the policy network, we used PyTorch [17]. PyTorch allows one to define neural networks by inheriting from `torch.nn.Module` and implementing the `forward` method of those classes. The forward method corresponds to the forward pass of the neural network and once it is performed, the automatic differentiation engine of PyTorch keeps track of operations to allow gradients to flow in the backward pass. We implemented the update of Algorithm 1.

Optimization was performed using the RMSProp [27] gradient-descent optimizer with learning-rate $\alpha = 0.001$, discount factor $ \gamma = 0.99 $, with 10 workers, each collecting 200 trajectories per epoch prior to learning. We’ve implemented both baselining strategies discussed in Section V-C by using a fork-join model: the first process kept the main copy of the parameters in memory and spawned child processes which would sample trajectories. Once sampling was done, updates to the parameters were performed in a shared-memory copy of the parameters. PyTorch facilitates this style of programming by providing a multiprocessing module, responsible for spawning workers, and automatic shared memory of parameters. In the centralized version, we centralized trajectories in the main process before computing gradients. In the decentralized version, each worker computed their own baselines and only propagated gradients to the main process. Each iteration of this version is faster than the centralized one because there is no need synchronize all workers to apply updates.

Fig. 4 shows the learning curves of the various agents. To reduce noise, the curves were smoothed with a moving average with window size 10. In the figure, DeepRM is the version of the agent with centralized baseline calculation, while DeepRM-worker uses HOGWILD!-style learning. We can see that both DeepRM implementations start outperforming SJF at around iteration 200. We see that both learning curves shown reproduce those of Mao et al. [14]. We also evaluated slowdown learning curves, which exhibited similar behavior to those of previous research. This complete integration test gives us confidence that the environment is sound, and enabled us to replicate results from the literature. The fact that we were able to reproduce results from the literature should be emphasized. Reproducing existing work and judging their improvements is vital to sustaining progress not only in RL as an area, but in areas that can make use of it. As Henderson et al. [8] have shown, the dynamics of environments help determine what algorithms are more successful. Therefore, if we are to apply RL to resource management and be able to translate research progress to practical situations, we need stable, sound, and representative environments.

C. Faster convergence and state-of-the-art performance

Finally, we’re now able to evaluate other algorithms in the same environment. Of note is the ease of use of the environment with packages that target the Gym toolkit. While our implementation of DeepRM plus support code for training and persistence took 212 lines of code disregarding comments and empty lines, our implementation of A2C and PPO, with support code, took 44 lines. In these experiments we used the default hyper-parameter values and did not perform any hyper-parameter search. The policy and value networks had identical topology to that of DeepRM’s.

To keep agents comparable, we configured the agents to execute the same number of steps for learning as a DeepRM worker would have used. Recall that, in our experiments, we had 10 agents collecting 200 trajectories for 50 timesteps. Here, we had a single agent interacting with the environment for $200 \times 50 = 10,000$ time steps. Therefore, the A2C and PPO agents performed roughly $10x$ less interactions with the environment than DeepRM.

Fig. 5 shows the learning curves of A2C and PPO. As can be seen from the figure (5a), average slowdown performance from PPO and A2C is more stable than DeepRM, learning rapidly to around 2.5 and decreasing slowly from there. Upon inspection of the behavior of the agents in our environment, we noticed the agents learned quickly to output a fixed action corresponding to the first positions of the waiting queue. This made slowdown performance better on average than DeepRM’s, which exhibited closer to random behavior in early iterations. Even though average performance was better in early iterations, those iterations had higher variance than later ones, which exhibited more purposeful behavior.

Both A2C and PPO have a regularization term that improves exploration by discouraging premature convergence to suboptimal deterministic policies. This regularization term is a function of the entropy of policy $\pi_0$ [30] and we noticed that when it was zero, the algorithms did get stuck in local minima of policies. In both the A2C and PPO implementations, we used a value of 0.01 for this parameter.

Fig. 5b shows that total reward follows the same trend of Fig. 4b and indicates learning is improving. Deeper inspection shows that the maximum total reward of DeepRM is $\approx 20\%$ better than that of A2C, but mean total reward is only $\approx 10\%$ better, indicating A2C converges faster than DeepRM. With regard to slowdown (Figs. 4a, 5a), A2C’s minimum slowdown was $\approx 9\%$ better than DeepRM’s, with A2C’s mean slowdown
over all training iterations $\approx 19\%$ better than DeepRM’s.

In short, from these experiments we can see that, although not outperformed easily, algorithms from the literature that have no knowledge of the inner workings of the environment were able to achieve performance comparable to that of DeepRM’s. An environment such as the one proposed here will make algorithm discovery faster, by providing a standard interface to batch scheduling simulators.

VII. DISCUSSION AND CONCLUSION

In this paper we presented the development of an RL environment with support for a framework that enables faster, more effective research. We’ve discussed how correctness of the environment was assessed and how it was able to reproduce results from the literature, increasing confidence in its implementation and usefulness. We concluded by showing that we’re able to obtain state-of-the-art results by using baseline implementations of advanced RL algorithms that would’ve been error-prone and expensive to implement had we decided to use a custom environment. Our results shows convergence to results comparable to custom-made algorithms using 10 times less interactions with the environment. We open-sourced the environment and made it available on the Internet in the hopes it will be useful for other researchers.

Although not discussed in this paper, the environment is able to simulate real traces, as briefly discussed in § V-A. It is our intention to extend the environment to support job dependencies, preemption, and an unlimited set of actions for the agent. Having shown the environment is a strong foundation, we leave these improvements to future work.

REFERENCES

Fabsim-X: A Simulation Framework for the Analysis of Large-Scale Topologies and Congestion Control Protocols in Data Center Networks


Email: *malek.musleh, †roberto.penaranda, ‡allister.alemania, §pedro.yebeles.segura, ¶gene.wu, ‖jan.zielinski, **krzysztof.raszkowski, ††nan.ni, ‡‡scott.diesing, ‡‡‡anupama.kurpad, ‡‡‡ram.huggahalli@microsoft.com§, ***curt.e.bruns, ****steven.c.miller, *****sujoy.sen

Intel Corporation (@intel.com)

Abstract—The explosive growth in cloud-computing and overall data center system growth has created an unprecedented demand on system architects and designers to continuously develop more complex system networks to effectively satisfy the insatiable appetite to process, move, and store large amounts of data. Non-linear system behavior caused by emerging workloads and use-cases, varying end-to-end congestion protocols, and heterogeneity in the various compute and storage capabilities of custom designed accelerators further compounds the design problem. Modern simulation methodologies lack a cohesive and efficient framework to address the interoperability of the intersecting layers at scale. In this paper, we present a simulation framework for evaluating congestion control protocols. Furthermore, we present a set of optimizations that enable analysis for longer simulated times and at network scales up to 128K nodes, which is vital for proper analysis of workloads that require long run times (e.g., AI training) or workloads that are known to have scaling issues (e.g., RDMA). Specifically, we evaluate congestion control performance at various scales, study the implications of topology scaling on congestion, and the performance impact of simultaneous heterogeneous protocols.

Index Terms—Performance Simulation, Networking, Congestion Control, Fat-Trees, TCP, iWARP, RoCEv2

I. INTRODUCTION

In today’s technology-driven world, the demands for cloud compute and data movement are ever-increasing. A prime example of this is the move to cloud gaming with solutions such as Google’s Stadia [1], Nvidia’s GeForce Now [2], and Microsoft’s Project xCloud [3] replacing traditional set-top box gaming system hardware with a completely cloud-based service solution. As a consequence of this shift to cloud, modern data center networks (DCNs) are experiencing explosive growth in terms of both scale to handle increased concurrency of users and workloads as well as heterogeneity as more specialized accelerators and other hardware are deployed to handle workloads that are only becoming more complex. This increased scale and heterogeneity combine to create an increasingly complex environment that makes it very difficult for DCN operators and system architects to understand performance implications ahead of time. It also reduces the ability to determine causal relationships thereby diminishing visibility of the underlying causes of emergent behavior. Traditionally, performance prediction is done using test clusters or extrapolation based on existing performance. However, this approach breaks down with today’s scales due to the extreme costs to deploy test clusters of meaningful size and the non-linear behavior of performance as DCN scale and heterogeneity increases. Consequently, large scale network simulators have become an attractive tool to predict performance at large scales and heterogeneity.

There are several simulators that can potentially be used for DCN performance research. Table IV compares a number of such simulators and highlights some of their key features or lack thereof with respect to large scale DCN simulation and performance prediction. In this paper, we introduce Fabsim-X, an event-driven simulation framework capable of several applications such as evaluating job placement, capacity planning, new architectural proposals or congestion control (CC) protocols at data center scales (we present data up to 128K network endpoints). We describe the optimizations implemented to simulate these scales, and demonstrate the capabilities of the simulator by presenting an in depth performance analysis of various CC protocols at a number of scales. We also analyze the implications of topology size and concurrent heterogeneous CC protocols on network performance.

This paper is organized as follows. In Section II, we provide an overview of Fabsim-X and its structure. Section III describes the enhancements applied to Fabsim-X to achieve very large scale DCN simulations. To demonstrate Fabsim-X capabilities in a controlled manner, we characterize Transmission Control Protocol (TCP) protocols at small scale in Section IV-A. In Section IV-B, we show the network performance and run-time evaluation of TCP protocols in large systems up to 128K endpoints. Section V discusses alternative simulation tools and explains why Fabsim-X is a more suitable simulation tool and discusses future directions and challenges in large scale DCN simulations.
Fig. 1: Fabsim-X software architecture framework for analyzing large scale DCNs. Finally, Section VI summarizes our findings and describes potential future work.

II. FABSIM-X OVERVIEW

Fabsim-X is an event-driven network simulator written in C++. It employs the popular JavaScript Object Notation (JSON) plain-text format to specify configuration and Python for data analysis and plotting. Fabsim-X features a modular software architecture (see Fig. 1) that consists of several abstraction levels that are described in the following paragraphs.

The first two layers represent the simulator kernel. The event-driven simulation engine layer provides and manages a distributed event queue (one per MPI rank) for event scheduling and execution. The distributed simulation layer facilitates large-scale simulation by enabling a simulation to be distributed amongst many concurrently executing cores that synchronize using the Message Passing Interface (MPI). Fabsim-X supports the ability to split a simulation across multiple MPI ranks at a simulated switch or endpoint granularity. For example, a common case is to have each MPI rank simulate a single switch and all of its attached endpoints.

The next set of layers model the network. The switch models layer includes two light-weight switch models, a Virtual Output-Queued (VOQ) switch [4] and a Combined Input-Output Queued (CIOQ) switch [5], as depicted in Fig. 2. These light-weight models accurately capture congestion and routing behavior while abstracting away the internal micro-architectural facets of an Ethernet switch. In contrast with the VOQ switch, the CIOQ switch model can accurately capture the Head-of-Line (HoL) blocking effect produced within a switch, but at the run time cost of executing more events [6]. With a focus on demonstrating first-order congestion phenomena, we therefore utilize the VOQ model to best enable simulation scaling to very large topologies. The node models layer represents the Network Interface Controller (NIC), which manages communication between the endpoint and the network. The routing & topology layer manages the interconnection of each component (e.g., endpoint and switches) of the network. This extendable layer supports many connectivity schemes (topologies) including but not limited to tree [7]), Dragonfly [8], and Megafly [9]. This layer also supports multiple configurable routing algorithms, which define the path a packet takes from source to destination endpoint. Various routing algorithms are supported such as Random Packet Spray (RPS) [10], Equal-Cost MultiPath (ECMP) [11], and LetFlow [12]. The end-to-end congestion control layer implements CC protocols including but not limited to Data Center Quantized Congestion Notification (DCQCN) [13] as well as several TCP protocols such as NewReno [14], Data Center TCP (DCTCP) [15], and Cubic [16].

The next set of layers represent the application layer, which defines the communication pattern between endpoints. The Scalable Workload Models (SWM) layer defines the communication characteristics of a simulated application, i.e., how the endpoints communicate with each other [17]. The layer provides a flexible programming model enabling users to define simple synthetic patterns or more complex patterns based on real applications, that is compiled as a separate dynamically loadable library to facilitate ease of sharing workloads. In proof of this, SWM compatibility has been recently extended to the CODES simulation framework [18]. The SWM runtime engine layer maintains the state of all outstanding messages, which models dependency relationships in communication such as blocking and non-blocking collectives.

The configuration files layer consumes user-specified simulation configuration and creates internal data structures that are used by the rest of the Fabsim-X simulation framework. The simulation configuration uses the JSON format and it is broken down into logical sections for switch-specific, endpoint, CC protocol, and topology configuration. This approach enables reconfiguration without recompiling.

The HDF5 static library [19] layer provides a centralized means to manage and aggregate statistics. These statistics are outputted to a compressed HDF5 file for post-processing and charting. While this layer only handles aggregate statistics such as total packets sent by each endpoint, Fabsim-X also provides the capability to output temporal statistics such as switch buffer usage, which can be incredibly useful when analyzing the transient network behavior. For example, while an aggregate statistic such as average throughput provides a measure of how an endpoint performs overall, it does not provide any insight on how that endpoint performs in the presence of bursty traffic. Fabsim-X can be used to quantify
Fig. 3: Topology connectivity of a 2048-endpoint fat-tree

temporal congestion issues caused by bursty traffic via detailed visualization. Finally, to increase ease of use, the visualization tools layer provides a wealth of auto-generated plots in an easy to consume format. Customized data mining and charting is also possible by using the statistics exposed in the HDF5 file.

III. SIMULATION METHODOLOGY AND ENHANCEMENTS

In this section, we describe the optimizations implemented to achieve reasonable run times of simulated large scale DCNs. This can be accomplished by increasing the abstraction level of certain network elements or mechanisms to achieve fast speeds at an acceptable loss of precision. Specifically, three key optimizations are performed: 1) Packet-Based Simulation Mode, 2) Efficient Distributed Simulation, and 3) Relaxed MPI Synchronization.

A. Baseline Methodology

Table I describes the configuration of the physical cluster used to simulate our experiments. All experiments simulate a fat-tree topology with the format depicted in Fig. 3. The figure depicts a two-level, full bisection-bandwidth fat-tree with 2048 endpoints where endpoints and switches are represented by circles and rectangles, respectively. Endpoints are connected to their switches using a single 200 Gbps link. The switches are interconnected in an all-to-all fashion between the Tier-0 switches (bottom row of rectangles) and Tier-1 switches (top row of rectangles) with each line composed of four individual 200 Gbps links resulting in a total of 800 Gbps of aggregate bandwidth between switches. Each switch features 128 ports, each operating at 200 Gbps – 25 Tbps of aggregate bandwidth.

B. Packet-Based Simulation Mode

Fabsim-X models multiple levels of communication granularity as illustrated in Fig. 4. Application-level communication is performed using messages where the set of messages communicated between endpoints is referred to as a flow per DCN terminology. These messages are broken into smaller units called packets, which are processed by CC protocols. For the purposes of simulating conventional Ethernet DCNs, the experiments in this paper are configured to use packets with a maximum transmission unit (MTU) of 1.5KB. Fabsim-X also supports breaking down packets into flow control units (flits), which are a network construct that enables flow control. CC protocols that operate on best-effort type of networks, do not require modeling flit-level behavior. For the purposes of CC protocol performance evaluation, we exploit this fact to reduce the number of required events via configuring flits to be equivalent to packets (i.e., make the packet the atomic data unit), which significantly increases simulation speed.

Fig. 5 shows the event count and run time benefit of using packet-based communication instead of flit-based communication for a 2048-endpoint topology running a TCP NewReno workload for 10 ms. The workload is a static permutation in which each endpoint continuously sends messages to exactly one other randomly selected endpoint, and no endpoint can receive messages from more than one endpoint. In this experiment, each endpoint is configured to send as fast as possible. However, due to NewReno’s congestion windowing mechanism, achieved injection rate is less than line rate.

The experiment is repeated multiple times varying numbers of MPI ranks to demonstrate the run time benefit of distributed simulation. We assign no more than 1 MPI rank per machine to avoid being compute bound, so the number of MPI ranks used corresponds to the number of Broadwell machines (see Table I) used to execute the simulation. Along the X-axis, the figure illustrates the corresponding bars for packet-based communication using the left Y-axis that illustrates the actual wall clock time in minutes. Fig. 5 also shows the percentage of run time speedup when moving to packet-based communication for different MPI ranks. The results reveal that we achieve a speedup between 9.49x and 19.49x.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor Model</td>
<td>Intel(R) Xeon(R) CPU E5-2667 v4 @ 3.20GHz (Broadwell)</td>
</tr>
<tr>
<td>Num Cores</td>
<td>8 core dual-socket (16 total)</td>
</tr>
<tr>
<td>L1Cache</td>
<td>32 KB (I/D split)</td>
</tr>
<tr>
<td>L2Cache</td>
<td>256 KB</td>
</tr>
<tr>
<td>L3Cache</td>
<td>25600 KB</td>
</tr>
<tr>
<td>Main Memory</td>
<td>256 GB Ram</td>
</tr>
<tr>
<td>Network Connection</td>
<td>10 Gbps Ethernet</td>
</tr>
<tr>
<td>Rack size</td>
<td>48 Machines</td>
</tr>
</tbody>
</table>

Table I: Physical cluster configuration
both the default and custom methods. The optimized policy realizes a ~13-17% improvement over the default policy, which comes as a direct result of a ~6% reduction in the number of link cuts (from 1020 to 960) and improved load-balancing. Note that the effectiveness of this policy depends on the workload and routing strategy simulated because they define the utilization of each link. Our goal is to demonstrate the capability and runtime benefit of reducing communication overhead, rather than proposing new partitioning schemes. Techniques from prior art can be applied [20], [21].

D. Relaxed MPI Synchronization

Similar to other networking simulators or frameworks such as SST [22] and CODES [18], Fabsim-X utilizes MPI to spread a simulation across many distributed ranks. Doing so increases the compute power to execute the simulation at the cost of synchronization between MPI ranks in order to maintain deterministic behavior. Existing research in this area observe similar opportunity to reduce synchronization overhead, but employ complex event queue rollback techniques coupled with run-ahead execution [23] that severely limit the scaling efficiency across topology and workload assumptions.

By default, Fabsim-X schedules a synchronization event every link cycle. This conservative synchronization policy can be inefficient if the communicating MPI ranks do not have any meaningful data/payload to transfer. To address this inefficiency, we implement an optimization called relaxed MPI synchronization. As illustrated in Figure 6, the simulation is partitioned across links. Therefore, MPI ranks need only synchronize when simulated payloads traverse between switches or endpoints on different MPI ranks, and need not synchronize simply to exchange simulated timestamps to ensure correctness and causality. The experiments we describe in this paper use static simulated link latencies of 100 ns. We take advantage of this to reduce the number of MPI synchronization events (to be once every simulated 100 ns) required to execute the simulation as illustrated in Fig. 7. Note that increasing the synchronization period reduces the total number of MPI events but may increase the average time per MPI event as there might now be more packets to process per event.

Fig. 8 demonstrates the run time performance benefit of the relaxed synchronization optimization. We simulate the topology in Fig. 3 with NewReno for a simulated duration of 10 ms. Relaxing the communication synchronization frequency is expected to reduce its overhead, but it may not achieve an overall run time reduction if the simulation is compute-bound. To better understand these implications, we repeat the experiment using different synchronization periods for packet-
In this section, we characterize the fundamental behavior of various CC protocols. We examine the congestion window (CW) and instantaneous injection bandwidth (IIB) temporal charts for a 2-to-1 in-cast scenario. Both of these temporal charts provide a level of insight on fairness and reaction of a CC protocol that cannot be achieved using an aggregate statistic such as average achieved bandwidth. They also provide a visual reference used to validate the expected behavior of the protocols according to their respective Request For Comments (RFC) references as well as academic and industry literature.

The figures in this section are interpreted as follows. In Fig. 9, the X-axis represents simulated run time. For the top row of charts, the Y-axis represents the achieved bandwidth relative to link rate (e.g., for an endpoint link of 100 Gbps, an achieved bandwidth of 0.8 represents 80 Gb/s). The Y-axis on the bottom row charts represents the congestion window of each flow. For example, a congestion window value of 20 indicates a flow can have up to 20 outstanding packets. Note that congestion window and instantaneous injection bandwidth are directly correlated—the greater the congestion window, the more packets a flow may have in flight, and therefore, the higher the instantaneous injection. We first analyze the simplest of the 2-to-1 in-cast scenarios in which both senders employ the NewReno CC protocol. Figs. 9a and 9d illustrate the IIB and CW evolution, respectively.

We observe two key findings from the figures: 1) there is significant bandwidth unfairness between the flows, and 2) the flows under-utilize total available bandwidth. Neither flow is able to achieve equilibrium as exhibited by the significant variability in IIB (see Fig. 9a). This behavior is caused by the dramatic changes in CW as illustrated in Fig. 9d and it matches the behavior described in the protocol’s RFC [14]. At the beginning, the initial surge of packets results in many dropped packets as the two flows converge on the single destination. These dropped packets subsequently result in timeouts in each flow causing the CW to reset to one (i.e., the injection rate drops significantly) and the flow to move into the slow start state. With every packet acknowledgement (ACK), the flow increases its CW (i.e., injection rate increases) and the flow eventually enters the congestion avoidance state. This process continues until a congestion event occurs, at which point the flow enters the fast recovery state (where the CW is halved instead of being reset to one to facilitate quicker rate recovery) in the case of the reception of multiple duplicated ACKs, or the flow enters the slow start state in the case of a timeout.

Unfortunately, the lack of coordination between the flows results in the key observations noted earlier: 1) there is significant bandwidth unfairness between the flows, and 2) the flows under-utilize total available bandwidth. We next investigate performance when flows using different CC protocols compete against each other, which is very common in modern DCNs. In this specific experiment, we examine the performance of a 2-to-1 in-cast where one sender employs the NewReno CC protocol while the other sender employs User Datagram Protocol (UDP), which does not feature any congestion avoidance. Fig. 9b and 9e show the IIB and CW, respectively. We see much more consistent behavior than we did for the NewReno/NewReno experiment described earlier. This occurs because the UDP flow does not react to congestion, it simply injects as quickly as possible. This presents very consistent feedback to the NewReno flow allowing it to quickly (~100 us) reach equilibrium. We expect the ideal equilibrium point to be 50% because the destination switch arbitrates between the competing flows in round-robin fashion (i.e., all competing flows receive an equal share of the bandwidth). The NewReno flow sees that about half of its packets dropping, so it lowers its injection to approximately 40%. The lost 10 basis points of injection are lost as a result of NewReno overhead, which is always present and can be seen in isolation after the UDP.
flow completes at around 375 us. Note that while the UDP flow injects at 100%, approximately 50% of its packets are dropped in the destination switch due to the arbitration. In the final set of experiments, we analyze the behavior of a newer TCP implementation, DCTCP, competing with a UDP flow. The primary difference between DCTCP and NewReno is that, while both can respond (i.e., throttle) to packet drops, DCTCP attempts to reach equilibrium quicker by anticipating congestion-related packet drops using Explicit Congestion Notifications (ECNs). A switch marks a packet with ECN when its buffers reach a configured threshold. This ECN is then fed back to the sender to proactively throttle back its injection rate in order to avoid overflowing the switch buffers (i.e., packet drops). The proactive throttling of DCTCP can be very effective when all other competing flows adhere to similar rules. However, when a competing flow, such as the UDP flow in this experiment, does not also perform proactive throttling, we see that DCTCP quickly becomes victimized. Fig. 9c shows that the UDP flow takes advantage of the DCTCP flow’s proactive throttling and quickly consumes all the bandwidth to the destination endpoint. This continues until the UDP flow completes, at which point the DCTCP flow no longer sees congestion and increases injection up to 100%. This experiment underscores the need to be cognizant of traffic interference and highlights that even today’s most advanced CC protocols are not immune to adversarial flows.

B. Simulation Scaling of CC Protocols to Large-Scale DCNs

In Section IV-A, we validate our CC protocol models using controlled experiments. In this section, we demonstrate run time and performance scaling on larger topologies of varying sizes—from 2K to 128K endpoints.

Fig. 10 illustrates the run time and performance scaling as measured by packets transmitted of Fabsim-X. We start with the baseline from Fig. 3, and then we increase the number of endpoints up to 64K. The 2048- and 4096-endpoint topologies are both 2-tier fat-trees while the remaining topology sizes are 3-tier fat-trees to accommodate the greater number of endpoints. All topologies maintain the same characteristics as much as possible simulated on 48 MPI ranks spread across 3 Broadwell machines (see Table I). In Fig. 10, the X-axis represents the topology size, the left Y-axis corresponding to the bars represents the number of simulated packets transferred in the network, and the right Y-axis represents the run time depicted by the plotted line. The results demonstrate Fabsim-X’s ability to deliver fairly linear run time and performance scaling as measured by the number of packets transmitted. The overall simulated work performed (i.e., packets transmitted) scales linearly because the different topology sizes maintain similar subscription ratio between tiers. Evaluating the run time effects of different connectivities is left to future work.

Table III lists the run time to simulate a 128K-endpoint,
Table III: Runtimes of a 128K-endpoint topology simulated for 50 ms for different sync. periods and CC with 360 Ranks

<table>
<thead>
<tr>
<th>MPI Synchronization Period</th>
<th>NewReno</th>
<th>DCTCP</th>
</tr>
</thead>
<tbody>
<tr>
<td>50ns</td>
<td>136h:37m</td>
<td>146h:33m</td>
</tr>
<tr>
<td>100ns</td>
<td>105h:59m</td>
<td>123h:26m</td>
</tr>
</tbody>
</table>

Fig. 11: Received and dropped packets for a 128K-endpoint topology simulated for 50 ms - grouped in 10 ms windows

3-tier fat-tree topology for 50 ms using different MPI synchronization periods and CC Protocols. The workload is a static permutation, described in Section III-B. Note that the attempted injection load is 100%, an unrealistic stress-test scenario meant to exercise the network as much as possible (actual injection load ultimately controlled by CC protocol). As expected, runtimes are reduced as the MPI synchronization period increases, translating into fewer overall MPI synchronization events observed regardless of the CC protocol employed. We observe here that using the most coarse-grain sync period provides the best runtime improvement. We also demonstrate the relative overhead of simulating DCTCP over NewReno (~18-30% depending on sync. period). DCTCP’s overhead likely stems from the complexity of the algorithm reacting to ECNs. We attempted to repeat these experiments with the runtime optimizations disabled (i.e., fit-based mode using the same number of MPI ranks), but could not complete 50 µs within a week—clearly demonstrating that these techniques provide over 1000x speedup and simulating topologies at this scale would not otherwise be possible.

Analysis of DCN performance often breaks down into understanding both its steady-state and transient behavior. By examining the change (or lack thereof) in packets received and packets dropped, we can infer from Fig. 11 that the NewReno experiment achieves equilibrium around 10 ms while DCTCP reaches equilibrium prior to 10 ms. Moreover, we also observe that DCTCP transmits more packets (while incurring fewer dropped packets) than NewReno in the same simulated time span. This is expected given DCTCP’s use of ECNs for proactive congestion avoidance (see Section IV-A).

V. RELATED WORK

In this section we discuss related work in the area of simulation methodology. Table IV provides a high-level feature comparison between Fabsim-X and existing state-of-the-art network simulators, such as NS-3 [26], Booksim [27], SST [22], and CODES [18]. To the best of our knowledge, we highlight the breadth and scope of Fabsim-X’s capability and performance scale. We further discuss specific contrasting features and limitations of the most relevant research.

The popular choice simulator for networking analysis is NS-3, a discrete-event network simulator whose primary targets are research and educational use [26]. Although NS-3’s capabilities intersect with those of Fabsim-X, the former lacks efficient runtime scalability to effectively study large scale networks. In contrast to Fabsim-X where we have demonstrated significant scaling potential, a majority of the works published using NS-3 have peaked at a few thousand flows. Existing research has explored network simulation using real Linux network stacks via NS-3 [28]. However, support for this feature appears to have languished to a point where the capability is not easily resurrected, let alone maintained.

The Structural Simulation Toolkit (SST) [22] was developed to simulate highly concurrent systems where the Instruction Set Architecture (ISA), microarchitecture, and memory interact with the programming model and communications system. Although SST provides a high level of model configurability and the ability to evaluate large-scale lossless networks—primarily High Performance Computing (HPC)—it lacks sufficient detail of the various CC protocols of DCNs. More specifically, the very nature of a lossless network with its tight feedback loop and backpressuring minimizes the complexity and run time overheads that would otherwise be present when simulating a lossy network such as a modern DCN.

OMNeT++ [29] is an extensible, modular, component-based C++ simulation library and framework primarily for building network simulators such as wired and wireless communication networks, on-chip networks, and queuing networks. OMNeT++ is not a simulator itself as domain-specific functionality must be developed independently. One of them is INET [30], which is the model library for the Internet stack (TCP, UDP, IPv4, IPv6, OSPF, BGP), wired and wireless link layer protocols (Ethernet, PPP, IEEE 802.11, etc.), and many other protocols and components. INET is a detailed model that accurately captures endpoint behavior, but the lack of abstraction hinders usability for large-scale DCNs.

VI. CONCLUSION

In this paper, we present a simulation framework to evaluate performance in large scale DCNs. In particular, we demonstrate that it is capable of evaluating CC protocols on fat-tree topologies up to 128K endpoints in less than 150 hours. We describe how existing simulation methodologies are ineffective methods in achieving both the necessary scale and fidelity.
to enable in-depth analysis of network performance. Specifically, we demonstrate the techniques used, CC performance at various scales, and the interference impact of simultaneous heterogeneous protocols. Future work includes efforts in framework enhancements, validation, and advanced CC analysis. Specific examples of the former include validation and correlation of simulation data against hardware testbeds and identifying more optimal partitioning schemes. Efforts in advanced CC analysis include evaluating heterogeneous protocol interference at large scales, routing policies, and efficiency of using In-band Network Telemetry.

VII. ACKNOWLEDGMENTS

We thank John D. Thompson, Stephen Maresh, Naader Hasani and Mike Parker for their contributions.

REFERENCES

Effective Elastic Scaling of Deep Learning Workloads

Vaibhav Saxena
IBM Research, India
K. R. Jayaram
IBM Research, USA
Saurav Basu
Microsoft, India
Yogish Sabharwal
IBM Research, India
Ashish Verma
IBM Research, USA

Abstract—We examine the elastic scaling of Deep Learning (DL) jobs and propose a novel resource allocation strategy for DL training jobs, resulting in improved job run time performance as well as increased cluster utilization. We begin by analyzing DL workloads and exploit the fact that DL jobs can be run with a range of batch sizes without affecting their final accuracy. We formulate an optimization problem that explores a dynamic batch size allocation to individual DL jobs based on their scaling efficiency, when running on multiple nodes. We design a fast dynamic programming based optimizer to solve this problem in real-time to determine jobs that can be scaled up/down, and use this optimizer in an autoscaler to dynamically change the allocated resources and batch sizes of individual DL jobs.

We demonstrate empirically that our elastic scaling algorithm can complete up to $2 \times$ as many jobs as compared to a strong baseline algorithm that also scales the number of GPUs but does not change the batch size, with average completion times up to $10 \times$ faster.

Index Terms—elasticity, deep learning, variable batch size

I. INTRODUCTION

Many existing elastic scaling techniques [20], [8], [3], [12] are hard to apply to DL platforms/clusters because they focus on cluster elasticity — scaling the size of the cluster with VMs in response to workload variations. DL workloads and their system (hardware/software) stack present unique challenges for cluster elasticity. DL training is generally performed on parallel systems using high performance GPU accelerators, due to their efficiency in matrix computations and convolutions, and high (GPU) memory bandwidth. Also, many DL frameworks and workloads benefit from and are optimized for high speed interconnection networks (like NVIDIA NVLink, Infiniband, RDMA and 100G Ethernet). To make effective use of such interconnects, DL workloads need locality aware scheduling [18], i.e., learning processes in a distributed DL job have to be located as close to each other as possible. So, unless the entire datacenter has the same hardware (GPUs) and interconnect, it becomes hard to quickly scale a DL cluster. Often, one cannot scale the cluster by acquiring VMs, even GPU enabled VMs, because they can be provisioned anywhere in the datacenter. Cluster scaling with support for locality aware placement and scheduling requires manual intervention, installation and interconnect configuration taking several hours [18]. Second, there are often cost-related reasons why DL cluster scaling is undesirable.

However, DL training clusters, like other clusters, experience wide variations in workload and are often underutilized (leading to idle resources) or oversubscribed (leading to dropped/queued jobs). Hence, in this paper, we consider job elasticity, where jobs themselves are scaled to fit a fixed size cluster. We leverage a key characteristic of DL jobs – batch size – for job elasticity.

Batch size is the number of training samples used to determine the gradient at each update step of the gradient descent algorithm [4]. A small batch size may result in noisy gradients, significantly increasing the number of iterations to converge, while using large batch sizes has shown to result in poor generalization [19]. A peculiar characteristic of a DL job is that the final accuracy of the model remains similar if the training is done within a certain range of batch-sizes, called the optimal batch-size range (as demonstrated for a variety of models and datasets by [25], [9], [26], [28], [19], [7], [1]).

One such example (our reproduction) is shown in Figure 1 to motivate the techniques in this paper. The model is VGG11 which is trained on the CIFAR100 dataset for 160 epochs. The curve in red (Fixed BS) is a typical training with a fixed batch-size of 512 images throughout the training. The other three curves (Dynamic BS) show a training where the batch-size has been increased or decreased at random epochs, but within a range of 128-1024 images. We observe that both fixed BS and dynamic (varying) BS lead to similar final accuracy (± 0.5%).

Fig. 1. Similarity of final accuracy in DL jobs when the batch size (BS) varies within a specific range. Three BS variation patterns are used. start_epoch lists the starting epoch for the corresponding BS change.

Batch-size has a direct implication on the use of computing resources in the cluster and hence the time to complete the training process. In other words, by increasing the batch-size a higher number of GPUs can be leveraged to finish the training faster and vice-versa.

The goal and main contribution of this paper is to use permissible batch size ranges for performing efficient resource allocation and elastic scaling of DL jobs. To this end, this paper (and its companion tech report [24]) makes the following technical contributions:

1) A job scalability analyzer (JSA) to determine the scaling characteristics of DL jobs and their runtime with respect to the number of GPUs allotted and batch sizes used (Section II-B).
2) A fast, dynamic programming based optimizer that uses the JSA to allocate optimal resources and batch sizes to DL jobs in real time to maximize cluster throughput (Section II-C).
3) An autoscaler that uses the optimizer, cluster metrics and user input to elastically scale jobs using checkpoint-
resume (Section II-D). We demonstrate that the design and implementation of the autoscaler is independent of the middleware used to manage DL training jobs.

4) A detailed empirical evaluation (Section III) of the efficacy of our elastic scaling algorithm demonstrating that it is able to complete up to $\approx 2 \times$ the number of jobs in comparison to the baseline algorithm that does not consider or vary batch size. When queueing of jobs due to resource scarcity is not possible/desired, our elastic scaling algorithm drops up to $\approx 3 \times$ fewer jobs and completes jobs $\approx 10 \times$ faster when queueing is enabled.

II. DL-AWARE ELASTIC SCALING

A. Overview and Design Principles

Our mechanism is composed of three core modules - (i) a job scalability analyzer (JSA), (ii) an optimizer and (iii) an autoscaler. The optimizer’s goal is to allocate GPUs to the current set of jobs in the cluster in an (optimal) manner that maximizes cluster utilization and throughput (c.f. Section II-C for the precise formulation of throughput; informally it captures overall progress). The role of the JSA is to determine the scaling characteristics of the jobs in order to estimate their runtime so that the optimizer can make optimal scheduling decisions. The goals of the autoscaler are to (i) receive and buffer incoming jobs, (ii) periodically consult the optimizer to determine the set of jobs that can be allocated resources, and the number of GPUs to allot to each of the jobs in the set, and (iii) interact with the DL platform to scale jobs up or down.

![Fig. 2. Overview of our DL-aware elastic scaling mechanism](image)

A DL platform [13], [18], [24] is a multi-user, multi-tenant middleware for deep learning training. Architecturally, it resides above a cluster manager like Kubernetes and uses the cluster manager for deployment, placement, and failure detection of training jobs. Our mechanism is designed to be agnostic to the internals of specific DL platforms. The autoscaler, in conjunction with the runtime estimator and optimizer performs resource allocation; placement of jobs on nodes is performed by the DL platform and the cluster manager inside the DL platform.

Compared to existing DL job schedulers, our system also takes as input (i) (from the user) the permissible batch size range ($b_{\min} - b_{\max}$), and (ii) (from the user/admin) the maximum number of GPUs ($k_{\max}$) allowed for a job. Our system varies the batch size for a job (within the limits specified by the user) in order to maximize the cluster utilization. The range of acceptable batch sizes is typically well known for commonly used models. In many cases, users determine the range of batch sizes based on the acceptable time for completion of the job. In such cases, the JSA can also be used to predict the run-time for different batch-sizes.

B. Job Scalability Analyzer (JSA)

JSA’s goal is to determine the scaling characteristics of each job, so that the run time of the job can be estimated for various $(\# \text{ of GPUs, batch size })$ combinations as required. Certain scaling characteristics are job dependent whereas others are generic, i.e., dependent on the cluster. The JSA determines the generic scaling characteristics once at startup and repeats it infrequently to account for changes to the cluster hardware. Job dependent characteristics are determined whenever a new job arrives.

The run-time for a DL training job can broadly be segregated into two components: the processing time and the communication time. DL training being data parallel, a batch is typically divided equally amongst the GPUs and each GPU works on the part of the batch allocated to it; we refer to the size of the batch allocated per GPU as “batch-size-per-GPU” (computed as batch size divided by number of GPUs). As the processing happens in parallel over all the GPUs allotted to the job, the processing time is simply the time required to process batch-size-per-GPU on a single GPU. The communication time is the time required to reduce the gradients at the end of each iteration; this is dependent on the number of weights in the model and the number of GPUs.

Thus it suffices for the JSA to estimate: (i) for a given job, the processing time for various batch-size-per-GPU values on a single GPU; this is done by collecting certain job specific scaling characteristics. (ii) time for communicating different sizes of model weight buffers across a range of GPUs, i.e., the Allreduce-time for various combinations of # of weights and # of GPUs in the cluster; this is estimated using generic scaling characteristics.

1) Job specific scaling characteristics: To estimate the processing time:

1) The JSA considers every new job in the temporary holding buffer, executes it on a single GPU for a few iterations with different values of batch-size-per-GPU and records the average time taken per iteration. The methodology for estimating the runtime per iteration is similar to [27].

2) The batch-size-per-GPU values are chosen uniformly between $b_{\min}$ and $b_{\max}$ (the max. batch-size-per-GPU feasible for the job).

3) These scaling characteristics (average run-time per iteration) are tagged to the job metadata and the job is transferred to the temporary autoscaler buffer.

4) Note: Later, if the optimizer requires the processing time for a value of batch-size-per-GPU outside those considered in Step 2, it is determined by interpolating values computed in Steps 1-2 above.

Thus, this allows us to estimate the processing time $t_{j}^{\text{proc}}(b_{\text{gpu}})$ for a job $j$ when running with batch-size-per-GPU of $b_{\text{gpu}}$. 
2) **Generic scaling characteristics:** To estimate the AllReduce-time, the JSA performs the AllReduce operation over

- A range of GPUs \( \{1,\ldots,k_{\text{max}}\} \).
- for each number of GPUs \( k \in \{1,\ldots,k_{\text{max}}\} \), a range of model weight sizes (10M, 20M, 30M, ..., 100M weights).

Later, if the communication time is required for other values of number of weights, it is determined by interpolation of the available values. We note that it suffices to perform this estimation infrequently (it is not required to be done for every job; it can be performed whenever there are major changes to the cluster). Thus, this allows us to estimate the communication time \( t_{\text{comm}}(p,k) \) for a job having \( p \) weights when running on \( k \) GPUs.

The data communication time \( t_{\text{comm}}(p,k) \) for a particular job principally determines how well it scales. A higher proportional increase of \( t_{\text{comm}}(p,k) \) indicates a higher proportion of time is spent in reducing the gradients and hence less actual processing (or learning) by the JSA (this estimation functionality is provided by the JSA trimmer). This will ensure that the optimizer will never trim negative numbers. This will ensure that the optimizer will never trim negative numbers.

The per-iteration run-time for the job is then estimated as:

\[
t_j^{\text{iter}}(b,k) = t_j^{\text{proc}}([\frac{b}{k}]) + t_{\text{comm}}(p_j,k)
\]

Hence, the processing rate (number of input samples processed per unit time) for job \( j \) with batch size \( b \) on \( k \) GPUs is:

\[
T_j(b,k) = \frac{b}{t_j^{\text{proc}}([\frac{b}{k}]) + t_{\text{comm}}(p_j,k)}
\]

In order to make optimal scheduling decisions, the optimizer needs the processing rate estimate for a job \( j \) when scheduled with a specific batch-size, say \( b \), and number of GPUs, say \( k \). This estimate can be computed using the scaling characteristics collected by the JSA (this estimation functionality is provided by the JSA for use by the optimizer).

For configurations that are infeasible, for instance if batch-size-per-GPU \( \lfloor b/k \rfloor \) does not fit on a GPU, we take \( T_j(b,k) \) to be a large negative number. This will ensure that the optimizer will never consider this configuration in the optimal solution.

C. Optimizer

1) **Objective:** The optimizer has to determine, for each job, the number of GPUs to be allocated to the job and the batch size it should be run with, such that the overall throughput of the cluster is maximized. Every job must be allocated at least 1 GPU. If it is not possible to do so, the optimizer reports that the problem is infeasible.

The cluster throughput is determined as follows. For a job, \( j \), we treat \( T_j(b_{\text{max}}^j,1) \) as the baseline processing rate of the job, where \( b_{\text{max}}^j \) is the maximum batch-size-per-GPU that can be scheduled for the job. We now define the throughput scaling factor, \( T_j(b,k) \), to be the factor increase in processing rate obtained when the job is run with batch size \( b \) and \( k \) GPUs in comparison to the baseline, i.e.,

\[
T_j(b,k) = \frac{T_j(b,k)}{T_j(b_{\text{max}}^j,1)}
\]

We shall use the notation \( b_j^{\text{opt}}(k) \) to denote the optimal batch-size-per-GPU for job \( j \) when running on \( k \) GPUs, i.e., the one that yields the best throughput scaling factor; this can easily be determined as follows

\[
b_j^{\text{opt}}(k) = \arg \max_b T_j(b,k)
\]

Let \( J \) be the total number of jobs in the list obtained by the optimizer. Then the objective of the optimizer is to determine \( k_j \) for \( j = 1 \) to \( J \), i.e., the number of GPUs to be allocated to each job, so as to maximize the total throughput scaling factor of the jobs, i.e.,

\[
\text{maximize} \quad T = \sum_{j=1}^J T_j(b_j^{\text{opt}}(k_j),k_j)
\]

2) **Algorithm:** Given the objective function (3) for the optimization problem, a mixed integer program can be formulated to solve the problem. However, such programs can take very long to solve in practice depending on the number of jobs and GPUs. We show in our companion tech report [24] that the optimal solution to objective function (3) satisfies the optimal substructure property and thus admits a dynamic program.

Thus we can formulate a dynamic program (DP) to compute the optimal solution every time the autoscaler consults the optimizer by using a DP table for \( P(\cdot,\cdot) \) and populating the entries of this table iteratively using the following relation:

\[
P(j,K) = \max_{1 \leq k \leq k_{\text{max}}} \left[ \frac{P(j-1,K-k) + T_j(b_j^{\text{opt}}(k),k)}{1} \right]
\]

The optimal allocation \( k_j \) to the \( j \)-th job is thus given by:

\[
k_j = \arg \max_{1 \leq k \leq k_{\text{max}}} \left[ \frac{P(j-1,K-k) + T_j(b_j^{\text{opt}}(k),k)}{1} \right]
\]

A practical way to solve the dynamic program (DP) in (4) is to initialize the array \( P(\cdot,\cdot) \) to a large negative number and build the DP table progressively from \( j = 1, k = 1 \) to \( j = J, k = K \). Note from (4) that the optimizer might return infeasible for such a problem setting, which will mean that after the completion of the DP, \( P(J,K) \leq 0 \). In such a case, the autoscaler (next section) is responsible for figuring out a feasible solution by appropriately trimming the input job-queue to the optimizer.

The implementation of the dynamic program in (4) is shown in Algorithm 1. The entries \( P(0,K) \) of the DP table are initialized to 0 as the throughput is 0 for 0 jobs irrespective of the number of GPUs. All other entries of the DP table are initialized with a large negative number.

The solution array \( SOL(j,k) \) denotes the optimal GPUs allocated to the \( j \)-th job when the total GPUs allocated to all the \( j \) jobs is \( k \).
Algorithm 1 OPTIMIZER

Require: exec-queue $\mathcal{E}_t$, total GPUs $K$
\begin{align*}
\text{return} & \quad \text{Updated $\mathcal{E}_t$ with new GPU allocation, status} \\
J & \leftarrow \text{LENGTH($\mathcal{E}_t$)} \\
\mathcal{P} & \leftarrow -\infty_{j \times K} \quad \{\text{initialize $J \times K$ array of $-\infty$}\} \\
SOL & \leftarrow 0_{j \times K} \quad \{\text{initialize $J \times K$ array of $0$}\} \\
\mathcal{P}(0,:) & \leftarrow 0 \quad \{\text{no jobs mean zero utilization}\}
\end{align*}

for $j = 1, \ldots, J$ do
\begin{align*}
& \text{for } k = 1, \ldots, K \text{ do} \\
& \quad \text{if } k-g \geq 0 \text{ then} \\
& \quad \quad p \leftarrow \text{JSA.RECALL($\mathcal{E}_t(j), k$)} + \mathcal{P}(j-1, k-g+1) \\
& \quad \text{if } p > \mathcal{P}(j,k) \text{ then} \\
& \quad \quad \mathcal{P}(j,k) = p \quad ; \quad SOL(j,k) = k \quad \{\text{better utilization found}\}
\end{align*}

if $\mathcal{P}(J,K) > 0$ then
\begin{align*}
\text{status} & \leftarrow \text{"feasible"} \quad ; \quad j \leftarrow J \quad ; \quad k \leftarrow K \\
\text{while } j > 0 \text{ do} \\
& \quad \text{UPDATEGPU($\mathcal{E}_t(j), SOL(j,k)$)} \quad \{\text{set GPU allocation for job } j\} \\
& \quad j \leftarrow j-1 \quad ; \quad k \leftarrow k-SOL(j,k)
\end{align*}

else
\begin{align*}
\text{status} & \leftarrow \text{"infeasible"}
\end{align*}

$\mathcal{E}_t$ is the input job queue that is sent to the optimizer. The function call JSA.RECALL($\mathcal{E}_t(j), k$) looks up the value of the throughput scaling factor $T_j^g(k)P_j^g(k)$ for the j-th job $\mathcal{E}_t(j)$ in the job-queue when run on k GPUs.

It is easy to see that the run-time complexity of the dynamic program is $O(JKk_{max})$ based on the three for loops. A problem will only be feasible if the number of jobs is no more than the number of GPUs. Thus, we can assume that $J \leq K$. Therefore, even for 400 GPUs and $k_{max} = 10$, the complexity is no more than an order of 2M operations (milliseconds on modern CPUs).

If a feasible solution exists ($\mathcal{P}(J,K) > 0$), each job in the queue will have their respective GPU allocation field updated. Otherwise, an ‘infeasible’ status is sent back to the autoscaler and the existing GPU allocation for all the jobs in the input queue is left untouched.

D. Autoscaler

The autoscaler maintains a list (EXECUTING) of jobs that are currently being executed by the DL platform. It queues all arriving jobs in another list (ARRIVED). Each job in the queue is first processed by the JSA. The JSA computes the scaling characteristics of the job as described in Section II-B and adds these characteristics to the job metadata. The autoscaler is also notified by the DL platform whenever a job completes (or fails due to user error) and stores these jobs in a list (FINISHED), removing them from the EXECUTING list before invoking the optimizer. To be clear, the optimizer will be invoked even if no new job arrives, but jobs leave.

Ideally, the autoscaler should invoke the optimizer every time a job arrives or leaves. However, in practice, this can lead to thrashing – a significant amount of time is spent scaling DL jobs up and down through checkpoint/resume that it affects the progress made by the DL job. Consequently, in practice, the autoscaler consults the optimizer periodically (say every $\triangle$ minutes). $\triangle$ is typically based on the amount of time a DL job can be queued without affecting user experience and expectations.

Every $\triangle$ minutes, the autoscaler invokes the optimizer to determine (i) whether GPUs can be allotted to as many newly arrived jobs (from ARRIVED) and (ii) how many GPUs are to be allotted to each job. This is done in an iterative and incremental manner – first by removing jobs that have terminated, and then trying to add jobs from ARRIVED one by one until the optimizer returns ‘infeasible’. Once the set of jobs that can be executed and the allocations to said jobs are determined, the autoscaler interacts with the DL platform to spawn new jobs (using its CREATE API), and scale existing jobs up or down (using the HALT and RESUME API). Hal/Resume typically involves checkpointing the current state of the job and restarting with the new resource allocation (details in [17] & [24]). The autoscaler can also be modified to drop (i.e., reject) pending jobs (which are not feasible to accommodate) if queuing is undesirable. We refer the reader to our techreport [24] for the Autoscaler pseudocode.

E. Simulator Design and Implementation

We develop a simulator in order to evaluate our techniques on large cluster settings; given the difficulty in getting access to large GPU-enabled clusters for long periods of time. The simulator is based on the discrete event simulation (DES) [22] methodology, wherein the simulation is driven by time based events. An event is created in the simulator whenever a job arrives or completes.

The input to the simulator is a job arrival file that contains metadata for the jobs along with their arrival time. The metadata includes the job details (such as maximum batch size, minimum batch sizes, number of epochs to process, etc.) along with the job specific scaling characteristics collected by the job scalability analyzer (JSA) (see Section II-B). Recall that the JSA runs the jobs for only a few iterations up to $k_{max}$ GPUs to collect the job specific scaling data.

The simulator reads the job arrival information from the file to create arrival events for the jobs based on their arrival time. Based on the schedule returned by the optimizer, the simulator updates the GPU allocations to the jobs and updates their completion time based on the run-time estimates as determined in Section II-B. It also adds a completion event for any new job that has been scheduled. In case the optimizer returns that no allocation is feasible, then the newly arrived job is either dropped or put into a queue depending on the configuration of the simulator; the simulator can be run in both modes – with and without queuing. In case of queuing, the first job from the queue is considered for execution on the next job completion event; it is treated like a fresh job and the schedule is determined accordingly. The simulator then proceeds to process the next event until all the events are processed.

III. EXPERIMENTAL EVALUATION

We used a real 40 GPU cluster (each machine had 2 P100 GPUs, 16 Intel Xeon E5-2620 v4 2.10 GHz cores, 64 GB RAM and 10Gbe uplinks) to both evaluate our elastic scaling techniques and validate the simulator (of Section II-E). We then use the validated simulator for large-scale experiments (400 GPU cluster). Experiments conducted on the actual cluster will be tagged with the “Cloud” label when results are presented or plotted, and simulated ones with the “Simulator” label. We present key results here; additional results are available in our tech report [24].
A. Experimental Setup

<table>
<thead>
<tr>
<th>Category</th>
<th>Dataset Model</th>
<th>Weight Size</th>
<th>Min &amp; Max BS</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CIFAR100</td>
<td>24M</td>
<td>16, 256</td>
<td>Elastic Compute Bound</td>
</tr>
<tr>
<td>2</td>
<td>CIFAR100 Alexnet</td>
<td>58M</td>
<td>10M</td>
<td>Elastic Communication Bound</td>
</tr>
<tr>
<td>3</td>
<td>CIFAR100 VGG11 bn</td>
<td>10M</td>
<td>128, 128</td>
<td>Elastic Balanced</td>
</tr>
<tr>
<td>4</td>
<td>Food101</td>
<td>58M</td>
<td>128, 128</td>
<td>No Elasticity</td>
</tr>
</tbody>
</table>

Table I: Job Category Information

The neural network model and the dataset of a job determines the communication vs compute characteristics of the job. We use four categories of jobs described in Table I. A compute bound job scales well with larger number of GPUs compared to a communication bound job which scales poorly due to communication overhead - this has been captured in the throughput scaling factor calculation for a job in Section II-C1. The jobs in a benchmark arrive in one of three patterns: low arrival, high arrival or bursty arrival over a 240/480/720 minute period (will be clear from context).

Job arrivals have been generated by sampling from a Poisson distribution with mean job arrival rate as the design parameter. A base job arrival rate $\lambda$ is defined as the expected completion rate (i.e., reciprocal of expected completion time) of a job sampled uniformly from the 4 categories, on a single GPU with maximum batch size per GPU. A high job arrival rate signifies that the mean arrival rate of the Poisson distribution has been set to $k_{max}\lambda$ where $k_{max}$ is the maximum GPUs allocated to each job. A low arrival rate sets the mean of the Poisson distribution to $\frac{k_{max}}{4}\lambda$. A bursty arrival rate alternates the mean of the Poisson distribution between high and low (defined previously) every 60 or 120 mins as required by the experiment. The job lengths (time) for categories 1, 2, 3 and 4 when scheduled on a single GPU are 16, 21, 41 and 27 mins respectively; these are determined from practical settings involving incremental training. This provides a good mix of jobs from different categories with different run times.

B. Choosing the Baseline

A completely non-elastic scheduler can force a constant total batch size on a fixed number of GPUs through the job lifetime; however, that is evidently a weak baseline.

For a stronger baseline, we fix the total batch size of a job to be a value selected from the range of allowed batch sizes for that job category, but we allow the total batch size of the job to be distributed over several GPUs; this models a traditional elastic scaler and does not take into account the characteristic (batch size range) of the DL job. The allocation of GPUs to baseline jobs is done through the same optimizer that is used in our elastic scaling technique. The optimizer is invoked periodically (every $\Delta$ minutes) to allocate the relevant number of GPUs for baseline jobs based on their throughput scaling factors such that the total batch size remains fixed across the allocated GPUs. To reiterate, our baseline is also elastic in terms of total number of GPUs but doesn’t vary the total batch size. Jobs may be run with: maximum batch-size (Max-BS) for that category, minimum batch-size (Min-BS) for that category, and random batch-size (Random-BS) which is a value picked uniformly randomly between the Max-BS and Min-BS for that category. Running all jobs either with max-BS or min-BS is unlikely to reflect real-world scenarios (based on our experience deploying our techniques at IBM). Hence, this paper focuses on the random-BS baseline, and comparison with min-BS and max-BS baselines are deferred to our techreport [24].

C. Metrics

Unfortunately, existing metrics for evaluating elasticity [6], [15], [14] have been designed for and apply only to cluster elasticity. We therefore define new metrics based on:

- **Optimal GPU time for all scheduled jobs** (Opt_Sch_Time). This is defined as the sum of job lengths (time) for all the scheduled jobs on a single GPU. This is the minimum GPU-time needed to complete all the scheduled jobs (GPU-time means actual runtime on GPU, and we expect imperfect scaling).

- **Actual GPU time for all scheduled jobs** (Act_Sch_Time). This is defined as the sum of [(actual number of GPUs used for running the jobs) x (Time duration for which these GPUs were used)]

For example, if a single scheduled job requires 10 mins to complete on a single GPU but requires 6 mins to complete on 2 GPUs, then its Optimal GPU time will be 10 mins whereas the Actual GPU time will be 12 mins (=6x2).

We measure and report the performance in terms of two metrics (i) **Scheduled Job Scaling (SJS) Efficiency**, defined as $\text{Opt\_Sch\_Time} / \text{Act\_Sch\_Time}$, and (ii) **Job Drop Ratio**, defined as $\text{Number of jobs dropped} / \text{Total number of jobs}$. The first metric shows the average scaling efficiency of all the scheduled jobs i.e. how well the scheduled jobs scale across GPUs. The second metric shows the proportion of jobs that have been dropped.

D. Effect of different job categories (Figure 3).

Figure 3 illustrates the total number of jobs completed for various job categories (Section III-A) and hence job characteristics. We observe that the results of the simulator is quite close to the real cluster. We also observe that all the plots, except for Category 4, show that the elastic approach is more effective as it demonstrates a significant improvement in the number of jobs completed in comparison to the baseline. The number of jobs completed improve by 82%, 64.4% and 90% for Categories 1, 2 and 3 respectively (Figure 3). The reason, as discussed before, is that with high arrival rate the baseline is able to schedule only a limited number of jobs as the batch-size is high and thus the jobs cannot be scaled down to a small number of GPUs. Our elastic scaling algorithm, however, can reduce the batch size of the problem itself to fit the minimum number of GPUs feasible with the minimum batch-size specified. Thus it minimizes the number of dropped (rejected) jobs. To further support this argument, we examined the best throughput scaling factors for category 1 (compute bound) and category 2 (communication bound) jobs with their minimum batch size. We
observed that the best throughput scaling factor for the category 1 job was 1.3x the best throughput scaling factor for the category 2 job.

For the 4th category (Figure 3), results for both the baseline and our method match with each other. This is because in this category, the jobs are non elastic, i.e. the batch size cannot be changed, and hence our elastic algorithm does not perform better than the baseline.

E. Effect of Arrival patterns. (Figure 4)

Having discussed high arrival rate in the previous section, in this section we discuss the effects of low and bursty arrival rates. For the case of low arrival rate in Figure 4, the number of jobs completed are more by 97% ($\approx 2 \times$) with the elastic approach in comparison to the baseline. This is because both the algorithms try to maximize the GPUs used; however, the elastic scaling algorithm is able to increase the batch size as it scales to larger number of GPUs whereas the baseline algorithm cannot. As the scaling behaviour is better when there is more work to be done by every GPU, the scaling algorithm is able to achieve better scaling resulting in quicker completion of the jobs. This better scaling results in reduced drop rate. To further support this argument, we present the throughput scaling factors for a category 1 jobs on 2 GPUs with different batch-size-per-GPU in Table II. It is clear that the throughput scaling factor increases monotonically as the batch-size-per-GPU is increased.

When the arrival pattern is bursty (again Figure 4), the number of jobs completed by the elastic approach is 119% ($\approx 2.2 \times$) more in comparison to the baseline. This is due to a mix of two effects that have already been discussed before. During periods of low arrival rate, the elastic scaling algorithm can increase the batch size as it scales to larger number of GPUs to get better scalability. On the other hand, during periods of high arrival rate, the elastic scaling algorithm is able to reduce the batch-size in order to fit on fewer number of GPUs, thereby accommodating more jobs. We also observe that the simulator closely match the cloud results.

F. Effect of Queuing (Figure 5)

We next study the effectiveness of our algorithm with queuing; no jobs are dropped (rejected) in this case. We consider a longer bursty job arrival pattern wherein the jobs arrive for 720 mins (12 hours). The results for this experiment without and with queuing are presented in Figure 5. We observe that the simulated results are consistent with the actual results on the cloud. Our elastic approach performs much better than the baseline; the number of jobs completed is 50% more than the baseline without queuing and almost matching the job arrival rate with queuing. As explained

<table>
<thead>
<tr>
<th>Batch size per GPU</th>
<th>8</th>
<th>11</th>
<th>16</th>
<th>22</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scaling Factor</td>
<td>0.86</td>
<td>1.06</td>
<td>1.3</td>
<td>1.45</td>
<td>1.66</td>
</tr>
</tbody>
</table>

TABLE II
THROUGHPUT SCALING FACTORS FOR CATEGORY 1 JOBS ON 2 GPUS
before, during periods of low arrival rate, the elastic scaling algorithm can increase the batch size as it scales to larger number of GPUs to get better scalability and during periods of high arrival rate, the elastic scaling algorithm is able to reduce the batch-size in order to fit on fewer number of GPUs, thereby accommodating more jobs.

Analyzing the baseline more carefully, we see that without queuing the baseline curve tends to flatten (reduction in slope) during periods of low arrival indicating that it is able to schedule all the jobs. On the other hand, with queuing the baseline curve remains straight (same slope) throughout indicating that it is scheduling jobs that have accumulated in the queue during the period of high arrival. In contrast, the elastic scheduling curve flattens in both settings, with and without queuing, indicating that it is able to service all the queued jobs. Thus, the baseline ends up taking considerably long to complete all the jobs. This is further reflected in the average job completion time for the jobs presented in Table III. The average job completion time increases by $\sim 35\%$ (24.97 mins to 33.79 mins) for the elastic scaling algorithm whereas it increases by nearly 10X (34.12 mins to 351.02 mins) for the baseline algorithm. The average job completion time for the baseline is nearly 10X that of the elastic scaling algorithm.

![Graph](image)

### Table III

<table>
<thead>
<tr>
<th></th>
<th>SJS Efficiency (%)</th>
<th>Job Drop Ratio (%)</th>
<th>Avg. Job Compl. Time (mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actual Simulator</td>
<td>82.02</td>
<td>42.4</td>
<td>34.12</td>
</tr>
<tr>
<td>Elastic-withdrop</td>
<td>80.46</td>
<td>39.57</td>
<td>27.75</td>
</tr>
<tr>
<td>Elastic-nodrop</td>
<td>89.53</td>
<td>-</td>
<td>351.02</td>
</tr>
<tr>
<td>Baseline-withdrop</td>
<td>51.31</td>
<td>42.4</td>
<td>28.10</td>
</tr>
<tr>
<td>Baseline-nodrop</td>
<td>42.87</td>
<td>-</td>
<td>320.32</td>
</tr>
</tbody>
</table>

**Performance Metrics for 400 GPUs with and without Job Drops (Random-BS Baseline)**

We finally evaluate the efficacy of our algorithm on a larger setup comprising of 400 GPUs using the simulator (Figure 6). Here, the jobs arrive with a bursty arrival pattern for 480 mins (8 hours). The observations are quite similar to those of the 40 GPU experiments. Our elastic approach performs much better than the baseline; the number of jobs completed is 60% more than the baseline without queuing and almost matching the job arrival rate with queuing. As in the case of 40 GPUs, we see that without queuing the baseline curve tends to flatten during periods of low arrival, whereas with queuing, the baseline curve remains straight; this indicates that the baseline cannot schedule all the jobs in the queue during periods of low arrival. In contrast, the elastic scheduling curve flattens in both settings, with and without queuing, indicating that it is able to service all the queued up jobs. We observe from Table IV that the average job completion time is almost the same for the elastic scaling algorithm whereas it increases by nearly 6X (27.84 mins to 166.82 mins) for the baseline algorithm.

### IV. Related Work

The bulk of existing research (e.g., [3], [12], [14], [23], [20], [8], [11], [15], [16]) on elasticity has focused on cluster elasticity. Mechanisms for cluster elasticity do not adapt the workload to best use a fixed (or semi-fixed) set of resources. Programming elasticity directly into the application, i.e., making the application self-adaptive by monitoring its environment [15], [16], [23], [5], potentially applies to both cluster and job elasticity. However, aforementioned research [15], [16], [23], [5] is limited to cluster elasticity.

Project Philly/Fiddle [18] is a DL platform internal to Microsoft that enhances YARN’s scheduler to support locality-aware scheduling and gang scheduling. However, elastic scaling is not supported. The SLAQ framework [29] explores quality-runtime tradeoffs across multiple ML training jobs to maximize system-wide quality by adjusting resource allocations of all running jobs. The intuition behind SLAQ is that in the context of approximate ML training, more resources should be allocated to jobs that have the most potential for quality improvement. Litz[21] supports
API driven programming of ML constructs, specifically elastic addition/deletion of parameter servers and worker shards. Its benefit is that it can handle elasticity of individual jobs seamlessly (without restarting), but the elasticity decisions (when to scale) are still manual, and there is no concept of an autoscaler that decides when to scale to improve cluster level throughput. Tiresias [10] proposes a GPU cluster manager for distributed DL jobs which efficiently schedules and places DL jobs to reduce their job completion times (JCT). Their scheduling system assign priorities to jobs in order to reduce their JCT. It doesn’t change the batch size for reducing JCT. OASIS [2] describes scheduling in a machine learning cluster that forecasts resource availability and uses that in conjunction with the utility of arriving jobs to arrive at a resource configuration for a particular job. In addition, an admission policy for the job is also prescribed in order to prioritize most useful jobs.

Gandiva [27] exploits intra-job predictability of mini-batch iterations to time-slice GPUs efficiently across multiple jobs in order to improve cluster efficiency. It also dynamically migrates a communication intensive job to preserve affinities between learners. Gandiva also supports limited elastic scaling by allowing a learner in a DL training job to occupy all GPUs on a machine, but does not alter hyperparameters while scaling.

V. CONCLUSIONS

We have demonstrated that (i) to effectively scale deep learning jobs, we have to determine whether they are compute or communication bound and leverage the fact that they can be executed with a range of batch sizes in addition to considering other factors like cluster utilization, (ii) the exploration of a range of batch sizes can be formulated as an optimization problem, which also considers the scaling efficiency of individual DL jobs when run on multiple resources, (iii) this problem admits a dynamic programming based solution which makes it efficient (for real-time use from autoscalers) and scalable (to typical DL clusters with hundreds of GPUs), and (iv) our elastic scaling techniques have significant performance benefits in most scenarios (up to 2× increase in job completions and 10× better job completion time).

REFERENCES

Merkle Hash Grids Instead of Merkle Trees

Jehan-François Pâris
Department of Computer Science
University of Houston
Houston, TX, USA
jfparis@uh.edu

Thomas Schwarz, SJ
Department of Computer Science
Marquette University
Milwaukee, Wisconsin
thomas.schwarz@marquette.edu

Abstract—Merkle grids are a new data organization that replicates the functionality of Merkle trees while reducing their transmission and storage costs by up to 50 percent. All Merkle grids organize the objects whose conformity they monitor in a square array. They add row and column hashes to it such that (a) all row hashes contain the hash of the concatenation of the hashes of all the objects in their respective row and (b) all column hashes contain the hash of the concatenation of the hashes of all the objects in their respective column. In addition, a single signed master hash contains the hash of the concatenation of all row and column hashes. Extended Merkle grids add two auxiliary Merkle trees to speed up searches among both row hashes and column hashes. While both basic and extended Merkle grids perform authentication of all blocks better than Merkle trees, only extended Merkle grids can locate individual non-conforming objects or authenticate a single non-conforming object as fast as Merkle trees.

Keywords—Merkle trees, authentication

I. INTRODUCTION

Merkle trees, also known as hash trees, [10–13] are a fundamental building block in many sophisticated authentication schemes. In its basic functionality, a Merkle tree authenticates a set of blocks in a file that is transmitted through time in a storage system or through space in a messaging system. As Fig. 1 shows, a Merkle tree consists of as many leaves as there are blocks in the file we want to test and as many internal nodes as required by the degree of the tree. Each leaf contains the hash of one of the file blocks and each internal node contains the hash of the concatenated values of its children. As a result, any change in any file block will affect the value of the root of the tree. Thus signing that root suffices to authenticate the whole tree. Its strength relies on the strength of the hash function, i.e. the incapability to invert a hash with realistic resources and the strength of the signature.

Building a binary Merkle tree for a file that contains \( N = 2^n \) blocks requires \( 2N - 1 \) hashes and one digital signature. In comparison, identifying a single non-conforming block is a relatively inexpensive operation as it only require \( 2 \log_2 N = 2n \) comparisons.

When we want to verify the contents of a local copy of a file, we construct the Merkle tree of that copy and compare its root with the root of the Merkle tree of the remote replica. If they are identical, we know (with overwhelming probability) that the two copies are identical. If they are not, identifying each non-conforming block requires a traversal of the tree.

II. MERKLE GRIDS

We introduce Merkle grids, a two-dimensional organization that has a smaller construction cost than the corresponding Merkle tree and only requires \( 2 \sqrt{N} + 1 \) comparisons to identify all blocks that are likely to be non-conforming. As we will see later, we can even reduce the number of comparisons needed to

![Fig. 1. A Merkle tree. \( N = 2^n \) blocks are arranged in a line. They are represented by the gray, rounded rectangles. We adorn each block with its hash (the light gray square). The arrows indicate taking the hash after concatenation. The root (dark blue) is a signed hash.](image1)

![Fig. 2. The base version of our scheme. \( N = n^2 \) objects are arranged in a square (represented by the gray, rounded rectangles). Light gray squares represent hashes. For each row and each column of hashes, we calculate the hash of the concatenation of the hashes in the line (the lighter gray squares on top and on the right). Finally, we calculate the hashes of each row and each column and sign it (the dark blue square in the upper right corner).](image2)
locate individual non-conforming blocks to $2\log_2 \sqrt{N} + 1$ by adding two auxiliary Merkle trees to our grid.

As Fig. 2 shows, Merkle grids organize the blocks of the file we want to verify in a square n-by-n array where $n = \sqrt{N}$ is the square root of the total number of blocks in the file. Rectangular arrays and arrays with incomplete rows or columns are possible but are somewhat less efficient. Each row of the array has a row hash containing a hash of the concatenated hashes of all blocks in that row. In the same way, each array column has a column hash containing a hash of the concatenated hashes of all blocks in that column. More formally, if $b_{ij}$ denotes the block at the intersection of the $i$-th row and the $j$-th column of the matrix, row hash $r_i$ and column hash $c_j$ are defined as

$$r_i = h(h(b_{i1}), h(b_{i2}), ..., h(b_{in}))$$

$$c_j = h(h(b_{1j}), h(b_{2j}), ..., h(b_{nj})).$$

where the dot . denotes concatenation. In addition, a master hash $M$ contains a hash of the concatenated values of all row and column hashes. In other words.

$$M = h(r_1, ..., r_n, c_1, ..., c_n).$$

Unlike the row and column hashes, this master hash is signed. We can thus see that building a Merkle grid requires $N + 2\sqrt{N} + 1$ hashes and one digital signature verification. This is significantly less than the number of hashes of a comparable Merkle tree and the space savings gets closer to 50 percent as the file size increases as

$$\lim_{N \to \infty} \frac{N + 2\sqrt{N} + 1}{2N - 1} = 0.5.$$

To verify the contents of a file, we first construct the Merkle grid of the local copy of the file and compare its master hash with the master hash of the remote copy. If they match, we know that the two copies are identical. Otherwise, we compare the row and column hashes of the local copy with the Merkel grid of the remote copy. The operation will require $2\sqrt{N}$ additional comparisons and will detect all blocks that are likely to be non-conforming. The main advantage of our approach is that it detects all suspect blocks in a single sweep. Its main disadvantage lies in the additional diagnostic work it requires in the presence of several non-conforming blocks. If there are two non-conforming blocks, they are likely to be located in two different rows and columns. Since there are four potential non-conforming blocks in the intersections of the two rows and the two columns, we need to check these four blocks for non-conformity.

A. Extended Merkle grids

While basic Merkle grids require up to 50 percent fewer hashes and occupy up to 50 percent less space than Merkle grids, their search performance is much less impressive as they require $2\sqrt{N}$ hash comparisons to locate a non-conforming block while Merkle trees only require $2\log_2 N$ comparisons.

A simple but effective way to reduce this number of hash comparisons is to add two auxiliary Merkle tree to the grid. As Fig. 3 shows, the first of the two trees is built upon the $\sqrt{N}$ row hashes while the second is built upon the $\sqrt{N}$ column hashes. As a result, both trees counts $2\sqrt{N} - 1$ nodes and the Merkle grid now counts a total of $N + 4\sqrt{N} - 1$ hashes, that is, $\sqrt{N}$ block hashes, $2\sqrt{N} - 1$ hashes in each auxiliary Merkle tree, and a single signed hash of the concatenated root of these two trees. Even so, the completed Merkle grid occupies up to 50 percent less space than a comparable Merkle tree as

$$\lim_{N \to \infty} \frac{N + 4\sqrt{N} - 1}{2N - 1} = 0.5.$$ 

The main advantage is that the number of comparisons required to identify a non-conforming block becomes $4\log_2 \sqrt{N} + 1$, which simplifies into $2\log_2 N + 1$, that is, one more comparison than with a Merkle tree.

III. MERKLE TREES REVISITED

A Merkle tree is a binary tree whose interior nodes are hashes and whose leaves are the blocks to be authenticated. The security of the Merkle tree depends directly on the security of the hashes, and in particular their collision resistance. The blocks could be messages, sensor data, blocks or objects in a file system, or any other type of large binary object. We give an example of a Merkle tree in Fig. 1. That Merkle tree has 16 objects (some of which could be null objects) represented by the rounded rectangles. The lighter gray squares represent hashes. The first level hashes are just the hashes of the objects. All other hashes are calculated recursively from the hashes of the two children of each node, namely the hash of the parent is the hash of the concatenation of the children. Formally, if $p$ is the content of the parent and $l$ and $r$ are the contents of the children, the dot . denotes concatenation and $h(\cdot)$ is our hash function, then

$$p = h(l, r).$$

The root of the tree is signed using a public signature. If a sender sends all objects, then the receiver would build the
of hashes in a Merkle tree with fan-out $k$ and height $n$, hence for $k^n$ blocks, is $(k^{n+1} - 1)/(k - 1)$. The number of hashes per block is this value divided by $k^n$, which has limit 1. The authentication of a single block uses now $kn + 1$ hashes as at each level with the exception of the top level, we access $k$ hashes.

Until now, we implicitly assumed that Merkle trees are just used for authenticating a file. In other words, the generic case to be optimized is that all blocks are true and have not been changed, be it by an attacker or by accident. We contend that for a large set of applications that use Merkle tree this is not the case. There are transmission errors and there are errors in storage devices that result in a device returning a different block than the one that was stored, for example, because of a write misdirect. These errors might be more frequent than attacks and failure of authentication because corruption then becomes the normal case. We do not deny that a malicious attack once detected has to be taken much more seriously. We deal with this scenario by talking about a sender, the entity that signed the hash, and the receiver, the entity that verifies the hash of the root node.

Assume that a single block out of the $k^n$ blocks protected by a Merkle tree of height $n$ and degree $k$ is corrupted. Sender and receiver will then build different Merkle trees and the signed master hash sent by the sender does not agree with the locally constructed master hash of the receiver. Once the discrepancy between the sender’s and the receiver’s root hashes has been detected, the receiver asks for the $k$ hashes at the level below the root. Since only a single block does not conform, only one of these $k$ hashes differs between sender and receiver. The receiver now recursively proceeds by asking for the $k$ child hashes of the non-conforming hash, and so on. The verification takes the same path as the authentication of a single block.

If two of the objects are corrupted, then we do not usually need to access twice the number of hashes. We first treat the case of a classic Merkle tree with fan-out 2 and height $n$. The exact number of accesses to hashes depends on luck, for example, if the first and the second block are corrupted, then we only need to access $2n$ hashes. We argue inductively. Let $d_n$ denote the expected number of hashes and assume we have a Merkle tree of height $n$ and that we know that the root hash does not match. In this case, we access the two child hashes. We know that at least one of the two child hashes is off, but we also need to consider the case where both are off. In the case illustrated at the left of Fig. 6, the two corrupted objects are the leaves of the left subtree rooted in the hash with the non-conforming hash. The probability of the two peccant block being located in two different halves is

$$p_{1,1} = \frac{(2^{n-1})^2}{\binom{2^n}{2}} = \frac{2^{n-1}}{2^n - 1},$$

which is of course almost $\frac{1}{2}$. In this case, we have read two hashes (the children of the root) and also need to find the two corrupted objects in a tree of height $n - 1$. Thus, in total, we need $2 + d_{n-1}$ accesses. Conversely, if we are in the case illustrated at the right of the figure, we need to find one non-conforming object in each of the two sub-trees of height $n - 1$, so that together with the two child hashes, a total of $2 + 2 \cdot d_{n-1}$.
\[ 2(n - 1) = 4n - 2 \] hashes need to be accessed. If the height is one, then there are just two objects and we access two hashes. This gives us the recursion

\[ a_n = 2 + \frac{2^{n-1}}{2^n - 1} 4(n - 1) + \left(1 - \frac{2^{n-1}}{2^n - 1}\right) a_{n-1}. \]

This recurrence is solved by

\[ a_n = 2 \left( \frac{1}{2^{n-1} - 1} + 2 \right) n - 4, \]

which can be approximated for large \( n \) by \( 4(n - 1) \). In the case of \( k \)-ary Merkle trees, the probability that the two non-conforming blocks are located in the same of the \( k \) sub-trees of the root is

\[ \left(\frac{k^{n-1} - 1}{k^{n} - 1}\right). \]

If the non-conforming blocks are located in two different subtrees, then we need to access \( 2k(n - 1) \) hashes. If they are located in the same subtree, then we use recursion. This gives a recurrence

\[ b_n = k + \frac{k^{n-1} - 1}{k^{n-1} - 1} b_{n-1} + \left(1 - \frac{k^{n-1} - 1}{k^{n-1} - 1}\right) k(n - 1). \]

Its solution is

\[ b_n = k\left(-k + n + 4\right) - 3 \frac{k^{n-1} - 1}{k^{n-1} - 1} + k \left(2n - \frac{k}{k - 1}\right), \]

which is approximately \( 2k(n - 1) \) for large \( n \). Fig. 7 shows the similarity to a linear function very well.

We can extend this procedure to the case of three non-conforming objects, at least for binary Merkle trees. Thus, we assume the existence of \( 2^n \) objects but require \( n \geq 2 \) in order to have at least three objects. Again, we assume that we know that the root hash does not match. The probability that the three corrupt objects are in the same subtree with \( 2^{n-1} \) leaves is

\[ p_{3,0} = 2 \left( \frac{2^{n-1}}{3} \right) = \frac{2^{n-1}}{4(2^{n-1})}, \]

which is of course zero for \( n = 2 \), but then very quickly converges to \( \frac{1}{4} \). The only other case is that of the three objects, two are in one sub-tree and the remaining one in the other, which happens with probability

\[ p_{2,1} = 2 \frac{2^{n-1}}{2^n - 1} \left( \frac{2^{n-1}}{3} \right) = \frac{2^{n-2}}{2^n - 1} \]

which starts out at 1 for \( n = 2 \) and then quickly sinks to \( \frac{1}{4} \).

If we denote by \( \tau_n \) the number of hashes accessed in the case of three corrupt objects in a Merkle tree of \( 2^n \) objects, we can now first develop a recurrence equation and then obtain its solution. Again, we assume that \( \tau_n \) does not include the root hash, which is known to be non-conformant. We then access the roots of the two subtrees. According to the probabilities just calculated, either three of the non-conforming objects are in the same subtree or they are divided \( 2 + 1 \) among the sub-trees. In the latter case, we have already determined the expected number of the hashes, in the former case, we proceed by recursion. This gives us

\[ \tau_n = 2 + p_{3,0} \tau_{n-1} + p_{2,1} (a_{n-1} + 2(n - 2)). \]

This complicated recurrence relation has a surprisingly simple solution, namely, \( \tau_n \) equals

\[ 2\left(9 \cdot 2^{n} - 9 \cdot 2^{n+1} n + 6n + 9 \cdot 2^{n+2} - 7 \cdot 2^{2n+1} - 22\right) \frac{3(2^n - 2)(2^n - 1)}{3}. \]

In an overabundance of caution, we used simulation to verify the result.

Finally, we treat the behavior of binary Merkle trees with \( 2^n \) leaves in case of a burst error of size \( l \). While it is possible to give a recurrence relation for the number of hashes, this approach does not provide additional insight over a reasonable set of examples. We wrote a Python script that counts exactly the number of hashes accessed for various values of \( n \) and \( l \) and their position and then calculates the exact expectation under the assumption that all \( 2^n - l \) possible positions of the burst are equally likely. While this is true only in certain scenarios, the results remain typical. The results are given in Fig. 8, which shows that the behavior is essentially linear. For small values of \( n \) and large values of \( l \), the number of hashes approaches the total number of hashes. This explains why the curves overlap at their onset.

IV. PERFORMANCE EVALUATION

In this section, we evaluate the number of hashes required to identify non-conforming blocks in both basic and extended
Merkle grids and compare these values with those obtained in the previous section for Merkle trees. Strictly speaking, we should make a distinction between taking the hash of objects and the hash of hashes as the latter are considerably smaller.

### A. Basic Merkle Grids

Let us consider first the case when we have a single non-conforming block to locate. In that case, the faulty block can be uniquely identified by its row and its column indices and the total number of hashes we need to compute is the total number of row and column hashes, that is, $2\sqrt{N}$ or $2n$.

If the file contains two non-conforming blocks, we still need to compute the hashes of the $2n$ line hashes but they are not necessarily enough to identify the two faulty blocks. We have two subcases to consider:

**Case 1:** The two non-conforming blocks share either a common row or a common column. In that case, we can uniquely identify the two blocks by their row and column indices.

**Case 2:** The two non-conforming blocks do not share a common row nor a common column. In that case, the row and column indices identify four potential non-conforming blocks without telling us how many of these blocks are faulty. Identifying the two non-conforming blocks will thus require computing four extra hashes. There are $\binom{n^2}{2}$ distinct ways the two non-conforming blocks can be located on the grid and $2\binom{n}{2}^2$ distinct ways these blocks would not share a common row nor a common column.

Assuming that the two non-conforming blocks are uniformly distributed over the square array, the probability of having to compute four additional hashes is

$$\frac{2\binom{n}{2}^2}{\binom{n^2}{2}} = \frac{n^2(n-1)^2}{n^2(n^2-1)} = \frac{n-1}{n+1} = 1 - \frac{2}{n+1}$$

and the average number of hashes required to identify two non-conforming blocks is

$$2n + 4 \left(1 - \frac{2}{n+1}\right) \approx 2n + 4 = 2\sqrt{N} + 4$$

If there are three non-conforming blocks, we have to take into consideration the four distinct failure patterns displayed in Fig. 9. We can distinguish them by noting how many rows and columns contain non-conforming blocks. The possibilities are Case 1: $1 \times 3$ and $3 \times 1$, Case 2: $2 \times 2$, Case 3: $2 \times 3$ and $3 \times 2$, and Case 4: $3 \times 3$.

**Case 1:** There are twice $n \binom{n}{3}$ possibilities to select a single row and three different columns or a single column and three different rows. This selection not only determines the non-conforming blocks, but also their number. No additional hash computations are required as we can infer the identities of the three non-conforming blocks from their row and column indices.

**Case 2:** There are $\binom{n}{2}^2$ possibilities to select two rows and two columns in the grid. There are then four possibilities to select the three non-conforming blocks at the four intersection points of these rows and columns for a total of $4 \binom{n}{2}^2$ possibilities. We also need to ascertain the four object hashes to diagnose as we do not know *a priori* that we only have three non-conforming blocks.

**Case 3:** There are $\binom{n}{2}\binom{n}{3}$ ways to select two rows and three columns for the non-conforming blocks and the same number to select three rows and two columns. In both cases, there are then six possibilities to select three non-conforming blocks such that each row / column selected has at least one non-conforming block. The total number of patterns is therefore $2 \cdot 6 \cdot \binom{n}{2}\binom{n}{3}$. All object hashes at the six intersection points need to be compared.

**Case 4:** There are $\binom{n}{3}\binom{n}{3}$ ways to select three rows and three columns for the non-conforming blocks. There are six possibilities to place the three non-conforming blocks in the nine intersections such that all non-conforming blocks are in three different rows and three different columns. In this case, we need nine additional hashes in order to ascertain number and locations of non-conforming objects as the algorithm has *no a priori* knowledge of the number of non-conforming objects.

Taking everything together, the expected number of additional hashes is

$$\frac{8 + 9(n-2)n}{n^2-2},$$

which converges slowly to 9. The number of total hashes is

$$2n + \frac{8 + 9(n-2)n}{n^2-2}.$$
Fig. 10. Expected number of hash accesses needed to identify one, two or three non-conforming objects in a Merkle Tree or a Basic Merkle Grid with $2^m$ objects.

Fig. 11. Number of hashes to be accessed for simple Merkle grids in the presence of burst errors of length $l$.

Fig. 12. Number of hashes to be accessed for simple Merkle grids counting 64, 256 or 1,024 objects in the presence of burst errors of length $l$.

number of hashes to be accessed for simple Merkle grids in the presence of burst errors is again dominated by the need to access all row and column hashes. This eliminates the potential of having to access more than the needed block hashes if the burst stretches over two rows. As can be seen from the Figure, the curves lie virtually on top of each other for all bursts regardless of their length. This is shown more clearly in Fig. 12, which focuses on small grids.

B. Extended Merkle grids

As we have seen, Merkle trees can authenticate a single block among N blocks by providing $\log_2(N)$ hashes and potentially verifying one signature. Merkle grids need the sender to send the $\sqrt{N}$ hashes of all column hashes and the $\sqrt{N}$ hashes of the blocks in the same column as the one to be authenticated.

We depict the situation for an extended Merkle grid in Fig. 13. We want to verify the block in red. Both sides calculate its hash. We then send the hashes of the blocks in the same column to the receiver. Alternatively, we could have chosen the row instead of the column. The receiver then calculates the column hash (indicated by the red-white double striping of the square representing the hash) from the hash of the block at its side and the witness hashes just sent. The sender sends a companion hash, i.e. the column hash of the column next to the column with the block to be authenticated, of the lowest leaf layer of the column Merkle tree (indicated in violet) as a witness. The receiver now calculates the hash of the concatenation of both and starts this way working up the tree. At each step, there is a just calculated hash and a companion hash sent by the sender. When the top of the Merkle tree is reached, the hash is compared to the corresponding part of the signed concatenation of two root hashes. This finishes authentication. We have $n$ object hashes and $\log_2(n)$ hashes of hashes to calculate. We also need $\log_2(n)$ witness hashes to be sent possibly in a single message from sender to receiver.

We now calculate the cost of an Extended Merkle Grid with $2^m \times 2^m = 2^{2m}$ objects. If only a single block is non-conforming, then the row and column Merkle trees will need twice $2m$ additional hashes to isolate the non-conforming row and column hashes, which suffice to uniquely identify the non-conforming block uniquely. This gives a total of $4m$ additional hashes. Recall that we always assume that the root hash is shown to be not compliant.
For two non-compliant blocks, we use the same four case distinctions as before. In Case 1, the two non-conforming objects share a row or a column, and no additional hashes from the grid are required. The non-conformance requires accessing $4m$ hashes in one Merkle tree and

$$a_m = 2 \left( \frac{1}{2^m - 1} + 2 \right) m - 4$$

hashes in the other. In Case 2, four hashes in the grid are needed as well as $2a_m$ in the two trees. In total, we have

$$- 4 + 4 \left( \frac{1}{2^m - 1} + 4m \right) m$$

accesses by taking the expected number.

For three non-compliant blocks, we again use the same case distinctions as before, see Fig. 9. In Case 1 ($3 \times 1$ or $1 \times 3$), no additional hashes in the grid itself are needed, and $2m$ hashes in one tree and $\tau_m$ in the other tree. In Case 2 ($2 \times 2$), four hashes from the grid itself and $2a_m$ in the trees are needed. In Case 3 ($2 \times 3$ or $3 \times 2$), 6 additional hashes in the grid itself and $a_m$ in the trees are required. Finally, in Case 4 ($3 \times 3$), we need $2\tau_m$ accesses in the trees and nine in the grid itself. We already calculated the probabilities for these cases before assuming that non-conformity is equally likely for all objects. The resulting expectation simplifies into

$$- 36 + 9 \cdot 2^{m+3} + 4m(72 - 29m) + 24m + 16m(-29 + 36m)$$

$$\frac{3(2 - 4^m + 16m)}{5}$$

This value is virtually indistinguishable from that for the binary Merkle tree.

Finally, we consider the effects of a burst error affecting $l$ consecutive objects. In the basic Merkle grid, this cost is dominated by accessing all line hashes, whereas in the Merkle tree, it is almost as efficient as finding a single non-conforming object, see Fig. 9. An extended Merkle grid has almost the same performance as a Merkle tree. We obtained our results, given in Fig. 14, through a program that enumerated all possible locations of $l$ bursts and collected the mean number of hashes accessed. The numbers are slightly worse than for the Merkle tree.

C. Our findings

Looking at Fig. 14, we can see that the performance of the Merkle tree and the extended Merkle scheme are almost identical. For the basic scheme, we have better or almost identical performance for small sizes. Since we can assume that the probability of encountering non-conforming blocks is very low, that fact is of little practical interest.

An important consideration is the cost of authenticating all blocks. To perform this task, a Merkle tree must verify all its elements, which requires $2^{N+1} - 1$ steps. A basic Merkle grid would just have to authenticate its master hash and compute $2\sqrt{N}$ hashes. An extended Merkle grid would just have to verify its master hash and its two auxiliary trees, which would require $4\sqrt{N} - 1$ hashes. The savings can be quite considerable for large files or large collections of messages. Consider for instance a file consisting of 1,024 blocks. A Merkle tree authentication would require 2,047 hashes while a basic Merkle grid and an extended Merkle grid would respectively require 65 and 128 hashes.

Our findings are summarized in Table 1. They highlight the excellent performance of extended Merkle grids that take slightly more space than basic Merkle grids—and much less space than Merkle trees, but still perform as well or better than Merkle trees in all the cases we investigated.

V. Related Work

Originally used for a public key signature scheme [2] [10–13], Merkle trees have been used in a wide array of settings. We only present here some of them. He, Xu, Fu, and Zhou [4] as well as Lin and Sung [8] use Merkle trees for authenticating the sender of multicast messages. In multicasting, the sender will not automatically retransmit lost packets and the communication between receiver and sender must be minimized. At the same time, the cost of authenticating messages should be amortized over a large number of messages. Both sets of authors propose to group the messages into the leaves of a Merkle tree and judiciously append partial matches to spread calculations over the processing of each message. As is usual, only the root hash is signed by the sender and authenticated by receivers. Ferrag and his colleagues [7] mention various applications of Merkle trees for authentication protocols for the Internet of Things.

File systems use checkpointing extensively [15] and often use checksums [5] to implement them. For instance, Open Solaris’s ZFS file system uses checksums for detecting data corruption. ZFS organizes all on-disk data and metadata into objects that are further grouped into object sets. Checksums for on-disk blocks are kept separate by storing them into a parent block. They form part of a Merkle tree. In this manner, ZFS can detect all types of silent data corruption [16].
Another problem area where Merkle hash trees are used is the authentication of outsourced data, especially that of a database. Li and colleagues propose to embed Merkle trees in the nodes of a B+ tree [6] Niaz and Saake [14] propose to replace the binary structure of Merkle trees with a B+ tree. In this context, Martel and colleagues [9] propose to replace binary Merkle trees with generalized directed acyclic graphs (DAGs) in the context of authenticating general data structures. The owner of a data structure provides first a deterministic search procedure that takes a query, searches the data structure, and returns the correct answer. The procedure is modeled as accessing nodes in a DAG. Each leaf node contains the hash of the data found there, and all interior nodes contain a hash of a concatenation of the hashes of their child nodes.

Auvolat and Traīni [1] have introduced Merkle Search Trees and proposed to use them to build causally consistent event stores that can provide causally consistent eventual delivery of updates to all connected nodes of a distributed system. Dahlberg, Pulls, and Peeters [3] investigated sparse Merkle trees and presented the first complete, succinct, and recursive definitions of sparse Merkle trees and related operations.

VI. CONCLUSION

We have presented a new data organization that replicates the functionality of Merkle trees while reducing their transmission and storage costs by up to 50 percent. All Merkle grids organize the objects whose conformity they monitor in a square array. They supplement this array with:

1. Row hashes that contain the hash of the concatenation of the hashes of all the objects in their respective row
2. Column hashes that contain the hash of the concatenation of the hashes of all the objects in their respective column.
3. A single signed master block that contains the hash of the concatenation of all row and column hashes.

Extended Merkle grids add to this basic scheme two auxiliary Merkle trees to speed up searches among both row hashes and column hashes.

While both basic and extended Merkle grids perform authentication of all blocks better than Merkle trees, only extended Merkle grids can locate individual non-conforming objects or authenticate a single non-conforming object as fast as Merkle trees. In addition, both basic and extended Merkle grid can authenticate $N$ objects much faster than Merkle trees, as the number of steps they take is proportional to $\sqrt{N}$ instead of being proportional to $N$.

REFERENCES

\textbf{μCache: a mutable cache for SMR translation layer}

Mohammad Hossein Hajkazemi\*, Mania Abdi\†, Peter Desnoyers\‡

Department of Electrical and Computer Engineering\*, Khoury College of Computer Sciences\†
Northeastern University
hajkazemi@ece.neu.edu, abdi.ma@husky.neu.edu, pjd@ccs.neu.edu

\textbf{Abstract}—Shingled Magnetic Recording (SMR) may be combined with conventional (re-writable) recording on the same drive; in host-managed drives shipping today this capability is used to provide a small number of re-writable zones, typically totaling a few tens of GB. Although these re-writable zones are widely used by SMR-aware applications, the literature to date has ignored them and focused on fully-shingled devices. We describe μCache, an SMR translation layer (STL) using re-writable (mutable) zones to take advantage of both workload spatial and temporal locality to reduce the garbage collection overhead resulted from out-of-place writes. In μCache the volume LBA space is divided into fixed-sized buckets and, on write access, the corresponding bucket is copied (promoted) to the re-writable zones, allowing subsequent writes to the same bucket to be served in-place resulting in fewer garbage collection cycles.

We evaluate μCache in simulation against real-world traces and show that with appropriate parameters it is able to hold the entire write working set of most workloads in re-writable storage, virtually eliminating garbage collection overhead. We also emulate μCache by replaying its translated traces against actual drive and show that 1) it outperforms its examined counterpart, an E-region based translation approach on average by 2x and up to 5.1x, and 2) it incurs additional latency only for a small fraction of write operations, (up to 10\%) when compared with conventional non-shingled disks.

\textbf{Index Terms}—shingled magnetic recording, translation layer

I. INTRODUCTION

Shingled Magnetic Recording (SMR) is a modern technology that offers higher storage density compared to conventional magnetic recording (CMR) with the same head and platter technology. It achieves this by overlapping tracks as they are written, for an effective track width narrower than the write head. However, this density improvement comes at a cost: individual disk sectors cannot be over-written, as adjacent downstream tracks will be corrupted [1].

If all tracks on an SMR disk were shingled it would be a purely “write-once” media: once the last track was written, no re-use would be possible without damage to non-overwritten sectors. Instead, the disk is divided into zones, separated by “guard tracks” wide enough to prevent adjacent track corruption; each zone may be sequentially written (and rewritten) from the beginning without damage to data in other zones. This approach has been formalized in the SATA and SCSI extensions [2], [3] for zoned block devices, a storage model much like NAND flash: large regions (256 MiB for SMR) must be written sequentially, and the operation to allow a region to be rewritten—“reset zone pointer” for SMR, erase for flash—discards all data in that region.

The SMR restrictions may be addressed in the application or file system [4], using host-managed devices which expose SMR write restrictions and provide commands to clear zones for re-write. Alternately, existing file systems may be used over a block translation layer [5], implemented either in the host of a host-managed or in the firmware of a drive-managed SMR device. (a third standardized device type, host-aware [6], is a hybrid with features of both host-managed and drive-managed.) We focus on block translation layers in this work, exploring algorithms which may be implemented in either the host, via host-managed extensions, or the firmware of drive-managed devices.

Most SMR translation layers (STLs) [1], [7], [8]) are “E-Region-based” [9] translation layers, using a 1:1 mapping from logical block addresses to “data zone” locations, and a small region (the E-region or “on-media cache”) to cache exceptions caused by writes, which are written in log-structured fashion to the cache “write frontier”. If this cache is shingled the resulting algorithm is similar to the FAST flash translation layer (FTL) [10], with fixed locations for data zones as no wear-leveling is needed. As in the FAST FTL, when the on-media cache fills, the expensive garbage collection (GC) operation is performed to make room; space must be reclaimed at a zone at once, merging valid pages from that zone with unmodified pages from the corresponding data zone, and then writing the result back to a data zone.

A fully-shingled host-managed drive presents practical complications. For example, the first sectors of a disk (i.e. LBA 0) are typically used to store partition and file system metadata [11], [12], however, they cannot be updated safely if they are in a shingled zone, but must be “erased” and then rewritten. Yet tracks on an SMR drive do not all necessarily need to be shingled, as modern drives are able to vary the track pitch at formatting time [13], and it is possible to format a drive with a combination of shingled tracks and wider re-writable tracks. Thus host-managed drives available today provide a number of re-writable (mutable) zones, starting at LBA zero and totaling 16 to 30 GiB in the drives we have examined.

These data zones are used for hot data storage in both open-source translation layers (i.e. dm-zoned [14]) for host-managed SMR and (according to anecdotal statements) in proprietary translation layers in recent drive-managed devices. Yet to date SMR translation layers reported in the literature have either assumed a single fully-shingled disk [1], [5], [7], [15]–[17], or a costly hybrid system comprising e.g. SSD and SMR [8], [18] with no seek overhead for accessing re-writable storage.

In this paper, we introduce and analyze μCache, a translation layer for SMR disk that relies on both shingled and re-writable
zones to operate. Similar to E-region based approaches offered in [5], [16], [17], μCache divides the disk into a large data zone and a small on-media cache to address random writes. However, the on-media cache is formed by re-writable zones. Relying on workload spatial and temporal locality, μCache accommodates the largest possible write working set in the on-media cache. This results in improved performance compared to translation layers using only shingled zones as the cached data may be overwritten. To achieve this, on write access, μCache copies a chunk of data (typically larger than the original I/O size) to the on-media cache (if it is already not residing in the cache). Therefore, subsequent write accesses to the same chunk will overwrite this location in cache, rather than consuming new on-media cache space, and thus resulting in fewer multi-second garbage collection cycles.

In summary, the main contributions of this work are as follows:

- We describe μCache, an SMR translation layer using re-writable zones for the on-media cache;
- We show that by exploiting both temporal and spatial locality, μCache is capable of accommodating large write working sets in re-writable cache and therefore minimizes the garbage collection overhead;
- We simulate μCache with real-world traces [19], demonstrating the reduction in garbage collection overhead;
- We emulate μCache by replaying its translated traces against actual drive and show that for many of the workloads it makes an SMR drive nearly as performant as a conventional drive, imposing a modest burden of additional seek and copies. We further show that an average 2x performance improvement of μCache compared to its counterpart E-region based translation layer.

II. MOTIVATION

In both E-region-based translation layers and their alternative (dm-zoned), as soon as the cache fills, data is evicted from cache region via expensive garbage collection (GC) cycles. Thus, the performance of these translation layers are strongly affected by the size of the write working set—i.e. the number of unique locations written during some time window—and whether it fits in that cache. If the write working set is larger than the cache, a large fraction of writes will result in evictions, each incurring multi-second penalties due to the need for multiple reads and writes of 256 MiB zones. Conversely, if the working set is significantly smaller than the effective cache capacity, the cache would be utilized more highly and thus few or no GC cycles will be needed.

With a shingled on-media cache, however, the effective cache capacity may be significantly smaller than the space it occupies, with space used by outdated (invalid) data. In Figure 1 we see effective cache utilization for several workloads running on a simulation of an E-region STL from [9] with a 16 GiB shingled on-media cache, equivalent to that of a Seagate ST8000AS022 8TB 5900 RPM host-aware drive.

As seen, only a small fraction of the cache is used at any time, greatly restricting the write working set which can be held in cache. We see that the high effective cache capacity reaches 50% (for w87), with a mean utilization of 37%, while mean utilization is much lower for the other two: 9% (w78) and 12% (w37). The traces are taken from the CloudPhysics corpus [19] of virtual machine block traces, and are described in more detail later in Section IV.

Workload temporal and spatial localities are two parameters that determine the working set size and therefore caching performance. Temporal locality refers to the tendency of accesses to the same address to cluster in time. In other words, immediately after seeing a reference to address A, the expected time until the next reference to A is lower than the expected time until the next reference to an arbitrarily address. We see this behavior for write operations in Figure 2, showing CDFs of 4 MiB-granularity stack distance [20], i.e. the number of unique 4 MiB buckets touched between accesses to the same bucket) for w87, w78 and w37.

As seen, only a small fraction of the cache is used at any time, greatly restricting the write working set which can be held in cache. We see that the high effective cache capacity reaches 50% (for w87), with a mean utilization of 37%, while mean utilization is much lower for the other two: 9% (w78) and 12% (w37). The traces are taken from the CloudPhysics corpus [19] of virtual machine block traces, and are described in more detail later in Section IV.

Workload temporal and spatial localities are two parameters that determine the working set size and therefore caching performance. Temporal locality refers to the tendency of accesses to the same address to cluster in time. In other words, immediately after seeing a reference to address A, the expected time until the next reference to A is lower than the expected time until the next reference to an arbitrarily address. We see this behavior for write operations in Figure 2, showing CDFs of 4 MiB-granularity stack distance [20], i.e. the number of unique 4 MiB buckets touched between accesses to the same bucket, for the same three workloads. As can be seen, among all pair of 4 MiB consecutive write accesses, about 47%, 52% and 78% of the following accesses are to the same 4 MiB region in workloads w37, w78, and w87 showing a high temporal locality for write operations among these traces.

Spatial locality refers to the tendency of references to cluster within the address space; i.e. address A±ε is much more likely to be accessed soon after A than some address far from A. Our analysis shows high write spatial locality in most of the traces. For example, for w87 and w78 more than 75% and 33% of the second write accesses are within the range of less than 256 KiB of the preceding write, showing high spatial locality.

These two measures affect different aspects of cache design. Temporal locality determines LRU cache performance, as
accesses of stack distance $D$ or less will hit in a cache holding $D$ entries. Spatial locality motivates the use of cache lines larger than a single access, on average satisfying requests to more than a single location while needing only one operation to backing storage. For cache line sizes small enough that the fixed cost access dominates, increasing line sizes will (all other things being equal) result in higher performance. Other things are not always equal, however, a larger line sizes result in lower effective cache utilization, with space being used for data that is never accessed.

III. $\mu$CACHE

We describe $\mu$Cache, an SMR translation layer relying on re-writable zones to more efficiently utilize the on-media cache and thus improve the performance. Taking advantage of on-media cache re-write-ability, $\mu$Cache exploits both high temporal and spatial locality to perform writes in-place and therefore minimize expensive GC cycles.

A. $\mu$Cache algorithm

In Figure 3, we see the high-level $\mu$Cache data layout: the re-writable region is used for checkpoints and on-media cache, while the shingled region holds a temporary zone and the permanent data zones (checkpointing and temporary copies are described more fully in Section III-B). The LBA space of the volume is divided into fixed-sized buckets, and on write access a bucket is promoted and copied to the on-media cache, reading any necessary data (i.e. that not contained in the triggering write operation) from the data zone. Subsequent writes to that bucket will be performed in-place in the on-media cache until it is evicted by GC. The GC process selects a data zone, reads the data zone and corresponding cached data, merges them, and writes them back.

Details of the $\mu$Cache translation strategy are described as below. Given a disk with $N$ shingled and $M$ re-writable zones of size $Z$ ($M < N$), and buckets of size $b$:

- we divide the LBA space into $N \cdot Z / b$ logical buckets
- we divide the $N$ shingled zones into the same number of buckets, the home locations for each logical bucket

Given a write access to address $A$, we:

- determine its logical bucket number $Lbn = \left\lfloor \frac{A}{b} \right\rfloor$
- if $Lbn$ is cached in some physical cache bucket $Pcb$, perform the write to offset $A \mod b$ in bucket $Pcb$; otherwise:
  - allocate a physical cache bucket $Pcb$, promote the bucket to cache location $Pcb$, and again perform the write at the appropriate offset.

There are two options for bucket promotion on new writes: copying and mapping. Copying reads the contents of the bucket from the data zone, merging it with the new write, and writes the entire bucket in cache. Mapping allocates a bucket in cache but does not read from the data zone; a bitmap is instead used to track which portions of the logical bucket are in cache and which in the data zone. Mapping eliminates a seek and two bucket transfers during promotion, but incurs overhead to persist mapping information for every write, as well as read seeks due to fragmentation between cache and data regions.

We argue below that copying is the better approach, due to its exploitation of spatial locality and potential to eliminate map persistence overhead, and focus on this option in our evaluation.

Given a read access, we:

- select a data zone $D$ and read its contents
- read all buckets from cache holding data from zone $D$
- merge contents of data zone and data from cache
- save a copy (to prevent data loss in next step)
- write back to the data zone

We note that $\mu$Cache is in fact a generalization of several existing translation approaches. With minimum-sized buckets, only incoming writes are sent to cache and the two promotion behaviors are equivalent; the behavior is that of an E-region translation layer with re-writable on-media cache, allowing full cache utilization and more efficient GC. At the other end of the range, dm-zoned [14] is equivalent to $\mu$Cache with a bucket size of one zone and bucket promotion by mapping.

Can promotion-by-copy with a modest bucket size outperform both extremes, avoiding low utilization and wasted cache space due to promoting entire zones, while taking advantage of spatial locality? The answer is yes; with a modest bucket size, almost the entire working set could fit in the on-media cache resulting in high cache utilization (see Table III). Support for this approach maybe seen in Figure 4, showing the write footprint of several workloads when accesses are rounded up to varying bucket sizes.

As seen, footprint increases with bucket size for all workloads; however the increase is gradual for some, and rapid for others. Moreover, with a reasonable cache size (e.g. 32 GiB) combined with a modest bucket size (e.g. 256 KiB) the entire working set of many of the workloads fits in the cache.

B. Implementation factors

A number of implementation factors that are important to the value of $\mu$Cache are listed below, but some are not addressed in our evaluation:

![Fig. 3: $\mu$Cache On-disk data structure: checkpoint zone, re-writable cache zones, temporary zone and data zones.](image)
Bucket allocation: \( \mu \text{Cache} \) uses a simple arbitrary bucket allocation policy: it keeps a pool of free physical cache buckets (\( Pcb \)), and assigns them arbitrarily to \( Lbn \) on promotion operations.

Memory usage: Memory usage is a significant issue for drive-resident translation layers, and even host-resident ones are limited in the resources they may demand. (e.g. when deployed on specialized storage servers housing as many as 60 or more drives) \( \mu \text{Cache} \) only needs to map as many buckets as are held in the on-media cache, keeping its memory demands modest. (e.g. with a very small bucket size of 128 KiB, a 16 GiB on-media cache would require a map of 128 K entries, taking less than 10 MiB of memory if implemented with one of several sparse map data structures.)

Map persistence and checkpointing: Translation layers using out-of-place writes must reliably record map updates, as a write is not truly durable until the information needed to locate the new data has been persisted safely in a way which will survive e.g. power failure. In our evaluation we do not implement map persistence; however we note that copy-based promotion eliminates map updates for overwrites, and at bucket promotion they may be logged (as is done in FSTL [5]) in a bucket header, eliminating additional seeks to persist them at a fixed location. Figure 5 shows an example where \( \mu \text{Cache} \) performs logging when \( Pcb_{n+2}, Pcb_{n+4}, Pcb_n \) and \( Pcb_{n+3} \) are allocated to \( Lbn \) 1 to 4.

As shown, besides the \( Lbn \) to which a bucket is allocated, a header contains a sequence number and a CRC to identify whether a particular bucket write completed before a crash. It also comprises a pointer to the next available \( Pcb \) header. Pointers are used to reconstruct the map in case of a crash. Depending on where \( \mu \text{Cache} \) is implemented (either in the firmware or on the host side), a header size will be a sector (512 B) or 4 KiB (to preserve 4 KB alignment) adding a negligible 2.5-5 µs or 20-40 µs of transfer time to each promotion (bucket write). The resulting space overhead is also negligible: assuming a small bucket size of 256 KiB, the header overhead will be around 0.2% or 0.8% which we consider quite acceptable.

Techniques such as periodic checkpointing may also be used to make the recovery process faster in case of any failure. To do so, the most recent map along with a latest \( Pcb \) number is stored at the checkpoint zone. On recovery, first, the most recent map is retrieved. Second, by traversing the header pointers starting at the latest checkpointed \( Pcb \) number, map reconstruction is completed.

Compared to copy-based promotion, map-based promotion requires map information to be persisted (at the cost of a seek) for each write to the bucket after promotion. This overhead may be mitigated by deferring writes until the receipt of a write cache FLUSH operation, as is done in \textit{dm-zoned}; however these are very frequent in some workloads.

Garbage collection: The garbage collection process itself in \( \mu \text{Cache} \) is almost identical to \( \text{E-region} \) translation layer cleaning (GC) described in Skylight [1] requiring at least 3 full zone transfers: select a data zone to clean, read all data from that zone residing in cache, read the zone itself, merge them and write a backup copy to the temporary zone, and overwrite the original data zone. Selection of the zone to clean is more complex, however, as e.g. maximizing the space freed might in fact evict hot data. In this work we omit discussion of zone selection for GC, and focus on sizing buckets to eliminate the need for GC.

While the cleaning process could affect the throughput significantly, it may not impact the I/O latency necessarily. A garbage collection cycle takes roughly 5 seconds on average, or even more at inner-track LBAs or if reading many extents from cache. The worst-case I/O latency, in turn, depends greatly on how well host I/Os can be interleaved with the operations making up these cycles. We believe that this is primarily an engineering issue, not an attribute of a particular translation layer, and thus in our evaluation we focus on the number of GC cycles incurred and resulting loss in throughput, ignoring the effect on latency.

IV. Trace-driven Evaluation

We evaluate \( \mu \text{Cache} \) using real-world traces, using simulators implementing the \( \mu \text{Cache} \) algorithm and the \( \text{E-region} \) approach, a comparison translation layer. We measure \( \mu \text{Cache} \) performance by replaying its translated trace (generated by \( \mu \text{Cache} \) simulator) against a physical drive and compare it with that of the \( \text{E-region} \) approach as well as a conventional drive; each experiment is run at least five times. This workflow is seen in Figure 6.

Workloads: \( \mu \text{Cache} \) was evaluated using the CloudPhysics block trace corpus [19]. 106 large block traces from a virtualized environment running Windows and Linux with modern file systems. As shown by Hajkazemi et al. [21] these traces are more representative of modern workloads than older traces such as the well-known MSR corpus [22]. A subset of the 106 traces were selected, choosing ones which (1) were long enough to trigger garbage collection, and (2) represented

![Fig. 4: Write footprint of several workloads vs. bucket size.](image)

![Fig. 5: \( \mu \text{Cache} \) logging mechanism: buckets may be written in arbitrary order, but are linked through their headers in the order they are written to allow log recovery on failure.](image)
different levels of read/write intensity. The selected workloads vary in size from about 4 to 50 million I/Os, and range from read-heavy (w21, 10% writes) to very write-heavy (w87, 80% writes); details are shown in Table I.

**Trace-driven experiment**: Trace-driven simulation and emulation of both μCache and the E-region STL [9] were used to measure behavioral statistics as well as performance such as I/O amplification, on-media cache hit ratio, additional seeks incurred, garbage collection cycles, promotion operation latency and throughput.

As shown in Figure 6, μCache was emulated by replaying its translated traces (generated by μCache simulator) against the actual device to measure latency and throughput. These traces combined remapped host I/Os, bucket promotion reads and writes for μCache, and garbage collection operations for both μCache and E-region STL. Promotion and GC I/Os were split into 512 KiB operations, a common limit for SCSI reads and writes. Trace replay was performed by fio, an I/O testing tool [23], using libaio I/O engine with iodepth of 31. A summary of experimental parameters may be seen in Table II.

**A. μCache performance**

As described in section III, the μCache goal is to hold the largest possible write working set in the on-media cache to maximize the number of in-place writes, and thus reduce garbage collection cycles. To measure its success, we report 1) the μCache throughput with bucket size of 256 KiB compared to that of both E-region STL and a conventional drive (CMR) in Figure 7 and, 2) the cache hit ratio for write accesses with different bucket sizes ranging from 0.125 MiB to 4 MiB in Table III.

As seen in Figure 7, μCache performance is on average twice as high as that of the E-region STL, and in the best case (w75) μCache outperforms E-region STL by 5x. For two of the workloads (i.e. w69 and w48) μCache performance is marginally lower, however, we note that by selecting a slightly larger bucket size, (e.g, 512 KiB for w69) μCache is able to beat the E-region STL in these traces.

As seen, CMR performance is higher than that of μCache for all workloads, however, this performance gap is marginal for several workloads (e.g., w91, w87) indicating the low overhead and high performance of μCache. Note that the large performance gap between both translation approaches and the CMR for w46 is due to the large footprint of this trace (see Figure 4). This large footprint results in a working set larger than the cache size, and therefore leads to more cache write misses (see Table III), more GC cycles (see Table IV), and consequently lower performance.

As seen in Table III, all but two traces (w10 and w46) show hit ratios over 85% at all bucket sizes, and nine out of ten traces reach a hit ratio of 99% with the proper bucket size. Even the cache-unfriendly w46 trace (54% hit ratio at 128 KiB) improves to 93% when the bucket size is increased to 4 MiB.

We see three different patterns in hit ratio vs bucket size: 1) for the majority of traces (w21, w26, w28, w46, w48, w69, w87, w91) the hit ratio increases monotonically as bucket size is increased; 2) for three of the traces, w09, w10, and w14 the hit ratio first goes up and then starts decreasing at a certain bucket size (2 MiB, 0.5 MiB, and 0.5 MiB for w09, w10 and w14 respectively); 3) for a single trace, w37, the hit ratio decreases with increasing bucket size.

We note that by increasing the bucket size we exploit the spatial locality of the traces more effectively resulting in a higher on-media cache hit ratio for the traces in case 1. However, if a workload has poor spatial locality, this increase in bucket size reduces the effective cache capacity (by caching unneeded data), in turn resulting in more capacity misses and an overall lower on-media cache hit ratio in cases 2 and 3.

**B. μCache garbage collection**

Garbage collection (GC) is the primary factor reducing throughput for shingled drives [7], as it is a multi-second

---

**TABLE I: Statistical summary of selected workloads.**

<table>
<thead>
<tr>
<th>workload</th>
<th>w09</th>
<th>w10</th>
<th>w14</th>
<th>w21</th>
<th>w26</th>
<th>w28</th>
<th>w29</th>
<th>w37</th>
<th>w46</th>
<th>w48</th>
<th>w69</th>
<th>w75</th>
<th>w78</th>
<th>w87</th>
<th>w91</th>
</tr>
</thead>
<tbody>
<tr>
<td>I/O count (M)</td>
<td>49.62</td>
<td>48.34</td>
<td>34.81</td>
<td>29.41</td>
<td>26.53</td>
<td>19.73</td>
<td>19.08</td>
<td>18.84</td>
<td>11.54</td>
<td>14.09</td>
<td>7.87</td>
<td>6.15</td>
<td>5.49</td>
<td>3.78</td>
<td>4.31</td>
</tr>
<tr>
<td>write ratio</td>
<td>0.55</td>
<td>0.39</td>
<td>0.68</td>
<td>0.1</td>
<td>0.58</td>
<td>0.33</td>
<td>0.14</td>
<td>0.21</td>
<td>0.62</td>
<td>0.42</td>
<td>0.3</td>
<td>0.79</td>
<td>0.55</td>
<td>0.8</td>
<td>0.27</td>
</tr>
<tr>
<td>read mean size (KB)</td>
<td>33.4</td>
<td>26.7</td>
<td>32</td>
<td>21.9</td>
<td>19.1</td>
<td>45.2</td>
<td>14.6</td>
<td>7.1</td>
<td>212</td>
<td>6</td>
<td>17.4</td>
<td>28.5</td>
<td>20.4</td>
<td>18.6</td>
<td>15.4</td>
</tr>
<tr>
<td>peak IOPS (every 10 sec)</td>
<td>941</td>
<td>2002</td>
<td>681</td>
<td>1321</td>
<td>1063</td>
<td>4489</td>
<td>2327</td>
<td>1414</td>
<td>2560</td>
<td>4520</td>
<td>760</td>
<td>521</td>
<td>1841</td>
<td>3378</td>
<td>3223</td>
</tr>
</tbody>
</table>

**TABLE II: Experimental parameters and drive specification.**

<table>
<thead>
<tr>
<th>μCache and baseline E-region STL simulation parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>on-media cache size &amp; zone size</td>
</tr>
<tr>
<td>max and min I/O size</td>
</tr>
<tr>
<td>μCache &amp; E-region eviction policy</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Physical drive specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>drive model</td>
</tr>
<tr>
<td>drive capacity &amp; rpm</td>
</tr>
<tr>
<td>Wcache &amp; read-ahead &amp; look-ahead</td>
</tr>
</tbody>
</table>
TABLE III: Write access hit ratio in \( m \)Cache with different bucket sizes captured in simulations.

<table>
<thead>
<tr>
<th>workload</th>
<th>bucket size</th>
<th>w09</th>
<th>w10</th>
<th>w14</th>
<th>w21</th>
<th>w26</th>
<th>w28</th>
<th>w29</th>
<th>w37</th>
<th>w46</th>
<th>w48</th>
<th>w69</th>
<th>w75</th>
<th>w78</th>
<th>w87</th>
<th>w91</th>
</tr>
</thead>
<tbody>
<tr>
<td>256 KiB</td>
<td>0.91</td>
<td>0.76</td>
<td>0.97</td>
<td>0.99</td>
<td>0.95</td>
<td>0.86</td>
<td>0.96</td>
<td>0.54</td>
<td>0.79</td>
<td>0.98</td>
<td>0.99</td>
<td>0.98</td>
<td>0.97</td>
<td>0.98</td>
<td>0.98</td>
<td></td>
</tr>
<tr>
<td>512 KiB</td>
<td>0.92</td>
<td>0.77</td>
<td>0.91</td>
<td>0.98</td>
<td>0.99</td>
<td>0.91</td>
<td>0.95</td>
<td>0.76</td>
<td>0.99</td>
<td>0.99</td>
<td>1</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 MiB</td>
<td>0.92</td>
<td>0.75</td>
<td>0.91</td>
<td>0.99</td>
<td>0.99</td>
<td>0.98</td>
<td>0.94</td>
<td>0.84</td>
<td>0.99</td>
<td>0.99</td>
<td>1</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 MiB</td>
<td>0.90</td>
<td>0.73</td>
<td>0.90</td>
<td>0.98</td>
<td>1</td>
<td>0.99</td>
<td>0.92</td>
<td>0.92</td>
<td>0.90</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0.99</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>4 MiB</td>
<td>0.88</td>
<td>0.70</td>
<td>0.87</td>
<td>0.98</td>
<td>1</td>
<td>1</td>
<td>0.91</td>
<td>0.92</td>
<td>0.93</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0.99</td>
<td>0.99</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TABLE IV: GC cycles for \( m \)Cache (bucket sizes 128 KiB to 4 MiB) and E-region shingled cache captured in simulations.

<table>
<thead>
<tr>
<th>workload</th>
<th>bucket size</th>
<th>w09</th>
<th>w10</th>
<th>w14</th>
<th>w21</th>
<th>w26</th>
<th>w28</th>
<th>w29</th>
<th>w37</th>
<th>w46</th>
<th>w48</th>
<th>w69</th>
<th>w75</th>
<th>w78</th>
<th>w87</th>
<th>w91</th>
</tr>
</thead>
<tbody>
<tr>
<td>128 KiB</td>
<td>30462</td>
<td>29753</td>
<td>30293</td>
<td>10</td>
<td>337</td>
<td>573</td>
<td>5238</td>
<td>46</td>
<td>16828</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>256 KiB</td>
<td>43942</td>
<td>51798</td>
<td>43531</td>
<td>96</td>
<td>377</td>
<td>570</td>
<td>8585</td>
<td>323</td>
<td>17592</td>
<td>41</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>512 KiB</td>
<td>76670</td>
<td>96607</td>
<td>71034</td>
<td>463</td>
<td>441</td>
<td>567</td>
<td>15517</td>
<td>1259</td>
<td>18813</td>
<td>102</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 MiB</td>
<td>144459</td>
<td>189933</td>
<td>137555</td>
<td>954</td>
<td>625</td>
<td>577</td>
<td>24335</td>
<td>3693</td>
<td>21059</td>
<td>261</td>
<td>0</td>
<td>0</td>
<td>57</td>
<td>53</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 MiB</td>
<td>328535</td>
<td>377267</td>
<td>300922</td>
<td>2412</td>
<td>843</td>
<td>592</td>
<td>37921</td>
<td>9206</td>
<td>25674</td>
<td>482</td>
<td>0</td>
<td>0</td>
<td>189</td>
<td>110</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 MiB</td>
<td>763952</td>
<td>724891</td>
<td>664792</td>
<td>6670</td>
<td>1483</td>
<td>715</td>
<td>59216</td>
<td>18466</td>
<td>32067</td>
<td>1035</td>
<td>0</td>
<td>0</td>
<td>401</td>
<td>224</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E-region STL</td>
<td>94597</td>
<td>30183</td>
<td>78849</td>
<td>2310</td>
<td>3459</td>
<td>1917</td>
<td>6967</td>
<td>637</td>
<td>29596</td>
<td>724</td>
<td>208</td>
<td>760</td>
<td>630</td>
<td>538</td>
<td>44</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 7: Throughput of \( m \)Cache (with bucket size of 256 KiB) compared to E-region STL and CMR captured in emulations.

operation (roughly 5 seconds at 160 MB/s). Thus we use the number of GC cycles as a proxy to show the impact on throughput.

In table IV we report the number of GC cycles for trace execution with two translation layers: \( m \)Cache with bucket sizes from 0.25 MiB to 4 MiB, and the E-region STL [9]. These experiments use a 16 GiB (64-zone) cache, equal to size of re-writable zones in the Seagate ST8000AS022 8TB 5900 RPM host-aware drive.

In a few cases, no GC cycles were seen for all or many bucket sizes (e.g., w91, w75 and w69); in all other cases smaller bucket sizes resulted in fewer GC cycles. In 10 and 11 out of 15 cases, \( m \)Cache with 1 MiB and 256 KiB bucket sizes outperformed E-region cache. Based on the trend we observe for w09, w10, and w14 where baseline E-region STL outperforms \( m \)Cache, we expect decreasing the bucket size will result in fewer GC cycles in \( m \)Cache over baseline E-region STL.

To examine the impact of cache size on performance, we repeated the simulation with 32 GiB of cache size for both E-region and \( m \)Cache (with bucket size of 256 KiB). Our experiments show an average reduction of 2x in number of GC cycles in both approaches if not eliminated completely.

3More recent drives have been observed to have slightly larger conventional regions, e.g. 31 GiB in devices available to the authors, and 1% of total capacity in other drives.

w91 is the only new case with zero GC cycles for E-region STL, whereas w37, w28, and w21 are the ones for \( m \)Cache.

The actual duration of a GC cycle includes time to read data from the cache, which can be high if gathering large numbers of small fragments. To quantify this possible increase in GC cycle, we measure the number of buckets retrieved from cache in each cycle; results for two representative workloads are seen in Figure 8. In the worst case (w48, 256 KiB bucket size) the mean number of buckets evicted per GC cycle was roughly 200; assuming 5 ms each this would add an additional 1 s to the average GC cycle. We note that the use of buckets bounds this overhead, as well, as the maximum number of 256 KiB fragments in a 256 MiB zone is 1024; under worst-case workloads, simple E-region STL may gather far greater numbers of independent extents in a single GC cycle.

C. \( m \)Cache bucket promotion

Copying an entire bucket for promotion results in an additional seek and bucket transfer, amplifying both I/O operations and bytes transferred. This may be seen in Figures 9a and 9b, which show I/O amplification in operations and bytes, respectively.

We note that although the I/O amplification in bytes is considerable, reaching a factor of 2 and 9 in two (w46 and

4Unlike flash, disk has symmetric read and write performance, thus performance impact is better quantified by I/O amplification than write amplification.
w29, 4 MiB bucket size), the total increase in I/O count is modest in almost all cases. The average byte amplification of all cases with the smallest bucket size is 30%. For capacity drives the overhead of a single seek is roughly the same as for a 2 MB transfer, so the overall impact on performance should be modest, especially for bucket sizes of 1 MiB or less. Although I/O volume amplification increases strictly with bucket size, I/O operation amplification varies by trace; in large part due to increased GC (and thus promotion) with larger bucket sizes.

In addition to transfer time, promotion-by-copy results in an additional disk seek to read the associated bucket from data zones for each promotion. Promotion-by-mapping incurs no promotion seeks, but fragments the bucket between the data zone and the cache, causing extra read seeks. Table V compares the read seek overhead of these two approaches, for bucket sizes of 0.5 MiB and 4 MiB. When promotion-by-copy is used, for a majority of the traces the read seek overhead is seen to be lower than that of promotion-by-mapping for both examined bucket sizes.

In addition to the read seek incurred to promote a bucket, promotion-by-copy introduces fragmentation at bucket boundaries as well; however the number of additional seeks incurred varies depending on the bucket size. Among the traces, w87 and w21 shows the highest and the lowest fragmentation at boundaries (20% and 1% respectively) with the smallest examined bucket size (0.125 MiB). However, increasing the bucket size to 4 MiB reduces the fragmentation ratios to 0.8% and 0.1% for the two workloads.

Bucket promotion is a synchronous operation, i.e. the write operation causing the promotion is not completed till the end of promotion process; this latency varies with bucket size. Figure 10 shows mean latency measured by replaying the µCache translated trace on actual device) incurred by bucket promotion for varying bucket sizes for three representative traces. Not surprisingly, interruption time goes up with bucket size; however the interruptions are modest and of bounded duration; mean interruption time ranges from 30 ms (w75 with bucket size of 128 KiB) to 240 ms (w48, bucket size of 2 MiB). However, these interruptions occur only for cache misses, which as we saw in Table III represent between 1 and 10% of writes in almost all cases. This overhead will be small in comparison to the latency already incurred by writes; Our trace analysis results show that depending on workload between 25% and 70% of writes involve seeks of greater than 256 MiB, incurring significant seek and rotational delays.

V. RELATED WORK

Host-managed SMR drives have shipped with both shingled and re-writable regions for several product generations [6], with a fixed number of re-writable zones starting at LBA zero. Recently announced plans extend this by allowing the size of these regions to be adjusted dynamically [24]; however no guidance is given for use of these regions. ZoneAlloy is among the very first research that proposes approaches to manage such a technology [25].

Re-writable regions for caching are proposed in recent work on track translation layers for interlaced magnetic recording (IMR) [26], where writes to “bottom” tracks may damage adjacent “top” tracks, while top tracks may be modified without such risk. In that work, however, the re-writable zone is used very differently than in our work: only writes to selected hot bottom tracks are forwarded to the cache. Furthermore, in µCache read-modify-write (RMW) operations are performed infrequently, at a zone granularity, whereas in the IMR work [26] RMW operations are performed at a track granularity and more frequently.

In publicly-disclosed SMR algorithms to date, only dm-zoned [14] makes use of re-writable zones; other work either assumes a fully-shingled drive [5], [15]–[17], [27], [28] or a costly hybrid system including SSD [8], [18]. Although dm-zoned represents a step in the direction of this work, we demonstrate the clear advantage offered by (a) smaller bucket sizes, allowing more effective use of the re-writable cache, and (b) promotion by copying, reducing fragmentation and additional read seeks incurred.

Cassuto et. al propose a translation layer using a set-associative cache as well as an additional circular buffer cache. FSTL [5] introduces a framework to design new translation
layers, and use it to explores different garbage collection algorithms for SMR translation layer. Both VGuard [16] and SMaRT [17] propose track-based STL solutions. Tancheff et. al [28] introduce ZDM, a fully page-mapped translation layer implemented as a host-side device mapper for SMR disks, similar to DFTL [29] for NAND flash. In each case μCache differs by taking advantage of conventional regions on the disk, allowing many (and often nearly all) writes to be handled without the overhead of out-of-place writes. Furthermore, with the exception of the track-based SMaRT [17] none of these cases exploit the spatial and temporal locality of workloads, while SMaRT differs in that its caching granularity is fixed to the (location-varying) track size.

Both FC [8] and SMRC [18] use NAND flash for the on-media cache, and thus may outperform μCache, at the cost of adding higher-priced storage to the device or system. μCache relies only on a single magnetic recording device, and thus is simpler and lower-cost.

VI. CONCLUSION

Although a combination of shingled and conventional magnetic recording has been already implemented in host-managed SMR drives, its characteristics when used by a translation layer have not been addressed in the literature yet.

In this work, we introduce μCache, a translation layer that takes advantage of both shingled and re-writable zones on the same device, exploiting both spatial and temporal locality of workloads to reduce the overhead of out-of-place writes.

Simulating μCache against real-world traces, we find that with appropriate bucket sizes the entire write working set of many of workloads can fit in the re-writable cache, resulting in the total elimination of garbage collection overhead. We further, emulate μCache and evaluate its performance by replaying translated traces against actual device and show that it outperforms its counterpart E-region translation layer on average by 2x and up to 5.1x.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1910327 and by a NetApp Faculty Fellowship.

REFERENCES

Symbiotic HW Cache and SW DTLB Prefetching for DRAM/NVM Hybrid Memory

Onkar Patil\textsuperscript{1}, Frank Mueller\textsuperscript{1}, Latchesar Ionkov\textsuperscript{2}, Jason Lee\textsuperscript{2}, Michael Lang\textsuperscript{2}

\textsuperscript{1}North Carolina State University \quad \textsuperscript{2}Los Alamos National Laboratory

opatil@ncsu.edu \quad mueller@cs.ncsu.edu \quad lionkov,jasonlee,mlang@lanl.gov

Abstract—The introduction of NVDIMM memory devices has encouraged the use of DRAM/NVM based hybrid memory systems to increase the memory-per-core ratio in compute nodes and obtain possible energy and cost benefits. However, Non-Volatile Memory (NVM) is slower than DRAM in terms of read/write latency. This difference in performance will adversely affect memory-bound applications. Traditionally, data prefetching at the hardware level has been used to increase the number of cache hits to mitigate performance degradation. However, software (SW) prefetching has not been used effectively to reduce the effects of high memory access latencies. Also, the current cache hierarchy and hardware (HW) prefetching are not optimized for a hybrid memory system.

We hypothesize that HW and SW prefetching can complement each other in placing data in caches and the Data Translation Look-aside Buffer (DTLB) prior to their references, and by doing so adaptively, highly varying access latencies in a DRAM/NVM hybrid memory system are taken into account. This work contributes an adaptive SW prefetch method based on the characterization of read/write/unroll prefetch distances for NVM and DRAM. Prefetch performance is characterized via custom benchmarks based on STREAM2 specifications in a multicore MPI runtime environment and compared to the performance of the standard SW prefetch pass in GCC. Furthermore, the effects of HW prefetching on kernels executing on hybrid memory system are evaluated. Experimental results indicate that SW prefetching targeted to populate the DTLB results in up to 26\% performance improvement when symbiotically used in conjunction with HW prefetching, as opposed to only HW prefetching. Based on our findings, changes to GCC’s prefetch-loop-arrays compiler pass are proposed to take advantage of DTLB prefetching in a hybrid memory system for kernels that are frequently used in HPC applications.

Index Terms—Prefetching, NVDIMM, Optane DC, DRAM, NVM, Hybrid Memory Architecture, HPC, DTLB

I. INTRODUCTION

Hybrid memory architectures are being increasingly adopted in modern computing systems. Intel’s Knights Landing (KNL) introduced High Bandwidth Memory (HBM) along with traditional DRAM-based main memory [1]. General Purpose Graphics Processing Units (GPGPU) are also equipped with HBM [2]. Fujitsu is using a hybrid memory cube (HMC) for its A64FX ARM based chips to deliver high bandwidth memory access to all the compute cores [3]. Recently, Intel launched their Phase Change Memory based Optane DC Persistent Memory Modules (PMM), which are byte-addressable NVDIMMs used as non-volatile main memory (NVM) [4]. The Aurora supercomputer [5], which will be launched in 2021, will have support for Intel Optane DC PMMs. Hence, such a DRAM/NVM based hybrid memory architecture will become more prevalent in future HPC systems. One of the reasons why memory architectures are becoming heterogeneous is because each memory technology brings different characteristics to the table with pros and cons [6]. HBM allows more data to be moved to the processor with the same access latency as DRAM, while NVM provides higher memory capacity supporting application runs with larger problem sizes on fewer systems (nodes), which can result in lower energy consumption and cheaper acquisition costs.

Modern architectures use hardware (HW) prefetchers to increase the amount of cache hits of applications and reduce the effective latency of L1 and L2 cache misses. This can improve overall application performance. HW prefetchers rely on hardware to fetch data by tracking cache line addresses accessed in the N most recently accessed 4KB pages for each core and then predicting future access locations during execution [7], [8]. The HW prefetchers are limited by the 4KB page boundary and so-called huge pages do not alleviate this problem as the HW prefetcher is unaware of the larger page boundary [9]. Also, the number of huge page Data Translation Look-aside Buffer (DTLB) entries are very limited, and their allocation requires specific API calls. Software (SW) prefetching is performed by using the prefetch instructions that are part of the instruction set architecture (ISA) on many processors. Compilers perform SW prefetching by adding prefetch instructions to the code after analyzing memory access patterns statically, which increases the amount of cache hits on the critical path. Each prefetch mechanism has its own advantages and disadvantages. However, currently both mechanisms are designed and fine-tuned for only DRAM-based main memory, and SW prefetching is very conservatively utilized. For instance, the “prefetch-loop-arrays” pass in GCC [10] determines the read and write prefetch distances based on mostly the L1 cache size, DRAM access latency and the 64-byte cache line granularity used for DRAM on Intel architectures. It is very rigid in terms of the heuristics it utilizes to decide if prefetching is profitable or not and explicitly tries to avoid clashing with the HW prefetcher.

SW DTLB prefetching was proposed almost three decades ago [11] but is not implemented in any modern systems or compiler frameworks. The DTLB is an address translation cache that contains entries mapping page numbers to page
frames. DRAM/NVM based hybrid memory systems aim to support larger datasets for applications, which means more memory pages will be fetched by the CPU [9]. This will create additional pressure on the DTLB because every DTLB-miss is served by a page walk across a 4 or 5 level radix tree, which can incur a high cost on performance at times. Further performance degradation can occur if the page walk results in a page fault and main memory has to be accessed to retrieve page table indirections. NVM access latency can further add to this problem. As we move towards hybrid memory architectures, prefetch mechanisms need to adapt in order to mitigate the performance degradation of applications.

High performance computing (HPC) applications frequently use different solvers, e.g., Partial Differential Equations (PDE), Fast Fourier Transform (FFT) and Conjugate Gradient (CG) [12]. The solvers consist of kernels that perform heavy compute and memory operations on large structured arrays. They perform computations, e.g., using various stencil shapes, which display strong spatial and temporal locality. It is critical to take advantage of the locality of references for good performance of the HPC applications on hybrid memory systems. Upcoming HPC systems will have hybrid memory devices that require a more effective prefetch methodology to achieve good performance.

This paper analyzes the performance of a DRAM/NVM based hybrid memory system for different HPC kernels. Computational kernels are enhanced by SW prefetch instructions and assessed in their relative effect on performance. The objective here is to characterize the prefetch performance for hybrid memory systems while executing HPC workloads so that the current prefetch mechanisms can adapt to new and potentially hybrid memory architectures.

Section II summarizes previous work on hybrid memory systems and SW prefetching. Section III provides an overview of the “prefetch-loop-arrays” pass in GCC and explores the architecture of a hybrid memory system. Section IV presents our experimental setup and the custom benchmark developed in this work for evaluation. Section V presents and discusses results and observations. Section VI proposes modifications to the “prefetch-loop-arrays” pass to achieve SW adaptive prefetching. Section VII summarizes our contributions.

II. RELATED WORK

A number of recent studies have been conducted recently after the launch of Intel’s Optane DC PMMs. Yang et al. and Izraelevitz et al. [13], [14] evaluated the read and write memory access characteristics of Optane DC PMM for different file-systems, database applications and performance benchmarks. They found that Optane DC improves the performance of file systems and database applications due to lower latencies than storage devices. Patil et al. [15] characterized the performance of a DRAM/NVM hybrid memory system for HPC applications. They measured the bandwidth performance and energy characteristics of HPC applications runs on Optane DC compared to pure DRAM and DRAM as cache for Optane DC. Peng et al. [16] evaluated Optane DC PMMs in all the configurations available and also measured the performance of separating read and write allocation on a DRAM/NVM memory system. All the above works focus on evaluating the basic performance characteristics of Optane DC under various execution contexts and workloads. Our work primarily focuses of characterizing the effects of prefetching and utilization of the cache in in a byte-addressable DRAM/NVM hybrid memory address space for HPC workloads.

Several works have been conducted on utilizing SW prefetching in order to improve performance on traditional DRAM based memory systems. Callahan et al. [17] were the first to introduce SW prefetching as non-blocking prefetch instruction to eliminate cache miss latency. Mowry et al. [18] introduced and evaluated compiler based SW prefetching that worked in coordination with the HW prefetcher. Bala et al. [11] introduced SW prefetching for DTLB to decrease both the number and cost of kernel DTLB misses. Margaritov et al. [9] proposed a HW-based DTLB prefetch mechanism to reduce the address translation times in modern processor systems. Badawy et al. [19] evaluated the use of SW prefetching and locality optimizations for various HPC workloads for DRAM-based memory systems and found that for some cases SW prefetching has more benefits.

Fuchs et al. [20] designed a HW prefetcher for code block working sets that predicted the future memory accesses of stencil based codes. Swamy et al. [21] introduced a hardware/software framework to support efficient helper threading on heterogeneous manycores, where the helper thread would perform SW prefetching to achieve higher sequential performance for memory-intensive workloads. Zhao et al. [22] used a dynamic approach to pool memory allocations together and fine tuned the data prefetching based on the access patterns observed in the profiler. Islam et al. [8] evaluated hardware prefetching in a flat-addressable homogeneous memory comprising HBM and phase change memory (PCM), where a large buffer was placed in HBM to hide the access latency of PCM. Meswani et al. [7] explored various schemes to manage the heterogeneous memory architectures and then refined the mechanisms to address a variety of HW and SW prefetch implementation challenges. Lee et al. [23] evaluated both SW and HW prefetching for various workloads and suggested techniques for cooperative HW/SW prefetching. We aim to utilize DTLB- and cache line-based software prefetching in order to adapt to the upcoming hybrid memory systems with fast and slow access memory devices while improving the performance of HPC workloads.

III. ARCHITECTURE

A. GCC prefetch-loop-arrays compiler pass

Mowry et al. [18] designed the GCC compiler pass to optimize HPC workloads with SW prefetch hints that work in coordination with the HW prefetcher. This section analyzes the operational characteristics of their prefetch algorithm. The algorithm aims to be fine tuned for DRAM-based memory systems. All constants and heuristics are fixed to values that conform with DRAM specifications, which differ from system to system. The algorithm works on a per loop basis:
1) Gather all memory references in the loop and convert them into a \( \text{base} + \text{step} \times \text{iter} + \text{delta} \) form. Classify them as read or write references and form groups of references based on \( \text{base} + \text{step} \) to identify different access patterns.

2) Calculate the profitability for each memory reference using a heuristic-based cost model that takes into account the total number of instructions, memory references, the miss rate, trip count, prefetch-ahead distance, unroll factor, the maximum prefetch-slots and temporal locality.

3) Determine the references that can be prefetched based on maximum prefetch-slots and prefetch-modulo (relative to the loop iterator).

4) Unroll the loops to satisfy prefetch-modulo and prefetch-before constraints in prologue size, but without loop peeling.

5) Emit the prefetch instructions.

The cost model used to determine the profitability of prefetching for different memory reference groups is as follows:

- First determine the prefetch-modulo and prefetch-before for every memory reference using the \( \text{step} \) and \( \text{delta} \) values. Their temporal locality is determined data dependence analysis on the memory references.
- The prefetch-ahead is determined by using the ratio of \( \frac{\text{fetch_time}}{\text{iteration_time}} \) indicating how far ahead to prefetch. Both these values are unavailable at compile time. Instead, target-specific constants are used to make an "educated guess".
- The acceptable miss rate is calculated based on the references that have the same \( \text{base} + \text{step} \times \text{iter} \) but different \( \text{delta} \). If \( \text{delta} \) exceeds cache line size then it is determined to be a miss. If the probability of this miss is less than 0.95, then prefetching is considered to be unprofitable.
- It determines if the loop has enough iterations to justify prefetching ahead using the trip-count-to-ahead-ratio with a cut-off threshold of 4 (i.e., no prefetching below this threshold).
- It also calculates the ratio between \( \frac{\text{total_instruction_count}}{\text{memory_references}} \) to determine if the loop has enough CPU instructions to overlap the cache misses. If the ratio is smaller than the machine specific threshold, no prefetching is done.
- It also calculates the prefetch-cost via the ratio of \( \frac{\text{total_prefetch_count}}{\text{total_instructions_count}} \). If this cost is too high, then no prefetching is performed.
- The ratio of \( \frac{\text{prefetch_mod}}{\text{unroll_factor}} \) is calculated with a threshold value of 4, below which no prefetching occurs.

Some of these thresholds are overly strict even by DRAM standards. If the same prefetching parameters were used for NVM memory accesses, the algorithm would fail to gauge the most efficient prefetch configuration or not perform prefetching at all. Although the algorithm acknowledges different types of streams/access patterns in a kernel, it does not consider varying thresholds. Hence, it cannot adapt to different memory access patterns or memory technologies. Also, all parameters are defined based on cache and cache line sizes. No DTLB prefetching is considered.

B. DRAM-NVM hybrid memory architecture platform

The system used in experiments is a single HPE ProLiant DL360 node with 2 CPU sockets equipped with Intel’s Xeon 8260 (code-named Cascade Lake). Each chip has 24 cores with a clock frequency of 2.4 GHz. Each core has 2 processing units under hyperthreading for a total of 96 CPUs. Each core has a 32 KB private L1 instruction cache, a 32 KB private data cache, and a private 1 MB L2 cache. There is a 35.75 MB L3 cache shared between all cores. It has a DTLB cache with 64 entries, which is 4-way set associative.

Each socket has 12 DIMM slots. 6 of the slots are occupied by 16 GB DDR4 DRAM modules and the other 6 slots are occupied by 128 GB Optane DC modules for a total of 192 GB DRAM and 1.5 TB NVM. The node has 4 memory controllers in total, two are connected to 6 DRAM DIMMs each, and the other two, known as iMC, are connected to 6 NVDIMMs each. The processor uses the standard DDR4 protocol on the regular DRAM memory controller and the DDR-T protocol for Optane DC on the i-memory controller (iMC). Using this proprietary extension of the protocol, the Optane DC achieves asynchronous command/data timing and variable-latency memory transactions. Optane DC has an on-DIMM Apache Pass controller that handles memory access requests and the processing required on NVDIMMs. The on-DIMM controller internally translates the addresses of all access requests for wear-leveling and bad-block management. It maintains an address indirection table on-DIMM that translates the DIMM’s physical addresses to an internal device address. The table is also backed up on DRAM.

Accessing data on Optane DC occurs after the translation. The controller translates 64 byte load/stores into 256 byte accesses due to the higher cache line access granularity of Optane DC, which causes write amplification [14]. Optane DC PMM can operate in different modes (1) as an uncached byte-addressable memory (Flat mode), (2) as DRAM cached main memory (Memory mode), or (3) as a block storage device (App-Direct mode). All modes (except for Flat) are provided by Intel. Flat is a custom mode introduced by patching the OS kernel to identify all DIMMs as DRAM, thereby creating a true hybrid memory address space. All experiments are performed on the Flat-mode.

IV. EXPERIMENTAL SETUP

The aim of this experiment is to characterize the performance of SW prefetching for different prefetch distances under temporal and non-temporal prefetching with allocations on DRAM and NVM separately. We switch HW prefetching on and off for all experiments to evaluate its effect, which requires a reboot after toggling on/off three BIOS setting: HW Prefetch, Adjacent Sector Prefetch, and DCU Stream Prefetch. We compare the SW prefetch performance with and without “prefetch-loop-arrays” compiler optimization of GCC 9.3.0 while using the O3 flag for all compilations. The symbiotic SW prefetching runs are compiled with “no-unroll-loops” in order to measure the effect of varying unroll distances.
We developed a custom benchmark that allows us to measure the prefetch performance for different kernels frequently occurring in HPC applications. These kernels include a write-only (Wr-only) stream, single-write-multiple-read stream (1W4R), and 3-, 5-, 7-, 9- and 27-point stencil streams. The Wr-only stream kernel consists of 5 sequential write streams of linear arrays. The 1W4R kernel has one write stream and four read streams, which are also accessed sequentially. All stencil kernels consist of a write stream and a read stream of a 3-dimensional (3D) dataset of linearly laid out arrays accessed in row-major order using three nested for loops.

The stencil codes are implemented as Jacobi iterative kernels, which are common in Computational Fluid Dynamics (CFD) applications, Partial Differential Equations (PDEs), and pointual automata [24]. Some examples of stencil code-based HPC applications are Vector Particle In Cell (VPIC) [25]–[27] and Algebraic Multi-grid (AMG) [28], which are compute- and memory-bound applications, respectively. The 3-, 5- and 7-point stencils use the Von Neumann neighborhood whereas the 9- and 27-point stencils use the Moore neighborhood [29]. The 3-point stencil is a one-dimensional (1D) stencil, where for every iteration the previous element and the next element are read along with the current one. The 5-point stencil is a two-dimensional (2D) stencil, where along with adjacent elements in the same row of the current element, adjacent elements in the same column of the current element are also read. The 7-point stencil is a 3D stencil, where along with the adjacent elements in the same row and column of the current element, adjacent elements in the next and previous plane are read. The 9-point stencil is a 2D stencil including diagonal elements beyond the 5-point stencil. Similarly, 27-point stencil is a 3D stencil with diagonals on every dimensional pair beyond the 7-point stencil. These stencils comprise one or more read streams, plus a write stream accessed sequentially. Each stream is 4 GB in size and is allocated separately on each NUMA node using numa_alloc_node() for every run.

We manually unroll and peel the kernels. Each kernel has a prologue and an epilogue loop. The prologue loop prefetches each element of the stream sequentially until up to given read or write prefetch distance. The main compute kernel is unrolled up to the unroll distance and the next elements are prefetched after unrolling is complete. The main loop stops when there are no more elements to prefetch and the remaining iterations are completed in the epilogue loop. Due to the variability in the read and write distances, the prologue and epilogue loops are split and separated by conditional statements to avoid over-prefetching and segmentation faults.

We use the GCC builtin_prefetch() function to prefetch the desired cache line at every iteration [30], which automatically calls the corresponding intrinsic for a given instruction set architecture. We change the read and write prefetch distance explicitly from 32 bytes to 16,384 bytes using nested loops that encompass all the kernels. The upper bound of the prefetch distance is kept at 16,384 bytes to avoid L1 cache contention at higher distances. We change the parameters of the prefetch call to perform non-temporal and temporal prefetching for linear and stencil read streams, respectively. We use non-temporal prefetching for all write streams. For non-temporal prefetching, the data is fetched into a non-temporal cache structure to avoid cache pollution; whereas for temporal prefetching the data is prefetch into L2 cache or higher [31], [32]. We also vary the unroll distance in another nested loop from 4 to 64. We limit the unroll distance to 64 to restrict additional pressure on available CPU registers. We perform cache line prefetching with distances from 32 to 2,048 and perform DTLB page prefetching from distances from 4,096 to 16,384. This is enabled by adding a conditional statement to the prefetch statement block, which prefetches only after certain number of iterations have elapsed nearing the page boundary. DTLB caching causes the address translation to be stored in the 4-tier DTLB cache, which reduces future page faults that are expensive in terms of CPU cycles [33].

We refer to our technique as “symbiotic prefetching” from here on. We execute the benchmark on 48 processes running individually on each core launched by MPI, but without imposing communication via message passing, i.e., just to provide realistic workloads in separate address spaces with contention on shared resources (last-level cache, DRAM/NVM). We divide the stream size equally between all processes and allocate them separately. We pin all processes to the cores and then calculate the data bandwidth for DRAM and NVM allocations using the ratio of total memory of all data structures accessed and wall clock time measured for each kernel. Each measurement is averaged over 10 runs, where a standard deviation of 4%-6% is observed for all kernels. To obtain cache performance metrics, we use the LIKWID Marker API [34] to measure metrics for every individual kernel obtained from HW performance counters. The performance counters are obtained for the configuration that delivers highest performance benefit and then reported relative to measurements of the same kernel without any prefetching as a baseline. We also perform the same comparison for the kernels when they are compiled using “prefetch-loop-arrays”. We term this as compiler prefetching from here on.

V. RESULTS

This section discusses the results of experiments and present observations. Prefetch performance is depicted as percentage changes over different compilation options. We depict percentage changes in data bandwidth observed with symbiotic prefetching (sp) as a heat map over a 3D graph relative to the baseline bandwidth observed when compiled with no prefetching (np). The x-axis depicts the write prefetch distance in bytes, the y-axis the read prefetch distance in bytes, and the z-axis the unroll distance in number of iterations. The heatmap colors represent percentage changes in bandwidth relative to the baseline. We plot the graphs for DRAM (left) and NVM (right) separately.

We only present heatmaps for the 7-point stencil in Figures 1 and 2 due to space limits, but we report and discuss the results (data bandwidth and performance metrics) for all kernels as all have similar graphs, albeit with different best values (subject of a forthcoming technical report). Figure 1 depicts
results of symbiotic and compiler prefetching for the 7-point stencil kernel without HW prefetching. Figure 1a and 1b depict performance changes under symbiotic prefetching for DRAM and NVM, respectively, where an arrow indicates the highest benefit configuration (write, read, unroll distances).

Observation 1: For DRAM, bandwidth increases for larger read prefetch distances (x-axis) but abruptly drops close to the largest write prefetch distances (y-axis) and is best for a small unroll distance (z-axis).

Observation 2: For NVM, the bandwidth decreases toward higher write and read prefetch distances (x+y axes) and, in contrast to DRAM, is best for largest unroll distance (z-axis).

Observation 3: For DRAM, symbiotic prefetching results in slightly higher L1-DTLB load misses, and even slightly more for compiler prefetching. For NVM, L1-DTLB load misses are significantly higher by a factor of 11x (1100%) under symbiotic prefetching compared to no SW prefetching — where compiler prefetching actually reduces the L1-DTLB load misses for NVM. Symbiotic prefetching also reduces the L1-DTLB store misses for NVM whereas compiler prefetching increases them.

Given the structure of a 3D 7-point stencil, a total of 5 read streams and 1 write stream exist. Hence, performance of the kernel critically depends on the availability of read data in the cache. As the stencil moves across the data set, there is reuse of all neighboring read data points. Hence, it is prefetched as temporal data into the L2 and L1 caches. The write data points do not have reuse, and are hence prefetched as non-temporal data into a separate cache structure. However, if the latency of the memory accesses is large then a high unroll distance is required to reduce CPU stall cycles. The dependence on unroll distance is reflected in the NVM heatmap where the performance benefits from higher unroll distances, which increases temporal reuse of read data and overlaps prefetch latency of the write stream with computation. Nonetheless, a short unroll distance is sufficient for faster DRAM memory accesses under DTLB prefetching for the read streams. The pages for read data are quickly cached into the DTLB, which increases the reuse within read streams by reducing page walks and page faults. Further, short unroll distances are sufficient to overlap with prefetches spaced according to the write stream. This is also reflected in hardware counter metrics, where a reduction in L2, L3 and L1-DTLB store misses is observed for symbiotic prefetching on DRAM, which is the source of the performance benefit. For NVM, the L1-DTLB load misses are high due to a smaller read prefetch distance, and the reduction in store misses provides the main performance benefit. Cache and memory bandwidths increase as a result of reduced L1-DTLB misses.

Let us also consider Figure 3a in this context, which depicts the performance of all prefetching methods relative to no prefetching as a baseline per kernel; and Figure 3b, which depicts the performance comparison of HW prefetching relative to no HW prefetching as a baseline. The y-axis depicts the percentage change in data bandwidth and the x-axis lists all benchmark kernels for both figures. For the 7-point stencil, compiler prefetching is not able to provide any performance benefit as seen in Figure 3a whereas symbiotic prefetching on the other hand provides a 12% and 19% performance benefit for DRAM and NVM, respectively, over no prefetching. This is also reflected in the hardware counters, where compiler prefetching shows the smallest changes over these metrics. This results from tight bounds plus low margins on heuristics and greater dependence on HW prefetching, where its absence harms performance.

Inference 1: The prefetching configurations show diametric behavior for the same kernel when its streams are allocated on DRAM and NVM. SW prefetching provides benefits for both DRAM and NVM without relying on or being complemented by the HW prefetcher.

Figure 2 depicts the results for symbiotic and compiler prefetching for the 7-point stencil kernel with HW prefetching, with the same subfigures as before.

Observation 4: For DRAM, the bandwidth increases toward lower write and also slightly toward higher read prefetch distances, and it slightly increases for smaller unroll distances.

Observation 5: For NVM, the data bandwidth increases as
we move higher on all three axes.

**Observation 6:** L1-DTLB load misses for DRAM increase under symbiotic prefetching but decrease under compiler prefetching. L1-DTLB store misses are reduced by symbiotic prefetching, whereas compiler prefetching increases it. A similar but more profound value change is observed for NVM.

As symbiotic prefetching operates independently of HW prefetching, hardware counters change according to bandwidth trends of Figures 2a and 2b. The increase in darkness on the heatmap indicates that HW and SW prefetching together are able to provide the best performance benefit. For DRAM, additional HW prefetching does not significantly affect symbiotic prefetching but complements its performance by enabling higher read prefetch and unroll distances to better take advantage of temporal locality for this 7-point stencil kernel. Every dispatched load operation by the HW prefetcher is first dispatched to the L1 cache and DTLB. Upon an L1 miss, it is relayed to lower caches, which can incur up to 80 cycles to be fetched. However, a TLB miss can result in a page walk or page fault accounting for thousands of cycles. However, due to the DTLB prefetching, the dispatched load encounters a TLB hit most of the time reducing the overhead of page walks and page faults.

Nonetheless, the addition of HW prefetching provides benefits (see Figure 3a). NVM is also assisted by HW prefetching with larger prefetch and unroll distances for symbiotic prefetching compared to no HW prefetching. SW read prefetching affects the DTLB cache, which results in fewer L1-DTLB load misses relative to the baseline (only HW prefetching). Here, HW prefetching serves the temporal locality of the access pattern. This frees up the SW prefetcher, which is now utilized to reduce TLB misses. Hence, the performance
benefit of symbiotic plus HW prefetching results in twice the performance gain compared to no HW prefetching for NVM (see Figure 3a). Compiler prefetching cannot improve performance by much without DTLB prefetching; it is DTLB prefetching that provides the main source of improvement (see heatmaps of DRAM and NVM).

**Inference 2:** SW prefetching improves DTLB performance in a manner symbiotic to HW prefetching driving cache benefits for the kernel for both DRAM and NVM.

Any requests of the HW prefetcher benefit upper caches while symbiotic prefetching effectively becomes DTLB prefetching and results in reduced L1-DTLB load misses. Although this is observed for NVM, symbiotic prefetching is not able to increase the L1-DTLB load misses for DRAM, as the latter (DRAM) does not benefit in performance. Due to fewer read streams, HW prefetch requests populate the DTLB cache quickly enough that symbiotic prefetching becomes redundant. The heatmaps of all the kernels show a similar behavior with subtle differences owing to the number of read and write streams present in the kernel. The SW prefetch configurations indicated by the arrows that deliver the highest performance benefit for each kernel are summarized in Table I. The table reinforces the inference that SW prefetching should be utilized alongside HW prefetching to combine DTLB caching and data caching due to the symbiosis between former and latter prefetch techniques, respectively. But configurations have to be adapted to the underlying memory technology (DRAM vs. NVM) and specific access patterns.

**Observation 7:** We observe that symbiotic prefetching by itself provides more performance benefit than compiler prefetching for DRAM and NVM over all kernels. The highest performance benefit is observed when symbiotic prefetching is used in conjunction with HW prefetching from 4 to 26%. The performance for NVM degrades when combined with HW prefetching for all kernels, except for 1W4R. But notably, symbiotic prefetching mitigates this degradation (except for the rather simplistic Wr-only and 1W4R kernels). In contrast, compiler prefetching remains ineffective, i.e., any degraded performance cannot be reduced. The HW prefetcher degrades the performance of linear array streams on DRAM, and SW prefetching is unable to mitigate this problem.

**Inference 3:** HW prefetching only serves short-distance read prefetches that have high temporal locality, which are typical signatures of 1D or 2D stencil kernels. HW prefetching also needs to be adaptive to access patterns in order avoid performance degradation. Here, SW prefetching comes to the rescue as it mitigates these performance degradations on NVM.

VI. ADAPTIVE SW PREFETCHING AS A COMPILER PASS

Based on our observations and inferences, we propose the following changes to the “prefetch-loop-arrays” compiler pass in GCC:

- To adapt to a hybrid memory, compilers should become memory allocation-aware. This can be accomplished by overloading malloc() where the programmer along with the size can also specify the desired memory device and the NUMA nodes need to be mapped to the correct memory device to complete the allocation.
- The pass already identifies different streams and their access patterns in any given loop. Hence, classifying specific workloads is feasible and should be incorporated.
- Instead of a single set of constants and threshold values to guide heuristics and the cost model, a static table for each memory technology should be maintained with specific constants/thresholds per access pattern. Our contributions in this work lay the foundation to automatically derive these constants and thresholds in a calibration run, which then allows the derivation of values similar to Table I. Once access pattern and memory type of streams have been determined, the compiler pass can readily decide on prefetches given the specific constants/thresholds.
- Condition check for non-temporal locality should be removed, and non-temporal prefetches should be supported. Checking of the HW prefetcher stride needs to be lifted to allow for both symbiotic SW and HW prefetching.
- The acceptable miss rates need to be lowered for large distances to accommodate DTLB prefetching. Similarly, many thresholds (trip-count-to-ahead-ratio, prefetch-mod-to-unroll-factor-ratio, memory-ref-count-reasonable and insn-to-prefetch-ratio-too-small also) need to be higher for symbiotic prefetching to allow DTLB prefetching in software — as well as support for higher unroll distances that are adaptive for slow and fast memories.
- With predictable access patterns, priority ordering of prefetches based on lowest prefetch-modulo can be replaced by the (more easily) predicted performance benefit.
- Finally, loop peeling is required when emitting prefetch instructions at the end of the pass.

The changes are beyond the scope of this work, but only become feasible due to the contributions of our memory-hybrid, adaptive and symbiotic prefetching.

VII. CONCLUSION

Our work provides novel insight that the existing rigid and conservative approach to SW prefetching leaves ample performance potential on the table for HPC workloads. We show that existing HW prefetchers are neither optimized for NVM memory nor for non-temporal workloads. We contribute HW and SW prefetch methods that are more adaptive and show that they succeed in extracting symbiotic performance while being sensitive to hybrid memory systems. Our DTLB-based symbiotic SW prefetching improves the performance of HPC kernels from 4 to 26% for data streams allocated on both DRAM and NVM, and our SW prefetching complements HW prefetching rather than competing with it. We also present a simple design to modify an existing SW prefetch compiler pass to implement our prefetch configurations with the potential to automatically improve performance for HPC workloads on future hybrid memory systems.

REFERENCES

TABLE I: Symbiotic prefetching configurations and performance benefits

<table>
<thead>
<tr>
<th>Kernel</th>
<th>Prefetch Distances(bytes), Unroll length and Performance Improvement(%)</th>
<th>NVM</th>
<th>ACX</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>With HW prefetch</td>
<td>Without HW prefetch</td>
<td>With HW prefetch</td>
</tr>
<tr>
<td></td>
<td>Read</td>
<td>Write</td>
<td>Unroll</td>
</tr>
<tr>
<td>Write-only</td>
<td>256</td>
<td>64</td>
<td>-</td>
</tr>
<tr>
<td>1W4R</td>
<td>128</td>
<td>64</td>
<td>96</td>
</tr>
<tr>
<td>128</td>
<td>38.19</td>
<td>34.66</td>
<td>-</td>
</tr>
<tr>
<td>7pt stencil</td>
<td>64</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>32</td>
<td>64</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>128</td>
<td>38.19</td>
<td>34.66</td>
<td>-</td>
</tr>
<tr>
<td>7-pt stencil</td>
<td>8192</td>
<td>64</td>
<td>12</td>
</tr>
<tr>
<td>256</td>
<td>8192</td>
<td>64</td>
<td>12</td>
</tr>
<tr>
<td>32</td>
<td>8192</td>
<td>64</td>
<td>12</td>
</tr>
<tr>
<td>128</td>
<td>38.19</td>
<td>34.66</td>
<td>-</td>
</tr>
<tr>
<td>7-pt stencil</td>
<td>1024</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>256</td>
<td>1024</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>27-pt stencil</td>
<td>1024</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>256</td>
<td>1024</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>32</td>
<td>1024</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>278-pt stencil</td>
<td>1024</td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>256</td>
<td>1024</td>
<td>32</td>
<td>64</td>
</tr>
</tbody>
</table>

Improving NAND flash performance with read heat separation

Roman Pletka, Nikolaos Papandreou, Radu Stoica, Haris Pozidis, Nikolas Ioannou
IBM Research, Zurich
{rap, npo, rst, hap, nio}@zurich.ibm.com

Tim Fisher, Aaron Fry, Kip Ingram, Andrew Walls
IBM Systems, USA
{fisher, aaronfry, ingram, awalls}@us.ibm.com

Abstract—The continuous growth in 3D-NAND flash storage density has primarily been enabled by 3D stacking and by increasing the number of bits stored per memory cell. Unfortunately, these desirable flash device design choices are adversely affecting reliability and latency characteristics. In particular, increasing the number of bits stored per cell results in having to apply additional voltage thresholds during each read operation, therefore increasing the read latency characteristics. While most NAND flash challenges can be mitigated through appropriate background processing, the flash read latency characteristics cannot be hidden and remains the biggest challenge, especially for the newest flash generations that store four bits per cell.

In this paper, we introduce read heat separation (RHS), a new heat-aware data-placement technique that exploits the skew present in real-world workloads to place frequently read user data on low-latency flash pages. Although conceptually simple, such a technique is difficult to integrate in a flash controller, as it introduces a significant amount of complexity, requires more metadata, and is further constrained by other flash-specific peculiarities. To overcome these challenges, we propose a novel flash controller architecture supporting read heat-aware data placement. We first discuss the trade-offs that such a new design entails and analyze the key aspects that influence the efficiency of RHS. Through both, extensive simulations and an implementation we realized in a commercial enterprise-grade solid-state drive controller, we show that our architecture can indeed significantly reduce the average read latency. For certain workloads, it can reverse the system-level read latency trends when using recent multi-bit flash generations and hence outperform SSDs using previous faster flash generations.

1. INTRODUCTION

Today, 3D-NAND flash is the prevalent memory technology in consumer and enterprise solid-state drives (SSDs) as well as mobile devices. The transition from 2D to 3D NAND flash enabled drastic storage density increases and cost reductions but also helped to address endurance limitations [1], [2]. Storage density continues to increase by the addition of more layers and stacking groups of layers, enhancing the number of bits stored per memory cell.

All these techniques affect the latency characteristics in multiple ways [3]. First, storing more bits per cell exponentially increases the number of threshold voltage distributions. For instance, when moving from triple-level cells (TLC) to quad-level cells (QLC), the number of threshold voltage levels grows from 8 to 16. Figure 1 gives an overview of the typical average read latency ranges from different 3D-NAND devices on the market. The read latencies of multi-level cells (MLC), TLC, and QLC, are normalized to the average latency of the single-level cell (SLC) technology. As can be seen, latency rises with increasing numbers of bits stored per cell.

Second, the denser the threshold voltage distributions, the higher is the likelihood that these distributions overlap, hence, more elaborate read algorithms are needed [4]. Flash media characterization has shown that QLC NAND flash is significantly more susceptible to errors than previous generations [5], [6]. The impact of these drawbacks on read latency is one of the main reasons why SSDs using QLC NAND flash are not widespread in the enterprise storage market today.

The latency degradation is not equal across all flash page types [7]. As we will discuss, some flash pages may have a lower latency and error rate than other pages. By placing frequently read data on fast flash pages, we can achieve a reduction in the average latency, reduce the number of high-latency reads which results in a tighter latency distribution, and reduce the probability of read retries. This idea is promising conceptually, but challenging to implement in practice. Implementing RHS in a flash controller is constrained by computational, space, and bandwidth limitations as the technique must be implemented in hardware and is part of the critical data path. More specifically, the challenges are:

- The size of the read heat metadata is strictly limited by the DRAM capacity available in an SSD. Alternatives such as paging read heat metadata from flash, introduce additional latency and would defeat the purpose.
- Flash controllers have an upper bound on the computational complexity as they cannot afford to introduce additional latency in the critical path. Generally, the chip real estate and power envelope is limited.
- It is not evident how a controller should use read heat information with minimally interfering the data path and other background operations.
- Initially there is no read heat information available, which is undesirable for real applications, as the read latency can be initially very high, but improve over time.

![Fig. 1: Average read latencies of different 3D-NAND flash cell technologies normalized to a typical average SLC latency.](image-url)
The contributions and findings of this paper can be summarized as follows:

- We present a new controller architecture that accommodates RSH in a non-disruptive fashion.
- We present different bit encoding schemes as well as scalable and lightweight read heat tracking strategies that explain the various tradeoffs involved.
- We evaluate the efficiency of RSH for various heat tracking schemes. This is the first study that analyzes the impact of read skew overlap and the ability to tolerate workload changes.
- We implemented our proposal in a simulation environment and evaluate our findings.

The remainder of this paper is structured as follows: Section II describes the background necessary for understanding the rest of the paper. Section III analyzes the potential average read latency reduction that can be achieved, and show that a QLC controller can ε equal or better performance characteristics when coupled with a traditional TLC controller architecture for workloads. Section IV demonstrates the end-to-end efficiency of our technique by implementing it in a commercial enterprise-grade controller architecture. Section V summarizes the related work before concluding our paper in Section VI.

II. BACKGROUND

We first describe some of the relevant NAND flash memory device properties and then give an overview of existing flash controller architectures.

A. Properties of NAND flash memory devices

NAND flash cells store data as electrical charge placed in the floating gate or charge trap layer of the transistor forming the cell. Typically, several thousand cells are connected through a single word-line (WL), while several hundreds or more than a thousand WLs form a block. Depending on the amount of charge stored, the cell belongs to one of several states or levels. Figure 2 depicts the probability density function (PDF) of the threshold voltage $V_{th}$ distribution in a TLC NAND flash device and the associated states. The number of states $l$ is given by the number of bits $n$ stored on the center bit (CB), and the least-significant bit (LSB) to the erase and program states of a TLC flash cell. A physical page is the granularity at which data is programmed. As charge cannot be removed from cells, programming can only increase the amount of charge in a cell and an extra erase operation is needed to remove the charge of all cells in a block. When reading a physical page, only a single bit of information is extracted from each cell of the selected WL. As only a single read voltage level can be tested at a time, the read out is done in an iterative process where each one of the tested threshold voltages from the set of read voltages $V_1, ..., V_7$ adds up to the read latency.

B. Gray coding

Gray coding schemes, also known as reflected binary coding, order the binary values in a code such that two successive values differ in only a single bit [8]. In flash, charge variations in cells may result in the crossing of a single read voltage level such that a neighboring state is detected upon a read operation instead of the originally programmed one. Such crossings increase the raw bit error rate (RBER). Preferably, one would like to minimize the number of affected bits from a single crossing event. The properties of Gray codes guarantee that only one physical page sees an increase in the error count from a single crossing.

There are many variants of Gray codes. Here, we focus on three particular Gray codes to pinpoint their effectiveness in RSH: The original reflected binary coding (ORBC) [8], a partially-balanced binary code (PBBC), as well as a maximally balanced binary code (MBBC) [9]. Typically, flash devices only implement a single coding scheme.

An example of the ORBC is given in Figure 2, where the read threshold voltage $V_4$ is applied to read the LSB, thresholds $V_2$ and $V_6$ the CB, and $V_1$, $V_3$, $V_5$, and $V_7$ the MSB. Table I gives an overview of the three selected Gray coding schemes for a QLC NAND flash device. The four physical pages in a WL are sorted in the programming order. In ORBC, reading a $P_d$ page implies testing eight read thresholds resulting in a significantly higher read latency than the $P_u$ page. PBBC retains the single threshold for the $P_d$ page, but balances the number of read thresholds for all other page types while the MBBC balances all read thresholds.
C. Controller architectures

Early multi-bit NAND flash devices only support a single mode meaning that the provided command set only allowed for programming all bits in the cells to operate in multi-bit mode or in a high-performance single-bit mode [11]. From a consumer perspective, hybrid flash controllers have the potential to improve performance of SSDs at a much lower cost, using workloads. A small number of SLC devices with many MLC devices were used to increase performance where the SLC portion was used as a fixed-size cache [10]. This architecture allows the cache and destages valid data later when the SLC cache is full. While these designs have write latency and throughput for bursty workload, the sustained write performance of SSDs is critical in real-world use cases [14].

Later, the second generation of hybrid controllers introduced adaptive SLC caches where the amount of SLC blocks is adjusted based on the current capacity utilization [12], [13]. Such controllers show excellent performance when the used capacity is low. When the used capacity increases, the SSD will hit the same sustained writes performance issue. A more recent approach tries to adjust the tier sizes and data placement strategy as a function of workload properties to optimize write performance and endurance [14].

III. DATA-Placement ARCHITECTURE

In this section, we first propose our data-placement architecture, describe lightweight alternatives for read heat tracking, and discuss the implications for a real controller.

A. Description of the architecture

Figure 3 illustrates our controller architecture with a data placement unit supporting RHS for QLC flash. The architecture is generic such that it is applicable to all types of flash controllers from Section II-C. All new components are marked in yellow. In flash controllers, a data placement unit picks erased blocks for writing data (1). A fully programmed block is moved to the occupied block pool (2). When the number of erased blocks is low, garbage collection (GC) selects a victim block, with the (close to) least amount of valid data from the occupied block pool (3) [15]. All data still valid is read and written to a new location (4), and cleaned blocks are erased and made available for data placement. This process is performed in the background such that a small number of already erased blocks are always available for data placement. Besides GC, other internal processes such as wear leveling (WLL) may select blocks from the occupied block pool. Finally, the logical-to-physical (L2P) table is used to map logical pages, also known as logical block addresses (LBAs), to physical locations in flash.

Host writes (5) are first placed into a cache (6). There are several types of caches typically used in SSDs. Simple destage buffers tend to be very small and are used to hide the flash write latency. Using a non-volatile destage buffer permits writes to be acknowledged to the host (7) before they are written to flash. They usually consist of storage class memory (SCM) such as MRAM, battery-backed DRAM or SLC flash. The cache could also be resized dynamically using a variable percentage of the flash blocks in SLC mode. In this case, blocks are initially configured in SLC mode as long as the used capacity is low which enables tracking of read heat information before data is destaged to QLC. As such a cache can hold up to a quarter of the total capacity, the L2P can be leveraged to point to either a location in the occupied block pool or the cache. This has the advantage that it already allows for the adjustment of the read heat while the host write is placed into the cache as the L2P has to be accessed.

When the non-volatile cache gets full, data is evicted from the cache using a cache replacement policy such as least-recently used (LRU) or first-in first-out (FIFO). The evicted data is then prepared for destaging to QLC and a relocation write command is queued to the data placement unit (8). The data placement unit maintains separate queues for each read latency class. The read latency classes discriminate between the different latency characteristics of physical pages. For example, using ORBC, each page type corresponds to one read latency class where the Pa page maps to the lowest and the Pd page to the highest read latency class. With PBBC, a single read latency class could cover write requests for the Pa and Pd pages. In our design, the queues can hold write commands to fill up to two QLC blocks. This is sufficient to achieve good separation as we will show in Section IV. Next, the read heat counter corresponding to the LBA being relocated is read from the L2P (9). The RHS unit maps frequently read LBAs to the lowest read latency class and rarely read LBAs to
the highest read latency class and so on. When accessing the read heat counter, the data placement unit may update its value depending on the used read heat scheme described below.

In flash, the physical pages in a block must be programmed in sequential order to minimize the propagation of programming errors. Hence, upon programming a physical page, the data placement unit dequeues a write command from the queue corresponding to the read latency class that best matches the current page type (10). In the case no such command is available, another one is taken from a neighboring queue. As soon as a page is programmed, the space in the cache can be freed.

Host reads (11) fetch data from either the location at which it is stored in the occupied block pool (12), the cache (13), or the QLC block in the data placement unit (14) that is currently being written. The location is determined by an L2P lookup. After the completion of the read operation, data is delivered to the host (15). All host read operations increment the corresponding read heat counter using one of the algorithms described below irrespective of where the page read is located.

B. Tracking read heat information

The key parameters that influence the efficiency of read heat tracking, are, the granularity at which read heat is tracked, the resolution of the read heat counters, the procedure to update the heat counters, and the mapping from the read heat counter value to a read heat stream.

1) Read heat counters: Preferably, we would like to collect read heat information for every LBA at a reasonable resolution. As the L2P maps LBAs, typically 4 KiB or 16 KiB today, to their physical location, it therefore sounds reasonable to integrate the read heat counters into the L2P to minimize memory accesses. However, per-LBA heat information at high resolution increases the size of the L2P by roughly 50% which is a non-negligible cost factor when the L2P is kept in DRAM for performance reasons as is the case for enterprise-level SSDs 1. Therefore, we extend the L2P with a space efficient k-bit saturating counter to track the read heat. Using k = 2, such a 2-bit counter increases the L2P by only a few percent and allows for tracking 4 heat levels which matches the maximum number of QLC page types. This permits a simple mapping from read heat to latency classes and hence QLC page types.

2) Incrementing read heat: When using low resolution counters, the read heat should be incremented probabilistically because the counters saturate quickly otherwise. For a k-bit saturating counter we denote \( p_i \) as the probability of incrementing the counter from value \( i \) to \( i+1 \) where \( 0 \leq i < 2^k - 1 \) and introduce

\[
\hat{p} = [p_0, \ldots, p_{2^k-2}]
\]

as the probability vector defining a read heat increase scheme. For write heat tracking, it has been shown that such a heat increase scheme is optimal and increasing the number of heat levels quickly leads to diminishing returns [16].

Note that SSD internal or higher-level scrub reads are not indicative of the read heat of data and should not increase the read heat counters. Without going into further details, this can be achieved, for example, using existing protocol features.

1We assume a 16-bit read heat counter and an SSD with a storage capacity of 10 TB using a logical page granularity of 16 KiB. This requires a mapping entry of at least 4 Bytes to address the full LBA space to which 2 Bytes would be added for the read heat counter.

\[
\begin{array}{|c|c|c|}
\hline
\text{RRHD} & \text{Reset} & - \\
\text{RHGW} & \text{Reset} & - \\
\text{RHWO} & \text{Reset} & - \\
\text{DGWO} & \text{Decrement} & - \\
\hline
\end{array}
\]

TABLE II: Summary of read heat decrease schemes.

3) Decrementing read heat: We consider four different read heat decrease schemes as summarized in Table II:

- **Random read heat decrease (RRHD)**: Whenever a read operation results in an increase of the read heat counter, another used LBA is randomly selected and its read heat is decremented.
- **Reset on host or GC/WLL writes (RHGW)**: When an LBA is overwritten or relocated by GC or WLL, the read heat counter is reset to zero.
- **Reset on host writes only (RHWO)**: The reset of the read heat counter is only done upon a host write to the LBA. Internal relocations do not change the read heat information.
- **Decrement on GC/WLL writes only (DGWO)**: Any internal relocation of an LBA will decrement its read heat counter by one. Host writes preserve the value of the read heat counter.

All four schemes are simple to implement in hardware. RRHD uses more resources than the others as it requires two accesses to the L2P and the controller has to select a counter for decreasing heat of an LBA that actually holds data.

C. Discussion on the architecture

The accuracy of RHS is influenced by the workload properties (e.g., the read and write skew, the used capacity, the overlap between the read and write LBA ranges), the size of the cache, the capacity of relocation write command queues, and the read heat tracking strategy. With a fixed-size cache, the gathering of read heat information while data is residing in the cache is very limited. When the cache is a destage buffer, a good separation can only be achieved after data is relocated from QLC-to-QLC.

From this perspective, a hybrid controller architecture with an adaptive SLC cache size has huge advantages and works in synergy with RHS. First, blocks are initially configured to operate in SLC mode and read heat information is collected over a long time frame until the device reaches a certain used capacity point at which destaging to QLC starts. Once blocks are converted to QLC, accurate read heat information is then available.

Second, a dynamically resizable SLC cache can be enhanced with an additional destage buffer. This has the following advantages: Updating read heat counters comes at no extra cost because the L2P has to be accessed at the same time.

Third, the SLC GC can be a simple circular buffer. Hence, SLC blocks remain about the same time (measured in incoming writes) in the pool during which read heat information is collected. Thus, RHS is significantly less sensitive to the GC policy of the QLC pool where the choice of an adequate GC algorithm is essential due to its limited endurance. For these reasons, our SSD controller evaluated in Section IV-F implements this architecture.

Note that our architecture can be combined with so-called superblocks that group flash blocks from different chips and planes together as well as write heat separation [2], [17]. We consider these approaches out of scope of this paper because they do not influence RHS.
IV. EVALUATION

In the first part of the evaluation, we focus on the average read latency at queue depth one (i.e., when a single read I/O is submitted at a time). As real SSD controllers vary significantly in their implementation, we want to exclude all effects not related to RHS such as the handling of internal parallelism and command prioritization. This allows us to study the fundamental aspects of RHS. Later on, we present results from a commercial enterprise-grade controller where we implemented RHS.

Most of our tests use skewed synthetic workloads. It has been shown that real-life workloads typically exhibit a skewed access pattern [18], [19], [20]. The skewed synthetic workloads we use in our evaluations follow a Zipfian distribution [21] to closely mimic real-life workload types. They have been widely used to study device characteristics [2], [22]. We denote these workloads as $\text{Zipf } x/y$ where we adjust the skew factor of the Zipfian distribution such that $x$ percent of generated operations access $y$ percent of the LBA space for the given device size.

A. Simulation environment

Our simulation environment is inspired from a real state-of-the-art flash controller developed in-house. The full flash translation layer (FTL) is simulated with a usable capacity of 1TB consisting of more than ten thousand flash blocks. We found that scaling up the capacity does not affect the actual results obtained. The simulator uses a cyclic buffer GC policy which has the advantage of inherently relocating data and hence emulating retention or read disturb limits. This is reasonable as we are not interested in internal write amplification here.

All flash operations are emulated and the actual read latency values obtained from characterization are used to determine the achievable latency reductions. The simulator uses a minimally-sized cache large enough to hold data for two full QLC blocks on which the data placement unit performs data separation. Upon a host write, a relocation write command is immediately queued to the data placement unit. Host writes and GC queue relocation write commands until the queues are full. Then, the data placement operation is performed. The presented results are normalized to the average read latency of an SLC device from Figure 1.

B. The RHS potential of different bit encoding schemes

We analyze the achievable latency reductions for the Gray coding schemes presented above for various read workloads. The L2P mappings are initially randomized such that each LBA points to a different physical location, a typical precondition workload for characterizing SSDs [23]. Next, we issue a large enough Zipfian write workload (i.e., two full device writes) during which we perform RHS. Using the apriori knowledge of the read workload that we will exercise afterwards, we can easily assign each LBA to the correct read latency class to get the optimal data placement. Finally, we issue 2TB of reads following the desired read workload and measure the number of reads for each page type to evaluate the average read latency for each scheme.

The results are shown in Figure 4. When the reads follow a uniform random distribution on the entire LBA space, the data placement has almost no influence and the read latency matches the average read latency of all page types irrespective of the Gray coding scheme used. Significant latency reductions can be attained with skewed workloads: For all Gray coding schemes latency is decreasing with increasing workload skew.

Optimal data placement with ORBC and PBBC outperform a TLC device without RHS when the skew is higher than a Zipfian 80/20 and achieve more than $2.1 \times$ reduction in average read latency for Zipf 95/20. When the skew is higher than a Zipfian 80/20, they outperform a TLC device without RHS. Further, ORBC is only marginally better than PBBC, indicating that the capability to identify the read hot data is key for performing RHS. For FBBC, despite the thresholds being maximally balanced among all page types, a slight read latency reduction can still be observed owing to the $P_0$ page that requires sensing one threshold less than the other page types.

C. Comparing read heat tracking strategies

We now study the sensitivity of the different read heat tracking schemes. In contrast to the previous test, RHS is performed based on the read heat information gathered during the test without prior knowledge. The comparison focuses on the ORBC, which exhibits the highest read latency reduction above. To determine how close the different tracking strategies approach the theoretically achievable read latency reduction, we evaluate a set of workloads using a read-write ratio of 80/20 and 98/2 where both, reads and writes, follow Zipfian distributions with the same skew factor but with an offset that minimizes their overlap (i.e., most reads go to an LBA region that is rarely written to and vice versa).

Figure 5 illustrates the results from a large set of read heat increase and decrease strategies. The first strategy uses the same heat increase probability for all $p_i$ (Figures 5a and 5d). In the other two strategies, the increase probability drops exponentially with higher read heat values. In Figures 5b and 5e the first increase probability $p_0$ is fixed at $p_0 = 1$ to quickly detect whether a page has been read. The results in Figures 5c and 5f vary $p_0$ between 1 and 0.05, but use the same exponential decrease for $p_1$ and $p_2$. These increase schemes are combined with all four read heat decrease strategies from Section III-B3.

Overall, there is no single read heat tracking strategy that performs best for all scenarios and the performance varies significantly. First, a high read heat increase probability only performs well when the read-write ratio is reasonably low (Figure 5a). As soon as the read-write ratio increases and the workload is less skewed, read heat counters saturate quickly. In some cases,
almost no benefit from RHS can be measured (Figure 5d). Second, equal read heat increase probabilities $p_1$ are very sensitive to workload properties. They may work very well for highly skewed workloads, but perform poorly with less skew (Figures 5b and 5d). Therefore, these strategies should generally be avoided. Third, a reasonable high probability (i.e., $p_0 \geq 0.4$) is important – especially when the workloads are less skewed (Figures 5c and 5f). Highly skewed workloads see significantly less accesses in the cold region and can hence tolerate a lower $p_0$.

Using exponentially decreasing probabilities for $p_1$ and $p_2$ with a reasonably high $p_0$ exhibits the least sensitivity to workload properties, albeit other configurations may occasionally perform better. DGWO reduces the average read heat latency generally better than all other schemes, while RHGW, RHWO, and RRHD perform similarly well.

**D. Read and write skew overlap**

Real-world workloads are not only likely to be skewed, but the read and write skew also tend not to overlap [24]. Here, we analyze the efficiency of RHS as a function of the overlap of the read and write skew.

The results in Figure 6 have been obtained using a moderate and a highly skewed workload (Zipf 80/20 and Zipf 95/20 for both, reads and writes) and using two different read-write ratios (i.e., 80/20 and 98/2). Our simulations are based on a fully utilized SSD. The read and write workloads therefore operate on the entire LBA space. All curves are plotted as a function of the offset between the read and write skew where the x-axis denotes the offset as a percentage of the total capacity by which the distribution of the writes is shifted relative to the distribution of the reads. We report the relative achieved latency reduction as a percentage of the maximum achievable reduction with optimal placement. Again, the simulations use ORBC. We only show the algorithms exhibiting a significant drop in achieved latency reduction. Almost no latency reduction can be achieved when the read and write workloads completely overlap.

**E. Adaptation speed to workload changes**

An important metric is the speed at which RHS adapts to workload changes. Here, we select the heat tracking strategy $\hat{\beta} = [1, 0.1, 0.01]$ and DGWO combined with ORBC, that performs consistently well and mostly outperforms the other schemes as presented above. In this experiment, all L2P mappings are randomized, all read heat counters are reset, and no RHS is performed initially. We then perform a large amount of reads following a skewed Zipfian distribution.

Next, we issue the same Zipfian-type of workload as writes at an offset that minimizes the overlap with reads executed before. Every 10% of the LBA space written, the same read workload as before is executed and the read latency is measured. The writes are used to trigger GC and the gathered read heat information so far is used in RHS when data are written to new locations. The sequence is repeated until RHS is no longer able to reduce read latency.
Figure 7 shows the average latency reduction relative to optimal data placement as a function of the number of writes. After writing an amount of data that corresponds to 20% of the logical capacity, a significant latency reduction can be achieved when the workload skew is high. In particular, for a Zipf 95/20 workload, 76.5% of the possible reduction is already achieved at this point. Even a lightly skewed workload such as Zipf 70/30 exhibits benefits from RHS at this point, although it only reaches 21.9% latency reduction. Zipf 80/20 and 95/20 reach their maximum latency reduction after an amount of data that corresponds to 30% of the logical capacity has been written and used for RHS while Zipf 70/30 reaches this level at about 40%.

This is important for hybrid controllers with dynamically resizable tiers: As initially all blocks are in SLC, they have time to track sufficient read accesses before blocks are converted to QLC. When the capacity used increases and the SLC cache shrinks, data is destaged to QLC blocks and written to flash pages in accordance with their read heat. Therefore, such a design can offer both, high storage capacity and good read performance. For non-hybrid controllers or controllers with a small cache, the amount of writes to get good RHS is significant such that actively identifying and relocating read hot data currently stored in slow pages is advisable [25].

F. Measurements on a real SSD

We implemented RHS in a commercial enterprise-grade SSD controller with QLC flash using ORBC and measured its efficiency. We preconditioned the drive with sequential and random writes such that 80% of the logical capacity was used. Using two concurrent read workloads, the first one issuing 95% of the reads to 5%, and the second one 5% of the reads to 95% to the written LBA space, we measured the read latency distribution first without RHS. We then added a concurrent uniform random write workload. At this point, all overwrites were placed using the read heat information gathered. We then measured the read latency again.

Figure 8 illustrates the cumulative density functions (CDF) of the two measured read latency distributions. The x-axis is normalized to the average QLC latency without RHS. The four steps visible in the plot are caused by the latency characteristics of the different page types. We clearly see that the levels change with RHS. While the number of the fastest page reads has increased from 24.2% to 44.0%, the number of the slowest Pd page reads has been reduced by 4× to only 5.5% of all reads. Overall, RHS reduced the average latency by 17.1%.

V. RELATED WORK

Gray coding schemes are widely used [8], [9]. In data storage systems, they can be used to reduce the raw bit error rate [26]. Choi et al. [7] introduced invalid data-aware (IDA) coding, where the voltage levels are reprogrammed when the lower bit is invalidated to accelerate future reads. The reprogramming of already programmed cells causes additional cell-to-cell interference which increases the RBER of neighboring pages whereas our approach does not affect the reliability. IDA coding could easily be combined with our heat-aware data placement.
Improving read performance of multi-bit flash based on latency variation of page types was first introduced in Fastread [25]. Data migration is only performed during idle time and only a subset of all pages are being tracked.

Read heat information can be leveraged for other purposes such as mitigating read disturb effects [27] or creating additional data copies to reduce the tail latency for frequently read data [28], [29]. Such techniques are synergistic to our proposal. Given that heat tracking adds a non-negligible amount of complexity, it is beneficial to leverage the read heat information for multiple purposes.

Some existing flash controllers perform write heat separation to reduce the internal write amplification of an SSD [2], [30], [31], [32]. There is a tension between performing write and read heat separation simultaneously. We propose to keep the write heat separation logic unchanged and only perform RHS for data that would be normally written inside a block.

VI. CONCLUSIONS

We show that in multi-bit NAND flash the selection of an appropriate bit encoding scheme enables read latency variations among different physical page types which can be efficiently exploited. This paper evaluated manifold key aspects and trade-offs in the controller architecture that influence RHS. Our RHS exploits the fact that the access patterns of real-world workloads are typically skewed. Our experiments show that close to the optimal read latency can be achieved, without impacting write performance and with limited controller changes. In contrast to other heat tracking schemes, we demonstrate that read heat tracking at a very fine granularity can be implemented in real controllers supporting large capacities of several tens of Terabytes. For reasonably skewed workloads, our design can even outperform previous faster NAND flash generations at the system level.

RHS efficiency improves with increasing workload skew. We show that more than 80% of the potential reduction in average read latency can be achieved when the read and write skews are not overlapping. Measurements on a real SSD with RHS show an average latency reduction of more than 17% on the system level. One key aspect is the efficiency at which the heat tracking scheme can detect never or rarely accessed data locations.

Further, read heat tracking can only start once data has been written. This is a clear drawback for non-hybrid controllers or controllers with a small cache as the data has to be relocated in the background at some point later. Until then, no benefit from RHS is achieved. We therefore strongly suggest the use of a hybrid controller architecture with a dynamically resizeable SLC cache. This also allows for simplifying the SLC GC, decoupling the performance of RHS from the QLC GC policy, and hence offer both, high storage capacity and good read performance.

REFERENCES

A Smart Background Scheduler for Storage Systems

Maher Kachmar  
Northeastern University  
Department of Electrical and Computer Engineering  
Boston, MA  
kachmar.ma@northeastern.edu

David Kaeli  
Northeastern University  
Department of Electrical and Computer Engineering  
Boston, MA  
kaeli@ece.neu.edu

Abstract—In today's enterprise storage systems, supported data services such as snapshot delete or drive rebuild can result in tremendous performance overhead if executed inline along with heavy foreground IO, often leading to missing Service Level Objectives (SLOs). Typical storage system applications such as Virtual Desktop Infrastructure (VDI) or web services follow a repetitive high/low workload pattern that can be learned and forecasted. We propose a priority-based background scheduler that learns this pattern and allows storage systems to maintain peak performance and meet service level objectives (SLOs) while supporting a number of data services. When foreground IO demand intensifies, system resources are dedicated to service foreground IO requests and any background processing that can be deferred are recorded to be processed in future idle cycles as long as our forecaster predicts that the storage pool has remaining capacity. The smart background scheduler adopts a resource partitioning model that allows both foreground and background IO to execute together as long as foreground IOs are not impacted, harnessing any free cycles to clear background debt. Using traces from VDI and web services applications, we show how our technique can outperform a static policy that sets fixed limits on the deferred background debt and reduces SLO violations from 54.6% (when using a fixed background debt watermark), to only 6.2% when dynamically adjusted by our smart background scheduler. 

Index Terms—enterprise storage systems, priority scheduling, time series forecasting, snapshot delete performance, QoS

I. INTRODUCTION

Storage servers and the cloud providers are today more than ever under marketing pressure to meet their service level objectives (SLOs) and maintain peak\(^1\) performance while supporting ever-expanding large datasets. Due to advances in hardware in general, and storage processors in particular, hundreds of cores can be packed in multiple CPU sockets and hundreds of gigabytes of memory are typical of today's storage processors, which are the building blocks for storage clusters [1], [2]. It is expected that storage providers will utilize these resources to scale to larger datasets, providing higher performance, all while supporting services such as data protection, storage efficiency (data compression and deduplication), and data analytics. For instance, in a storage world where thin provisioning and volume snapshots are standard, if the snapshot retention time happens to expire during a burst of foreground IO, or a request to unmap a large address space is issued by a thin application [3], or a disk needs to be rebuilt during a busy period, this can stress the hosting storage system and lead to quality of service (QoS) violations. A recent study by Qiao et al. [4] reports that drive rebuild can have as much as an 78.5% impact on the performance of a ZFS file system.

In addition to the above challenges, storage vendors are adopting log-structured designs to improve write performance by coalescing multiple writes into a larger data block, while deferring garbage collection of the overwritten data to a background process [5], [6]. This log-structured design often utilizes a fast, but expensive, media tier that requires a background process to continuously flush newly written data to slower media. This background process is considerably heavier as compared to foreground IO, especially during core data services such as inline compression and deduplication. Moreover, during bursty load periods, the foreground (FG) I/O load can be further optimized by adopting a fast-path that defers data/metadata updates that need be prefetched and/or modified, to another background process, enabling more IOPS to be squeezed in during said bursts. Storage application background processing includes the following types of background (BG) ops (the list is not exhaustive):

- Snapshot delete, volume delete and unmaps.
- Deep compression and offline deduplication.
- Garbage collection and defragmentation.
- Drive / stripe rebuild.
- Prefetching and relocation of data.
- Data and metadata flush.
- Replication sync, integrity checks, and data analytics.

Except for the last item on the list, which is involved in improving reliability and serviceability, the only drawback of delaying these BG operations is the potential that they can tie up free storage pool space. In a thin provisioning environment (i.e., just-in-time block allocation) [7], any volume may use all of the storage pool space, where a storage pool is made up of all or subset of the disks in a storage system. If we build up too much deferred debt, this can lead to the storage pool running out of space, which is a hard SLO violation, most notably known as a Data Unavailable (DU) event. To address these potential problems motivates the need for a smart background scheduler that is powered by a forecasting model leveraging statistics and machine learning. This allows us to characterize workloads on a per system deployment basis in order to strike a resource partitioning balance between servicing user IO.

\(^1\)Storage server’s peak performance is measured under favorable conditions, while data services are disabled.
requests and providing the required data services.

Our proposed smart background scheduler adopts a resource partitioning model that allows both foreground and background IOs to execute together as long as foreground IOs are not impacted. The scheduler is backed by a forecasting component that can harness any free cycles to clear the BG debt and keeps BG work minimized during heavy IO. Using traces from a Virtual Desktop Infrastructure (VDI) and web services applications, we show how our technique reduces SLO violations from 54.6% when statically deferring BG debt, to only 6.2% when guided by our dynamic BG scheduler.

The rest of the paper is organized as follows. Section 2 discusses related work. Section 3 covers the experimental setup used in the remainder of the paper. Section 4 presents a workload characterization of multiple popular applications hosted on a typical enterprise storage system. In Section 5, we present our forecasting scheme and the BG scheduler models. Section 6 presents results from trace-driven simulations, comparing multiple forecasting models. Finally, in Section 7 we conclude this work and discuss directions for future work.

II. RELATED WORK

There is a rich body of work already established on IO workload characterization and predictions. Many prior studies focused on cache warmup and prefetching [8]. In some cases, these workload characterization studies have been used to guide the scheduling of maintenance or background work [9]–[11], for capacity planning [12], or for improving the performance of bursty workloads [13]. Taylor et al. [14] describe a time-series forecasting method to elegantly capture short-term and long-term seasonality as a curve-fitting exercise, all without over-fitting training data. Alshawabkeh et al. [15] use Markov Chain Correlation of the spatial and temporal characteristics of storage blocks to intelligently place blocks on a tiered storage system. Ravandi et al. [16] model storage providers in the cloud as a blackbox. They are able to assess each vendor’s Quality of Service (QoS) level through a monitoring process. In turn, this information is fed to a machine learning algorithm to classify the level of QoS met, with the goal of reducing SLO violations and balancing storage capacity. Xue et al. [8], [17] use machine learning techniques to predict user workload intensities in order to schedule data analytics work during idle times. Stokely et al. [12] use ensembles of time series models to project disk utilization trends in a cloud setting. Zhang et al. [18] study IO characteristics of storage systems to ensure that reliability-based background jobs do not impact FG IO. Their analytical model incorporates characteristics such as burstiness, arrival dependence and system utilization.

However, we depart from previous work in several aspects. First, we take a more holistic approach and explore solutions that can be adopted by the storage industry. Prior approaches assumed that background operations cannot be executed along with FG IO. They also impose a hard requirement to warm up the cache after executing each workload. With recent advances in hardware, where a datacenter-on-a-chip is no longer science fiction [1], this assumption is no longer applicable. Isolating execution domains between FG and BG allows these different workloads to co-execute in their respective domains. Moreover, prior approaches lack key information needed for the scheduler to perform as an ideal storage cluster, able to meet both performance and capacity guarantees. It may be infeasible to delay BG processing if the storage pool is running out of blocks, or if reliability is compromised due to missing drive rebuild times. To provide a framework for the storage industry where performance and capacity are tightly coupled, we feed this knowledge to our priority-based scheduler where we:

- Quantify the cost of the deferred BG work.
- Track and forecast both pool utilization and deferred BG debt, in order to estimate when resources will be depleted.
- Evaluate current incoming load and harvest any free cycles to execute BG operations using isolation domains to reduce any impact on FG I/O.

For example, deleting hundreds of snapshots is typically CPU intensive, but also deferring large volume deletes or deferring offline deduplication can tie up tremendous amounts of storage, potentially causing a user application to run out of space. Adding this key information to the model helps provide a more practical priority-based scheduler for the storage server so as to balance both FG and BG processing.

III. EXPERIMENTAL SETUP

To guide the development of our smart scheduler, we run synthetic workloads on a live storage server. Our testbed consists of a server running Centos 7 with 2 NUMA Intel Xeon CPUs, 6 cores per CPU socket, 24 GB of memory and 2 SAS2 drives. Each SAS2 drive has a 300GB capacity and runs at 10,500 RPM. We use the FIO tool, version 3.7, to produce a synthetic block-based workload on this testbed [19]. FIO provides the capability to create various workload streams (random, mixed, etc.) with varying compression and deduplication ratios. This tool is initially used to produce workload traces to guide the design of our BG scheduler.

A variety of storage traces when running real enterprise applications exist. SNIA [20] provides VDI and web traces (discussed later in this paper), as well as several other storage traces. Wikipedia traces also exist for public research use [21]. These traces are used to compare forecasting models, which in turn drive the evaluation of our smart BG scheduler. After collecting traces, we designed a trace-driven simulator that:

- pre-processes block I/O traces (extract features such as IO load and load type mix at different time granularities),
- applies BG scheduling policies (protection, efficiency),
- computes BG processing costs for common load types,
- attaches to various BG scheduler models (enabling analysis of different forecasting and scheduler models), and
- generates performance metrics and visualizes results.

Our goal is to compare the performance of a system with and without BG data services supported. Ideally, the BG data services should have little to no performance impact on the user IO during busy phases. The proposed solution should outperform any method that statically limits the deferred BG debt.
IV. WORKLOAD ANALYSIS AND TRACE GENERATION

To better understand the potential performance degradation introduced by BG processing on XFS [22], a popular Linux filesystem, we ran FIO [19] on the testbed described in the previous section to generate a random read workload using 8 jobs (threads). This workload nearly saturates the drives (≥ 90%). We consider here two scenarios. In the first scenario, we ran the test without any BG load. In our second test, we introduce some BG load into the system. We produce BG load by concurrently running deletes of 20 large files. Figure 1 reports the IOPS and resulting response time on the XFS file system. Although the 20 files deleted may have only involved metadata updates in this filesystem implementation, this BG traffic was still enough to produce a measurable impact. We found that IO latency was doubled and the number of IOPS were cut in half. Given the popularity of thin provisioning storage and the fast snapshot technology utilizing a copy-on-write technique [23], the primary production volume and its snapshot share many data blocks. Thus, under load, the performance impact grows significantly in the case where the BG process is a snapshot delete of a volume that shares blocks between the deleted snapshot and the production volume.

Many storage workloads tend to be bursty in nature, follow repeatable patterns (day/night or weekday/weekend), and are tightly coupled to the number of time zones in which storage clusters are deployed. Figure 2 shows the IO arrival rate for a production storage system [8]. There are clear patterns present, based on day/night and weekday/weekend intervals. Figure 3 shows the IO arrival intensity of a web workload. This trace represents FIU’s CS Department webmail proxy and online course management over a 3-week period starting on 11/01/2008. Figure 4 shows the IO arrival intensity of a Virtual Desktop Infrastructure (VDI) workload run on 3 LUNs over a 10-day period. We include a zoomed-in snapshot of the first 24-hour period. These traces can be obtained from SNIA [20]. Even at a finer granularity (minute/hour), clearly, there is repeatable pattern present.

Autocorrelation is the degree of similarity between a time series and a lagged (i.e., time delayed) version of the time series over successive intervals. The autocorrelation of lag k is defined as the correlation between values that are k time periods apart. Figure 5 shows the autocorrelation of the incoming load intensity at increasing lag periods. The auto-correlation is strongest at a 24-hour lag for the VDI workload (datapoint 144 at the 10-minute interval). Similarly, the auto-correlation is strongest at a 24-hour lag for the web workload (datapoint 24 at an 1-hour interval). We chose a larger granularity for the web workload to reduce load burstiness. Figure 6 shows the 24-hour lag spread, which indicates a strong 24-hour correlation. The tighter the clustering, the more data points are correlated. All in all, typical storage workloads follow repeatable patterns.

Given the strong correlations observed at hourly, daily and weekly lags, the IO arrival and block write rates can be analyzed using a time series forecasting model. The forecast produced from the time series model is fed to the BG scheduler to populate an hourly, daily, weekly, and even monthly plan of how system resources (i.e., CPU and memory resources) are allocated. This plan will guide the scheduler on how much BG debt should be deferred (the size of the BG debt buckets) and indicate when the debt will be repaid (i.e., the time when we can start processing this debt).

The input features of our forecasting algorithm include:
The IO arrival rate, which can be specified for a range of granularities (one may favor a specific forecasting model),
• the load mix (breakdown of read/write/unmap ops),
• the free storage capacity, deferred BG debt (estimated in block size units), and a per-queue processing rate.

The smart BG scheduler predicts the idle cycles, remaining processing capacity, storage capacity and deferred debt. The scheduler leverages the resulting debt processing plan and sets the size of the deferred-debt bucket accordingly. The deferred-debt bucket size continuously grows and shrinks according to the debt processing plan. Growing the debt bucket allows the system to defer more BG work and reduce the impact of BG work on FG IOs during peak times. The scheduler will ultimately produces hourly, daily, and weekly forecast plans, identifying potential windows when the debt can be paid off. Each window includes a size, indicating how much debt can be paid off. The scheduler decides which debt to pay first, based on how much space can be reclaimed, and updates a weekly debt balance sheet to keep debt in check.

V. METHODOLOGY

A. Forecasting Model

The IO arrival intensity includes many parameters of a time series model, as discussed earlier. The series of incoming IO counts and the read-write mixes observed during sequential time windows becomes a time-ordered series. We consider prior work on time series forecasting [8], [14], [24]–[26], comparing some of the most promising methods. We enhance these models by adding new dimensions that are specific to our need to forecast the read-write mix, free pool space, and the cost to clear BG debt. BG operations such as data/metadata flushes require finer-grain time models (seconds/minute granularity). In particular, we explore and compare the following models, which are ordered based on complexity. We expect the more computationally complex a model, the more accurate the model should be and the more states it can identify. The models considered include:

• A time series model based on ARIMA(p,d,q) (Auto-Regressive Integrated Moving-Average) [24] - we calculate autocorrelation function (ACF) and partial autocorrelation function (PACF) to choose parameters p and q, which correspond to the lag size and the moving-average window size, while d becomes a factor if the trend is not stationary.
• A time-series model based on Triple Exponential Smoothing using the Holt-winters’ method [25].
• Machine learning techniques, in particular, Long-Term-Short-Term-Memory (LSTM) - this non-linear model is explored when a finer prediction model is needed.

In addition, we propose a forecasting model that is similar to Triple Exponential Smoothing. This method initially decomposes the time-series into trend, season, and noise where:

\[ Y_t = Trend_t + Season_t + Noise_t \]

An Exponential Smoothing forecasting model does not support multiple seasonality periods. Trend indices may be based on weekly cycle patterns. However, seasonal indices usually have a daily cycle (see Section 4). In order to predict a future day’s IO intensity or read-write mix, we use an Exponentially Weighted Moving Average (EWMA) model to capture the trend on a weekly cycle and season on a daily cycle. EWMA for a series \( Y \) is defined as a recursive function:

\[
S_t = \begin{cases} 
Y_1, & t = 1 \\
\alpha Y_t + (1 - \alpha) S_{t-1}, & t > 1 
\end{cases}
\]

where \( \alpha \) represents the degree of smoothing (ranging from 0 to 1), \( Y_t \) is the value at a time period \( t \), and \( S_t \) is the value of the EWMA at any time period \( t \). The weekly forecast is simply the sum of the trend index and the seasonal index (each based on its own time interval):

\[
\hat{Y}_t = EWMA_{Trend} + EWMA_{Season},
\]

Even though the prediction error might still be high, the Mean Percentage Error (MPE) is expected to be low. The white noise effect of over-prediction (positive error) and under-prediction (negative error) will offset each other. The cumulative prediction error at time \( t \) is computed as follows:

\[
\sum_{0}^{t} (Y_t - \hat{Y}_t) = \epsilon
\]

We are not concerned with minimizing the error between the predicted and actual value (i.e., \( Y_t - \hat{Y}_t \)), as long as the cumulative error \( \epsilon \) at any time \( t \) is small. If the prediction error causes us to burn through debt faster than required at any time \( t \), then the next cycle \( (t + 1) \) can lead to burning less debt, which will ultimately cancel out prediction errors.

![Fig. 5. Autocorrelation associated with different lags of the VDI (left) and web block (right) IO workload.](image1)

![Fig. 6. 24-hr lag spread of the VDI and web workloads respectively.](image2)
Forecasting the IO arrival rate, the read-write IO mix, and factoring in multiple data services policies helps us estimate the free pool space and the deferred background debt for a future scheduling period. Algorithm 1 shows the forecasting algorithm system flow.

**Algorithm 1:** Forecasting algorithm flow.

### B. Background Scheduler Model

As explained earlier, the scheduler needs the following data to make the right scheduling decisions. These parameters will become the input features to the learning algorithm:

- to forecast the IO arrival intensity, and write intensity,
- to compute the current free capacity and estimate the deferred free capacity, based on BG debt and adopted data services policies, and
- to determine the processing cost to retire the BG debt.

The learning algorithm will perform the following operations for the BG scheduler:

- partition the resources between FG and BG processing,
- identify the BG queues to prioritize, and
- develop a future plan of when and how to reduce BG debt and set the size of the various BG buckets, accordingly.

The BG scheduler may have to execute BG load along with FG I/O if the forecast predicts a high intensity load for the upcoming periods. In order to avoid cache thrashing and other performance impacts, we tend to execute BG jobs in an isolated domain using a separate CPU in multi-CPU storage processor, or a subset of cores from a single CPU storage processor. Partitioning the resources and segregating FG and BG workload allows FG requests to execute faster and without the need to warm up the caches. This is an advantage of our priority-based scheduler. Thus, hardware can be modeled as a tuple of < sockets, cores, memory >. Each core is confined to either a FG or a BG domain at any given time t. However, for simplicity, we exclude the memory cache from the isolated domain as a partitioned resource for the following reasons:

- Storage server memory is merely a big read cache.
- We assume that the cache eviction rate is low due to BG processing, since we are caching metadata.
- Metadata access is a tiny fraction (2-5%) compared to data. We assume that all metadata will fit in cache.
- A good prefetching algorithm is already deployed.

Figure 7 provides an example of how to execute the isolated domains according to the scheduler’s forecasted plan. FG I/O latency typically demands low latency as compared to other operations. Requiring FG I/O to wait behind a heavy BG thread can impact FG latency significantly. Thus, the importance of partitioning the resources and segregating cores used to service FG versus BG is a key part of this design.

The FG IO load is a combination of both read and write loads. The space capacity utilization is a cumulative function of previous time periods:

\[
L_{FG}(t) = L_{Read}(t) + L_{Write}(t)
\]

\[
L_{Write}(t) = (1 - r(t)) \times L_{FG}(t)
\]

\[
U(t+1) = U(t) + u(t) \times L_{Write}(t) + D(t)
\]

where \( L_{FG}(t) \) is the FG arrival IOPS and \( r(t) \) is the read ratio at a given time \( t \), \( U(t+1) \) is the capacity utilization at time \( t + 1 \), and \( u \) is the ratio of unique blocks (i.e., blocks that are not overwritten), and \( D(t) \) is the number of deferred debt blocks at time \( t \).

Each of the deferred debt types can be derived from the FG arrival rate using a unique formula for each type:

\[
D_{Overwrite}(t) = (1 - u(t)) \times L_{Write}(t)
\]

\[
D_{Unmap}(t) = \frac{Len(t)}{BSize} \times (1 + DMD\_Ratio)
\]

\[
D_{Snap\_Delete}(t) = L_{Write}(t) \times Snap\_Retention\_Time
\]

The forecaster is used to predict the FG arrival intensity rate \( L_{FG}(t) \) and unique block fraction \( u(t) \) of FG writes with a different granularity to cover both the short-term (minute/hour)
and long-term (day/week) goals. These predictions are then used to drive the future debt \( D(t) \) to be accumulated against required data services. For example, if our policy creates a snapshot every hour for a storage volume with a retention time of 1 hour, then the debt accumulated is equivalent to:

\[
D_{\text{Snap Delete}} = \sum_{t}^{t+3600} L_{\text{write}}(t)
\]

The time to service one item in this debt queue is directly proportional to the number of FG writes:

\[
RT_{\text{Snap Delete}} = c \times D_{\text{Snap Delete}}
\]

where \( c \) is the cost to read and manipulate one block.

Given that we modeled the cost to service the deferred debt, the deferred free pool space accrued in the various BG debt queues, and the future forecast and future debt, we use a multi-server queuing (M/M/c) model to allocate the cores to FG and BG workloads. However, one optimization we use is to suppress and segregate BG processing, and bias FG work as long as capacity is not violated in the near future. The priority scheduler assigns priority on each of the BG queues based on the expected reclaimed capacity of this queue, data service/reliability SLO (such as the time required to rebuild a drive), and the time to service one item off this queue.

Each core can deliver a throughput of \( CIOPS_{FG} \) (per-core IOPS), experiencing a latency of \( RT_{FG} \). This number can be quickly obtained when evaluating a new storage server platform, or measured on a live system using a calibration period where data services are paused. If the total system throughput (per forecast) for the next period is \( IOPS_a \), then:

\[
C_{FG}(t) = \text{MIN}\{N, \frac{IOPS_a}{CIOPS_{FG}} - CFF(t)\}
\]

\[
C_{BG}(t) = N - C_{FG}(t)
\]

where \( C_{FG}(t) \) and \( C_{BG}(t) \) are the number of cores allocated for FG and BG, respectively, for the next period. \( N \) is the total number of cores dedicated for use by the storage datapath.

\( CFF(t) \) is the capacity forecast factor. It is the number of cores we need to borrow from the core pool required to service FG IOs in order to guarantee that the storage pool never runs out of free space before the forecasted “idle phase”. As described earlier, the amount of BG work deferred, such as garbage collection of overwritten blocks and snapshot deletes, is directly proportional to FG load. Thus, processing less FG IO reduces the amount of deferred BG work. This factor \( CFF(t) \) certainly impacts FG IOs, and will ultimately lead to deferring less BG debt by these FG IOs, as well as processing more exiting deferred BG debt. This is a sign of an oversubscribed system where that we treat as a violation. If said violations increase dramatically over time, the operator is notified to add a new disk to the pool or reduce/rebalance application load. Algorithm 2 shows the end-to-end smart BG scheduler system flow.

| Data: Forecasted arrival intensity and read/write mix. |
| Result: Find the share of BG cores needed for each of the forecasted periods at specified granularity. |
| 1 if entering Calibration phase then |
| 2 Determine FG IOPS rate per core at a configurable response time (RT); |
| 3 Determine for each BG type the per-core processing rate; |
| 4 else |
| 5 while not at the end of forecasted period do |
| 6 Determine BG load generated based on incoming load and read/write mix; |
| 7 Determine BG load processed (i.e., \( C_{BG}(t) \)) and deferred based on incoming load; |
| 8 if deferred BG debt exceeds the remaining free pool capacity during any cycle \( t \) then |
| 9 Determine how many cores to borrow from FG, while not exhausting free pool capacity (i.e., \( CFF(t) \)); |
| 10 spread \( CFF(t) \) across several cycles prior to encountering the oversubscribed cycle \( t \); |
| end |
| 11 end |
| 12 Compute \( C_{BG}(t) + CFF(t) \) to determine the share of BG cores needed for each forecasted period; |
| 13 end |

Algorithm 2: End-to-end smart BG scheduler flow.

VI. RESULTS AND DISCUSSION

A. Workload characterization and forecasting

A time series can be decomposed into three components: 1) trend, 2) season, and 3) noise. The decomposition can be additive, where we add the three components together, or multiplicative where the terms are combined through weights [24]. Figure 8 shows the autocorrelation function (ACF), the partial autocorrelation function (PACF), trend, and season for the VDI trace IO intensity. There is a strong correlation at 24-hr lag. However, the IO intensity is not always stationary due to multiple seasonality effects, such as weekends and holidays. The time series additive decomposition of the VDI trace IO intensity, as shown in Figure 8, shows a stationary trend for Wednesday through Friday, a lower intensity on the weekend, and a higher intensity on Monday. Thus, we suggest to forecast each day in the week separately, or group multiple days based on trends, using a method like K-means clustering.

Initially, we used the standard ARIMA model, trained on a cluster of days that exhibit a stationary trend, where the autoregressive factor \( p \) was set to 144 (24-hrs @ a 10-min granularity) for the VDI trace and 24 (@ a 1-hr granularity) for the web trace, the differential factor \( d \) set to 0, and the moving average factor \( q \) set to 2. Figure 9 shows our forecast versus the actual of IO arrival intensity and the write intensity using the ARIMA model for both the VDI and web
Fig. 8. Auto-correlation function (ACF), partial auto-correlation function (PACF), trend, and season of the VDI workload IO arrival intensity.

Fig. 9. Prediction by ARIMA model for VDI (left) and web (right) workloads.

Fig. 10. Prediction by Triple Exponential Smoothing model for VDI (left) and web (right) workloads, respectively.

Fig. 11. Prediction by our model for VDI (left) and web (right) workloads.

Fig. 12. Performance of the BG Scheduler using a fixed debt bucket size.

Figures 8 to 12 provide visual representations of the data analysis discussed in the text.

taces. ARIMA achieves a weighted mean absolute percentage error (WMAPE) of 27.8% for VDI and 46.1% for web, when comparing prediction to actual.

In the next experiment, we applied the triple exponential smoothing model. Similar to ARIMA, the season period is set to 144 for VDI trace and 24 for web trace, but with a damped trend option. Figure 10 shows the forecast versus actual for the IO arrival intensity using triple exponential smoothing. The WMAPE achieved for triple exponential smoothing is 19.2% for the VDI trace and 32.8% for the web trace.

Finally, we experimented with our simple, but robust, forecasting model. Figure 11 shows the forecast versus the actual for the IO arrival intensity and write intensity using our method. We achieve a WMAPE of 14.8% prediction as compared to the actual for the VDI trace and a WMAPE of 26.2% for the web trace, which means we cut the error range by half as compared to ARIMA, and improve by a few percent versus triple exponential smoothing.

Even though the prediction error is still over 10%, MPE is well below 10%, as the white noise effect of over-prediction (positive error) and under-prediction (negative error) offset each other, as discussed earlier.

B. Smart Background Scheduler

Next, we compare two different implementations of our BG scheduler. Both implementations include replaying the VDI traces of 3 LUNs for the 10-day period, with a snapshot creation/retention time of 1-hr. We fix the number of cores in the simulation to 64, the per core FG processing rate to 50 IOPS, the BG processing rate to 20 ops/sec, and use a 4K block size. The first implementation used a fixed debt bucket size. We fix it to a value between 40 and 50% of the total pool size, where the scheduler becomes more aggressive as we reach 50%. Such an implementation leads to a performance SLO violation of 54.6%, where 21.7% of the IO was queued due to reaching an "out of resources" state, as shown in Figure 12. We record a single SLO violation for every user IO request the server is not able to meet in a fixed time window.

The second implementation uses a dynamic bucket size that utilizes the forecasted information described in Section 5. The debt bucket implementation can grow up to 95% of the total pool size, as long as the forecasted data allows. The smart
scheduler algorithm utilizes the forecasted data (i.e., FG IO and IO mix) for the upcoming period (e.g., a week) to extract information such as free pool capacity and debt size for the upcoming period. The dynamic bucket algorithm maximizes FG IO, as long there is free pool capacity. If a future period experiences that pool capacity is depleted, a greedy algorithm evenly spreads enough BG load during a prior period. This is necessary in order to avoid running out of pool space in the future when maximizing FG IO. With this scheme, we are able to minimize SLO performance violations to only 6.2% through the 1-week period, whereas 2.6% of the IO was queued due to “out of resources” status, as shown in Figure 13.

VII. CONCLUSION AND FUTURE WORK

In this paper, we have proposed a smart BG scheduler for storage systems that can harvest every ounce of processing capacity to absorb application bursts and meet SLOs. The design requires two key elements that work collaboratively to achieve better performance. The first module forecasts information such as IO arrival intensity, data blocks written, and unique data blocks written. The second module uses the forecasted data in the upcoming days to partition the resources between FG and BG operations, favoring FG IO as long as pool capacity will not be depleted. Through replaying VDI workload traces, the smart BG scheduler reduced SLO violations from 54.6% when using a fixed debt bucket to merely 6.2%.

While these are encouraging results, we plan to pursue the following directions to improve upon this work: i) compare our forecasting methods to recurrent neural network method such as LSTMs and ensembles of time series methods [27], ii) use a finer time granularity (second/minute) for BG operations, such as data/metadata flushing, and iii) consider additional BG operations such as efficiency or reliability tasks.

REFERENCES


Abstract—Achieving fast, scalable, and cost-effective genome analytics is always important to open up a new frontier in biomedical and life science. Genome Analysis Toolkit (GATK), an industry-standard genome analysis tool, improves its scalability and performance by leveraging Spark and HDFS. Spark with HDFS has been a leading analytics platform in the past few years; however, the system cannot exploit full advantage of cloud elasticity in a recent modern cloud. In this paper, we investigate performance characteristics of GATK using Spark with HDFS and identify scalability issues. Based on a quantitative analysis, we introduce a new approach to utilize Cloud Object Storage (COS) in GATK instead of HDFS, which can help decoupling compute and storage. We demonstrate how this approach can contribute to the improvement of the entire pipeline performance and cost saving. As a result, we demonstrate GATK with IBM COS can achieve up to 28% faster than GATK with HDFS. We also show that this approach can achieve up to 67% cost saving in total, which includes the time for data loading and whole pipeline analysis.

I. INTRODUCTION

As an increase of rapid development of Next Generation Sequencing (NGS) technologies, genome analysis has become an emerging research area in bioinformatics. Thus, this new research capability attracts a lot of interest from bio-scientists who want to perform genome analysis such as Single Nucleotide Polymorphism (SNP) genotyping, genetic variants identification, and so on. While reducing genome sequencing cost with the NGS tools, a huge amount of genome sequencing data has been produced day by day, usually in the range of 100GB. Therefore, it is always important to prepare a system that can handle the immense size of data as effectively as possible because genome analytics requires huge amount of compute and storage resource.

Genome Analysis Toolkit (GATK) [1] is the most popular and widely used open source genome analytics framework developed by Broad Institute. GATK also provides a typical genome variant discovery analysis workflow as a GATK Best practice [2], which combines multiple tasks into a single pipeline, so that various genome analytics tools (BWA and HaplotypeCaller etc.) are integrated with GATK to ensure a genome analysis ecosystem. Recently, GATK has leveraged Spark [3] in order to achieve higher scalability in their analysis. As a result, GATK pipeline can accelerate genome analysis throughput more easily as a whole by taking full advantage of the capability of node-level and core-level parallelism in each tool.

Meanwhile, those genome analysis tools running on Spark assume to utilize Hadoop Distributed File System (HDFS) [4] as an underlying data storage implicitly. It is a natural architectural design, because Spark is a successor to Hadoop and HDFS can effectively manage large data over multiple nodes in a fault-tolerant way. Besides that, the data locality in HDFS helps to co-locate compute and storage, which can exploit those resources as much as possible [5].

Cloud has been widely used in the last decade. Nowadays, several famous cloud platforms such as AWS, Azure, and IBM Cloud have become essential infrastructure to establish large scale system, services, and applications in a scalable and cost-effective way [6]. Cloud brings various benefits such as availability and elasticity to our applications, and those are effective even if the applications do not adopt cloud native principles. In such situation, it is reasonable to move genome analytics platforms from on-premise to clouds in terms of cost efficiency and scalability. However, the current reference architecture of GATK with Spark and HDFS is not ready to adopt recent modern cloud technologies due to the limitation of the analytics system that tightly couples compute and storage.

Thus, there are several challenges to take full advantages of cloud scalability in GATK pipeline. First, it is quite hard to dynamically add or remove storage resource capacity and nodes in HDFS. HDFS balances data pieces across all nodes and then maintains entire file system consistency. This limitation forces us to keep a fixed number of nodes to store dataset even if genome analysis does not require vast storage space. In addition, we must load the entire dataset to HDFS each time we want to adjust an adequate storage resource capacity to the analysis pipeline. This kind of additional data loading overhead also occurs when we launch Spark/HDFS cluster on a cloud from scratch. Second, similar to the concern for the storage elasticity, compute resource elasticity is also crucial for further optimization to utilize resources more efficiently in each stage of the analysis pipeline. Dynamic compute resource allocation based on the analysis demands could help to reduce the waste of unused resources. Third, workload characterization is most important to know the capability of

Investigating Genome Analysis Pipeline Performance on GATK with Cloud Object Storage

Tatsuhiro Chiba
IBM Research
19-21, Nihonbashi Hakozaki-cho
Chuo-ku, Tokyo, 103-8510, Japan
tyos@jp.ibm.com

Takeshi Yoshimura
IBM Research
19-21, Nihonbashi Hakozaki-cho
Chuo-ku, Tokyo, 103-8510, Japan
chiba@jp.ibm.com
optimization and also helps us to achieve both storage and compute elasticity in GATK pipeline.

To address these challenges, in this paper, we investigate the performance and workload characteristics of GATK-Spark with HDFS (GATK with HDFS, in short) on a cloud environment to identify potential bottleneck and optimization opportunities for the genome analysis pipeline first. Based on the detailed analysis, we enable Cloud Object Storage (COS) as a replacement for HDFS, which can help to bring elasticity to GATK by decoupling compute and storage. With leveraging Spark with COS in GATK, we provide a new best practice for GATK which introduces storage elasticity into the genome analysis pipeline. We also implement several features in GATK to support object storage access, which can bridge an architectural gap due to the difference between object storage and HDFS. Then, we demonstrate that our approach can contribute to not only performance scalability but also saving cost on the entire pipeline execution while comparing performance of GATK with COS and with HDFS. In a typical variant calling pipeline with whole genome sequencing dataset, we demonstrate that COS achieves a 9% performance improvement than HDFS when attaching a low throughput volume to HDFS, and is 20% worse than HDFS when attaching a high throughput volume to HDFS. We also found a fundamental overhead why the performance drawback exists when utilizing COS instead of HDFS in GATK. By eliminating the performance inhibitor, we finally show that COS achieves a 28% performance improvement or completely same performance than HDFS when attaching slower or faster volumes respectively. As for the cost perspective, we indicate that COS always has good cost performance with HDFS while executing an entire analysis pipeline. The main contributions of this paper are as follows.

- We characterize a typical genome analysis pipeline on GATK with HDFS, and identify the scalability and elasticity issues in the pipeline.
- We provide a new best practice by enabling COS instead of HDFS, showing that COS scalability and its cost effectiveness, and then explaining the fundamental cause of the overhead introduced by the new best practice utilizing COS.
- We demonstrate how we can overcome the overhead, increasing the entire pipeline performance up to 28% and decreasing the overall costs up to 67%.

II. BACKGROUND AND RELATED WORKS

A. Genome Analysis Pipelines

GATK defines and provides a typical set of DNA sequence analysis pipeline as GATK Best Practice [2]. Variant discovery, which identifies genome variants in the DNA, is widely used genomic analysis. Figure 1 explains an overall pipeline defined in GATK, consisting of multiple steps. First step is a data preprocessing to align vast sequences into a reference genome and create a DNA mapping for the further analysis. There exist several fast alignment tools, but Borrows-Wheeler Aligner (BWA) is a popular tool based on Borrows-Wheeler Transform (BWT) algorithm. GATK implements BWA-MEM in a first step of the pipeline for this purpose. In this paper, we skipped this step and used a BAM format as an input file. BAM format is compressed binary to represent aligned sequences.

Next step is MarkDuplicates phase, which marks duplicated fragment of sequences by utilizing reference genome data and alignment information. This steps reads a vast amount of genome data while comparing a large number of key value pairs. Then, the third step is Base Quality Core Recalibration (BQSR) phase, which adjusts quality scores for the aligned read dataset by applying a machine learning model to correct systematic technical sequencing machine errors.

After finishing these steps, variant discovery phase, named Haplotype Caller, starts processing as a final step. This phase searches all of the genome variants by comparing well known reference genome variants. To speed up this sequence comparison phase, Haplotype Caller implements PairHMM Forward algorithm, and several accelerator implementations are available to take full advantage of underlying CPU features, such as native SIMD function support and OpenMP multi-threading support. The last stage in this step materializes a final output, VCF file, which contains all of the discovered genome variants with some headers.

B. Analytics Engine and Storage on Cloud

Apache Spark [3] is one of the most widely used analytics engines especially for big data processing. With applying in-memory computing style, Spark has extremely higher throughput than predecessor engines such as Hadoop. Since HDFS [4] has been a primary data lake in on-premise analytics environments in the past decade, the combination of Spark and HDFS still remains as a first choice even if we move into a cloud, because this architecture is stable and does not require additional learning cost. Spark itself is flexible in a backend storage for reading/writing data, so it can work well in not only HDFS, but also with other types of storage such as RDB, key-value store, and object storage.

Cloud object storage such as AWS S3, IBM Cloud Object Storage (COS), and Google Cloud Storage (GCS) provides high capacity, reliable, and cost effective managed service to users and applications. Cloud object storage was basically suitable for the purpose of storing large amount of data, so it was not supposed to be a replacement file system for applications, because disk bandwidth was always higher than network capacity, and also disk access latency is lower than network latency. By enabling high network capacity in a
cloud, however, instances and services within a cloud can achieve higher throughput than ever before. This capability makes us rethink system architectural design regarding how much performance impact we can achieve in analytics platform [7]. In terms of storage scalability, object storage will be an essential piece of modern cloud which can help to decouple compute and storage while maintaining or improving performance. Of course, object storage is not a POSIX file system, but a storage service that can be accessible through RESTful API; therefore, applications must take into consideration the difference between object storage and file storage.

C. Related Works

A couple of earlier works have been studied before to address scalability problems and workload characterization of genome analysis pipeline. SparkGA [8] is an Apache Spark based framework for a DNA analysis pipeline, which introduces an optimal parallel implementation of the analysis with input load balancing and maximizing resource usage. A successor of the work, SparkGA2 [9] improves the efficiency of data access on HDFS and reduces memory footprint with an optimized compression method for intermediate data. CloudGT [10] proposes a Parquet columnar based optimization to improve IO performance in several genome analysis pipelines using Spark. Costa et al. [11] investigate performance characteristics of genome analysis pipeline on GATK using Spark, and introduce JVM and Spark configuration level tuning to accelerate performance. Doppio [12] proposes a disk IO aware Spark analytical model to estimate Spark application behavior. It utilizes GATK analysis pipeline as a representative workload of this work. However, all of the previous works assume HDFS as an underlying primary storage.

Several works have investigated how big data analytics platforms can utilize object store for their workloads. SwiftAnalytics [13] provides a system that considers data access locality with placement control to leverage OpenStack Swift object storage system for Spark. To easily and transparently access data in object store from Spark application, many storage connectors are available in Spark, such as AWS S3A connector, IBM COS connector (i.e. Stocator [14]), OpenStack Swift Connector, and so on. Since they ensure their connectors over Hadoop File System APIs, application can transparently access objects without any modifications. However, Hadoop File System APIs are originally designed for HDFS, so some APIs do not work well with object storage.

III. PERFORMANCE ANALYSIS AT SCALE

In this section, we first evaluate genome analysis pipeline performance on GATK4 using Spark with HDFS (GATK with HDFS, in short) to get a better understanding of the issues on a typical Spark Hadoop analytics framework in a cloud. We also reveal its performance characteristics and scalability with detailed Spark application and system metrics. Based on the results, we identify what challenges are remaining when we run genome analysis pipeline to leverage modern cloud scalability as much as possible.

A. Experiment Settings

Cloud Environment: We set up a Spark and Hadoop cluster on IBM Cloud Virtual Private Cloud (VPC) environment. According to the cluster virtual server configuration, we launched up to twelve mx2-32x256 instances on the same availability zone, each equipped with 32 vCPU cores and 256 GB RAM. We choose Ubuntu 18.04.1 LTS (kernel: 4.15.0-42-generic) for the operating system. These instances are connected with up to 16 Gbps network. We also attached two 1-TB block storages with different IOPS profiles for each node: 3-IOPS/GB and 10-IOPS/GB. As for the storage profile, we can select several storage IOPS profiles based on our workload requirements. Actual storage performance with storage IOPS profiles depend on volume size and profile limitation, so that a 1-TB volume with 3-IOPS/GB has up to 3,000 IOPS (3K-IOPS) and a 1-TB volume with 10-IOPS/GB has up to 20,000 IOPS (20K-IOPS) respectively. We formatted the volumes with XFS. These disks are utilized by not only HDFS, but also Spark executors, because Spark requires local temporary storage to store intermediate data for shuffling over nodes. Virtual server settings and specifications are summarized in Table I.

Software Stack: We installed GATK 4.1.4.1, Spark 2.4.5, Hadoop 2.7.7, and OpenJ9 JVM 1.8.0_242-b08 on the cluster. Software configuration tuning is always crucial to leverage application performance as much as possible. In terms of Spark, Hadoop, and underlying JVM parameter tuning, several works have already studied best configuration practice [15][11]. Configuration tuning itself is out-of-scope in this paper, so we borrowed knowledge on tuning from references. As discussed in those papers, we managed multiple Spark executors in each node; we launched four executors with 8 vCPUs, 35 GB heap, and 15 GB off-heap in each. Thus, almost all compute and memory resources are reserved by Spark executors. We applied default configurations to HDFS, like three replicas, 128 MB block size, and so on.

Genome Dataset and GATK Pipeline with Spark: We prepare an open whole genome (WGS) dataset obtained from GATK, originally coming from the 1,000 genome project [16]. We used WGS-G94982-NA12878-no-NC_007605.bam as input, which has 154 GB in total. GATK also provides a reference benchmarking pipeline from aligned reads to variant calling for Spark, named ReadsPipelineSpark, in their repository. Based on the reference runner scripts for the WGS dataset, we upgraded pairHMM option to AVX_LOGLESS_CACHING_OMP to accelerate Haplotype...
Table I

<table>
<thead>
<tr>
<th>Region</th>
<th>Nodes</th>
<th>OS</th>
<th>Profile</th>
<th>vCPUs</th>
<th>Memory</th>
<th>Network</th>
<th>Block Storage (IOPS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>London (eu-gb)</td>
<td>12</td>
<td>Ubuntu 18.04.1</td>
<td>mx2-32x256</td>
<td>32</td>
<td>256 GB</td>
<td>16 Gbps</td>
<td>1 TB (3K-IOPS) and 1 TB (20K-IOPS)</td>
</tr>
</tbody>
</table>

![Fig. 3. Weak Scaling Performance on GATK with HDFS](image)

Following three jobs with shuffling. The fifth job represents BQSR in the pipeline. It performs a significant amount of shuffle read and write in the computation. The last three jobs represent HaplotypeCaller, which finally writes variant result into HDFS with many data shuffles. (a) and (b) in Figure 4 show a breakdown analysis of how much time is spent in each job. Both represent the result on two and twelve nodes with 3K-IOPS and 20K-IOPS disk. We can observe several characteristics with the breakdown analysis. First, disk IO performance of jobs 0 and 1 are dominant. Although job 1 also reads the same data that job 0 has read, job 1 benefits from Spark in-memory architecture; job 1 finishes quite faster than job 0 because data is cached in memory as a file cache. Moreover, job 7 is dominant in all of the entire pipelines with the increase of nodes.

As for (c) and (d) in Figure 4, they describe how much each job scales when increasing nodes and cores. We can observe here that the scaling characteristics are also different in each job and analytics pipeline stage. Regarding the first two jobs related to MarkDuplicate, their scalabilities are bounded by performance of disk IO and memory respectively. The following two jobs, jobs 2 and 3 have good scaling because they are network and memory intensive. Job 4, BQSR in a pipeline, has also great scalability; it achieved 10x and 6x scaling with 3K and 20K IOPS HDFS respectively. Job 7, which is the last stage in HaplotypeCaller, accounts for half of the time in the pipeline when relaxing a disk performance constraint with higher throughput disk.

Next we perform breakdown analysis on the GATK pipeline. Table II describes characteristics of each Spark job, including input/output data size, shuffle read/write size, and the category this job corresponds to in GATK pipeline. Although it depends on the version of GATK and Spark, the version of GATK we used in this paper translates an entire pipeline shown in the Figure 1 into eight spark jobs. Each job has different characteristics: read IO heavy, network heavy, shuffle read/write heavy, and so on. The first four jobs represent MarkDuplicate with Aligned Reads. WGS genome data is loaded from HDFS in the first job, then the data is consumed in the

![Fig. 2. Overall Architecture of GATK Benchmarking Environment](image)

First we evaluate scalability of the ReadSparkPipeline workload while changing the number of nodes and total cores. Figure 3 shows a weak scaling result with two to twelve nodes. As shown in Figure 3, GATK ReadsSparkPipeline has good scalability and achieved around 5.5x speedup in twelve nodes compared to the performance in two nodes as a whole. Focusing on the result in twelve nodes that have 384 vCPUs and 3TB RAM in total, we finally achieved 68 minutes and 46 minutes to complete a full analytics pipeline with 3K-IOPS and 20K-IOPS disk respectively.

Next we perform breakdown analysis on the GATK pipeline. Table II describes characteristics of each Spark job, including input/output data size, shuffle read/write size, and the category this job corresponds to in GATK pipeline. Although it depends on the version of GATK and Spark, the version of GATK we used in this paper translates an entire pipeline shown in the Figure 1 into eight spark jobs. Each job has different characteristics: read IO heavy, network heavy, shuffle read/write heavy, and so on. The first four jobs represent MarkDuplicate with Aligned Reads. WGS genome data is loaded from HDFS in the first job, then the data is consumed in the following three jobs with shuffling. The fifth job represents BQSR in the pipeline. It performs a significant amount of shuffle read and write in the computation. The last three jobs represent HaplotypeCaller, which finally writes variant result into HDFS with many data shuffles. (a) and (b) in Figure 4 show a breakdown analysis of how much time is spent in each job. Both represent the result on two and twelve nodes with 3K-IOPS and 20K-IOPS disk. We can observe several characteristics with the breakdown analysis. First, disk IO performance of jobs 0 and 1 are dominant. Although job 1 also reads the same data that job 0 has read, job 1 benefits from Spark in-memory architecture; job 1 finishes quite faster than job 0 because data is cached in memory as a file cache. Moreover, job 7 is dominant in all of the entire pipelines with the increase of nodes.

As for (c) and (d) in Figure 4, they describe how much each job scales when increasing nodes and cores. We can observe here that the scaling characteristics are also different in each job and analytics pipeline stage. Regarding the first two jobs related to MarkDuplicate, their scalabilities are bounded by performance of disk IO and memory respectively. The following two jobs, jobs 2 and 3 have good scaling because they are network and memory intensive. Job 4, BQSR in a pipeline, has also great scalability; it achieved 10x and 6x scaling with 3K and 20K IOPS HDFS respectively. Job 7, which is the last stage in HaplotypeCaller, accounts for half of the time in the pipeline when relaxing a disk performance constraint with higher throughput disk.

Then we study pipeline characteristics from resource usage perspective. Figures 5 show each resource usage (i.e. CPU, Disk, Memory, and Network) focusing on a node when running benchmark over twelve nodes. (a) of the figures represents the metrics in 3K-IOPS and (b) in 20K-IOPS respectively. Each resource usage graph also has a map onto the Spark jobs where they run. In 3K-IOPS case while running jobs 0 and 1, disk read/write bandwidth reached up to the limit, around 45 MB/sec in total. In contrast, the 20K-IOPS case utilized around 300 MB/sec and it still has a bandwidth capacity up to the 20K-IOPS limitation. We can see the limitation from how much wait time accounts for in CPU usage graph as well; Utilizing 3K-IOPS accounts for 20-40% in wait while running jobs 0 and 1, but 20K-IOPS does not. We can also observe an interesting characteristics in disk usage; read operation happens only in job 0 and 1. In other words, the later jobs do not read data from HDFS, but read from shuffle write data stored in memory as a file cache. As shown in the memory usage graphs, file cache occupies over half of memory in both scenarios. Job 3, the final phase in MarkDuplicate, starts utilizing all resources evenly, but especially consumes a huge amount of heap memory and network for shuffling...
### C. Analytics Infrastructure at Runtime

Next we evaluate performance from a different angle; we investigate how much time is required to set up a genome analysis system on a cloud, such as instance start-up, software installation, and genome data loading time to HDFS. Most previous works assume that Spark and Hadoop are already available, but it is important to keep minimizing resource usage on a cloud in terms of cost reduction [6]. An ideal situation is that we construct genome analysis pipeline at runtime, and then deallocate the system after finishing all pipelines. So we evaluate an analytics system setup time from scratch to understand how practical it is.

Table III shows how much time is spent in each phase. We used Terraform to set up infrastructure, which can help provisioning volumes and instances easily on a cloud. We also prepared a VM image that has already included Spark and Hadoop jars. The data loaded into HDFS is stored in local on the driver node. As shown in the table, provisioning infrastructure does not take so much time, so it is trivial compared to the entire pipeline computation time. However, data loading time is quite large. It takes about 30 minutes in our test. This cost is dominant in the entire pipeline, since it takes 45 minutes to finish the computation pipeline on a twelve node cluster with 20K-IOPS disk as shown in figure 3. Although the transfer time and speed depend on the disk or network bandwidth, it might not be negligible if we copy vast mounts of genome data into HDFS every time when we deploy a new system at runtime.

### IV. CHALLENGES AND APPROACHES

In this section, we summarize what challenges still remain to be optimized in genome analysis pipeline on a cloud, based on the results and workload characteristics shown in the previous section. We also explain what approaches we can use do tackle those remaining challenges, especially for relying on a strategy of decoupling compute and storage.

#### A. Storage Elasticity

**Challenge:** As shown in the figure 4 and table II, each pipeline has different resource usage patterns and characteristics, such as CPU-intensive or data-intensive, and this tendency often causes resource waste. We hypothesize that by modeling the pipelines we can flexibly utilize resources. But with the current analytics system, it would be difficult to achieve this degree of elasticity, because a typical analytics system (i.e. Spark with HDFS) requires tightly coupled compute with storage. This collocation concept is always effective to achieve the best performance with an on-premise system, however, it is difficult to scale compute and storage independently. To take full advantage of modern cloud elasticity, genome analysis pipeline should be also decoupled from compute and storage to reduce cost and achieve high scalability.

Moreover, data copying is another potential overhead as shown in the table III. If we continuously execute genome analysis pipelines on an analytics system for an extended period, data loading time to HDFS might be negligible. However, HDFS architecture does not expect to scale or descale underlying nodes frequently so we cannot avoid restructuring HDFS. Otherwise we might accept consuming unnecessary resources if workload size and demands are changing.

**Approach:** Object storage architecture can overcome the limit of storage scalability. In addition, we can delegate storage durability and availability to cloud. Even though object storage is not a file system, it would be applicable if it can reduce cost and achieve sufficient performance compared to HDFS.

---

#### TABLE II

<table>
<thead>
<tr>
<th>Job</th>
<th>HDFS input</th>
<th>HDFS output</th>
<th>shuffle read</th>
<th>shuffle write</th>
<th>GATK pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>154GB</td>
<td>-</td>
<td>12.8MB</td>
<td>226GB</td>
<td>Read+MarkDup</td>
</tr>
<tr>
<td>1</td>
<td>154GB</td>
<td>-</td>
<td>-</td>
<td>226GB</td>
<td>Read+MarkDup</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>226GB</td>
<td>Read+MarkDup</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>498GB</td>
<td>45.8GB</td>
<td>BQSR</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>522GB</td>
<td>283GB</td>
<td>BQSR</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>262GB</td>
<td>-</td>
<td>HaplotypeCaller</td>
</tr>
<tr>
<td>6</td>
<td>-</td>
<td>-</td>
<td>14.2MB</td>
<td>14.2MB</td>
<td>HaplotypeCaller</td>
</tr>
<tr>
<td>7</td>
<td>104MB</td>
<td>995MB</td>
<td>524GB</td>
<td>-</td>
<td>HaplotypeCaller</td>
</tr>
</tbody>
</table>

---

#### TABLE III

<table>
<thead>
<tr>
<th></th>
<th>create volumes</th>
<th>create instances</th>
<th>load data into HDFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>elapsed time</td>
<td>56 sec</td>
<td>2.5 mins</td>
<td>30 mins</td>
</tr>
</tbody>
</table>

---

![Fig. 4. Breakdown analysis of job execution time (a and b) and speedup ratio in each job (c and d)](image-url)
B. Compute Elasticity

**Challenge:** Besides storage elasticity, compute elasticity might help to dynamically adjust the resource usage to our genome workloads. As shown in the Figure 4, required resources and core scalability characteristics are different in each job. Similar to the storage elasticity, we might be able to reduce unnecessary compute resource as well. This result indicates that GATK pipelines have a potential capability to accept different size of resources for each execution stage.

**Approach:** Execution runtime and framework supports are mandatory to schedule resource dynamically. Spark provides a dynamic resource allocation feature to adjust resource usage that our workloads consume. Based on resource request or remove policy, Spark scheduler increases or decreases additional executors. On the other hand, many applications including GATK are containerized recently, so it is natural to consider running on an orchestration framework such as Kubernetes. Cloud workflow engines (such as Kubeflow and Argo) are intended to manage data pipeline in a Kubernetes native way. Thus, it is important to redesign an overall pipeline and think how we can accelerate pipeline performance on these systems with minimal resources. We do not have enough space to discuss compute elasticity, so we only focus on storage elasticity here, but we plan to integrate it with GATK as a future work.

V. PERFORMANCE EVALUATION

A. Scalability: GATK with Cloud Object Storage

Figure 6 describes a new overall design for GATK integrating with COS to achieve cloud elasticity. Instead of HDFS, we enhanced GATK to read/write genome dataset from/to COS. To realize this design, we enable Stocator [14] in Spark. Since GATK and required libraries that heavily depend on HDFS, we modified them to access objects from the GATK analysis pipeline.

C. Design and Implementation

Figure 6 describes a new overall design for GATK integrating with COS to achieve cloud elasticity. Instead of HDFS, we enhanced GATK to read/write genome dataset from/to COS. To realize this design, we enable Stocator [14] in Spark. Since GATK and required libraries that heavily depend on HDFS, we modified them to access objects from the GATK analysis pipeline.
and 68 minutes in COS and HDFS. On the other hand, COS is always slightly worse than HDFS in the 20K-IOPS scenario; the performance gap between them was 5% in two nodes test, but it accounted for 24% in twelve nodes test, and the elapsed time in COS and HDFS were 57 and 46 minutes. We will discuss why this drawback exists later.

As for the resource usage shown in Figure 8, we can see several notable characteristics compared to the result with HDFS. For example, disk bandwidth is consumed only by shuffle write. Additionally, jobs 0 and 1 finished within 7 minutes even though it took 28 minutes with 3K-IOPS HDFS. That is because network throughput is much larger than disk IO throughput.

B. Optimization for GATK with Cloud Object Storage

Next, we investigate why this slowness is noticeable with COS (20K-IOPS). Figure 9 shows the breakdown analysis that shows how much time each job spends on the scaling test with a 20K IOPS disk. As shown in the graph, Spark job performance is almost the same between HDFS and COS, but COS has an additional part, writing to COS phase. This additional overhead takes around 10 minutes to save a final VCF file output into COS. This is a constant that does not depend on the node scale but on the finalized data size. As a result, the speedup ratio becomes gradually worse in the twelve node test since this constant cost is relatively dominant in the full computation, even though the pipeline computation part itself is competitive with HDFS.

Why does COS have this overhead when HDFS does not? The reason comes from the difference in the supported Hadoop FileSystem API between HDFS and COS. Figure 10 shows a diagram and operational flow of the final phase in a variant searching pipeline. Blue, orange, and green lines represent original flow using HDFS, original flow using COS, and optimized flow using COS without any concat operation respectively. The last stage of job 7 manages a large number of reducer tasks that generate a piece of the final VCF file, and then persist them into HDFS or COS. After finishing job 7, the GATK main driver explicitly calls a concat operation in Hadoop FileSystem API which merges them into a single file. HDFS implements all operations including concat, which can complete the concat operation without any copies inside or outside the cluster. An HDFS client has the capability to read these pieces as a file at runtime, so it does not need to physically merge them into one. As a consequence, this concat operation finishes immediately in HDFS. On the other hand, an object storage connector to COS does not implement the concat operation in its Hadoop compatible file system. Although the object storage connector imitates a Hadoop FileSystem, it is essentially not easy to support all APIs due to the difference of backend storage implementation and a general limitation existing in S3 compatible object storage system. As a result, the GATK main driver performs error handling; it copies all pieces into the local driver first, merges them into a final VCF file, then pushes it back to an object in COS. Thus, this additional constant data copying overhead in the finalization phase always exists in GATK with COS.

We have several possible approaches to eliminate this overhead. One approach is to modify GATK internal code to stop calling concat. Another approach is for the object storage connector to simply pass through concat operations without raising errors. In both approaches, a client must understand how to read these pieces, but it can suppress unnecessary data copy. We also implemented code to skip this file merge process to demonstrate how to reduce this overhead. As a result, by using all of these techniques, the performance of COS tests can be uniformly reduced by 10 minutes in the final sink phase; COS (3K-IOPS) is 28% faster than HDFS (3K-IOPS), and COS (20K-IOPS) shows almost same performance in HDFS (20K-IOPS).

C. Cost: Price Per Performance

Finally, we compare actual costs between COS and HDFS in our scenarios. Figure 11 compares costs while changing...
node scaling and disk configurations. We calculate total cost based on a public price list and then divide it by elapsed time. The first scenario focuses on pipeline computation time only, represented with solid lines. COS always achieves better cost performance than HDFS with 3K-IOPS disk. While utilizing 20K-IOPS disk, HDFS outperforms COS due to the difference in elapsed time for entire pipeline execution. Comparing 3K-IOPS with 20K-IOPS, cost performance in 3K-IOPS is 15-60% better than 20K-IOPS. This is because 20K-IOPS is 10x more expensive than 3K-IOPS.

The next scenario compares COS with HDFS when eliminating a concat overhead in OCS and loading all data into HDFS, represented with dotted lines. If we start from data loading phase into HDFS, analytics pipeline must wait to complete all data transfer. Moreover, we cannot load data before starting the cluster. Therefore, we appended the additional transfer time (i.e. 30 minutes) to the elapsed time in HDFS result. In COS result, we can manage data transfer task independently, so we do not account the cost here. In addition, COS results include the optimization which removes an additional data sink overhead (i.e. 10 minutes). In such a situation, as a result, it can achieve up to 67% cost savings with 3K-IOPS disk, and up to 61% cost savings with 20K-IOPS disk on twelve nodes.

VI. CONCLUSION

In this paper we investigate the performance characteristics of GATK using Spark with HDFS and identify scalability issues on a modern cloud. Based on a quantitative analysis, we introduce a new approach to utilize cloud object storage in GATK instead of HDFS, which helps decouple compute and storage. We demonstrate how this approach contributes to performance scalability and cost saving in a cloud. We also reveal an existing overhead when utilizing cloud object storage in current GATK. By mitigating this performance issue, we finally confirm GATK using COS can achieve a 28% performance improvement over than HDFS while using a slower but more expensive disk, and completely the same performance with HDFS using a faster but more expensive disk. Moreover, we show that it can achieve up to 67% cost savings to complete all genome analysis pipeline including data loading time into HDFS.
Self-adaptive Threshold-based Policy for Microservices Elasticity

Fabiana Rossi, Valeria Cardellini, Francesco Lo Presti

DICII, University of Rome Tor Vergata, Italy
{f.rossi,cardellini}@ing.uniroma2.it, lopresti@info.uniroma2.it

Abstract—The microservice architecture structures an application as a collection of loosely coupled and distributed services. Since application workloads usually change over time, the number of replicas per microservice should be accordingly scaled at runtime. The most widely adopted scaling policy relies on statically defined thresholds, expressed in terms of system-oriented metrics. This policy might not be well-suited to scale multi-component and latency-sensitive applications, which express requirements in terms of response time.

In this paper, we present a two-layered hierarchical solution for controlling the elasticity of microservice-based applications. The higher-level controller estimates the microservice contribution to the application performance, and informs the lower-level components. The latter accordingly scale the single microservices using a dynamic threshold-based policy. So, we propose MB Threshold and QL Threshold, two policies that employ respectively model-based and model-free reinforcement learning approaches to learn threshold update strategies. These policies can compute different thresholds for the different application components, according to the desired deployment objectives. A wide set of simulation experiments shows the benefits and flexibility of the proposed solution, emphasizing the advantages of using dynamic thresholds over the most adopted policy that uses static thresholds.

Index Terms—Hierarchical Control, Elasticity, Self-adaptation, Microservice, Reinforcement Learning

I. INTRODUCTION

To take advantage of cloud computing and to improve efficiency and scalability of applications, most of the IT companies (e.g., Amazon, Netflix, Spotify) are currently reshaping their applications from monolithic architectures to microservices. According to the microservices architectural style, an application can be split into many autonomous and decoupled services, each providing a specific functionality.

Exploiting elasticity, each microservice can be dynamically scaled, enabling to control the application deployment with a fine granularity and to reduce the scaling cost compared to conventional monolithic solutions. Besides controlling elasticity of single components, microservice applications require to efficiently coordinate the distributed scaling decisions so to properly process varying workloads and meet application-level Quality of Service (QoS) requirements. Although elasticity has been widely explored in the context of cloud computing [1], scaling microservice applications, where multiple components loosely cooperate and interact with one another, has only recently started to be investigated (e.g., [2]–[5]). In this setting, the complexity of managing microservice applications which include, among the others, the challenges posed by the need to map application into microservices’ requirements as well as the dynamism of the execution environments, demand novel and autonomic solution to control microservices elasticity. Today’s cloud providers that support multi-component applications (e.g., using containers and container orchestration engines) allow to create multiple, decentralized auto-scaler instances, each carrying out the adaptation of a single microservice deployment. Policies to coordinate the scaling decisions at the application level are missing. To determine the scaling actions, most of existing auto-scalers use static thresholds on system-oriented metrics (e.g., CPU utilization)\(^1\). The main idea is to increase (or reduce) the microservices’ parallelism degree as soon as the metric is above (or below) a critical scale-out (or scale-in) value. Although this decentralized approach scales well, we observe that manually tuning such scaling thresholds is challenging, especially when we need to define multiple thresholds, one for each microservice. A further challenge arises from the need of specifying a critical value on a system-oriented metric, whereas the application usually exposes its requirements in terms of user-oriented metrics (e.g., response time, throughput, cost).

In this paper, we design a self-adaptive threshold-based scaling policy that can automatically learn and update the scaling thresholds for each application component. Although self-adaptive thresholds have been already studied in different contexts (e.g., virtual machine consolidation [6], performance management [7]), to the best of our knowledge, they have never been applied to microservice applications. The main contributions of the paper are as follows.

- We propose a two-layered hierarchical solution for controlling elasticity. Having a complete system view, a high-level centralized entity estimates at run-time the relationship between application- and microservice-level QoS requirements (global policy). At low level, decentralized entities locally control the adaptation of single microservices (local policy).
- As local policy, we propose QL Threshold and MB Threshold, two dynamic threshold-based policies realized using model-free and model-based Reinforcement Learning (RL) solutions, respectively. Intuitively, a model-based approach allows us to account for the known (or estimated) system dynamic improving the algorithm learning speed. As global policy, we propose a simple heuristic

\(^1\)https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
that dynamically estimates and adapts each microservice contribution to the overall application performance.

- Using simulation, we demonstrate the benefits of a hierarchical control, the advantages of the proposed MB Threshold policy, and the flexibility of RL-based solutions that can identify different trade-offs between improving application performance and avoiding resource wastage. We also compare our solution against the widely adopted static threshold-based policy.

II. RELATED WORK

In this section, we analyze existing approaches proposed in literature for the adaptive deployment of microservice-based applications in cloud environment. We broaden the view also to monolithic applications because, so far, only few research works [2]–[5] have specifically targeted the elasticity of microservice-based applications. Existing elasticity policies range from model-free to model-based solutions, according to the degree of system knowledge exploited to approximate the application behavior. A model-free solution requires no knowledge of the system dynamics; so it can take sub-optimal decisions or may require manual parameter tuning. Different model-based techniques have been proposed for application scaling, such as control theory [8], queuing theory [3], time series analysis [9], or a combination thereof [10]. The model-based approaches usually need many training samples that can be extract from historical data. In general, it is hard to perform off-line model training of microservice-based applications because they can be complex and have highly dynamic behaviors.

The main methodologies to adapt the application deployment are: threshold-based heuristics, fuzzy logic, queuing theory, and machine learning-based solutions. The most popular approach uses best-effort threshold to change the application replication degree at run-time (e.g., [7], [10]). Most works propose a static threshold-based approach and use, as a QoS metric, the resource utilization of either the system nodes or the application replicas (e.g., [4], [10], [11]). Although threshold-based policies are simple to design, they require a manual threshold tuning that, in general, is not a trivial task. To overcome this issue, self-adaptive (or dynamic) threshold-based policies have been proposed in literature. They have been used in different contexts, such as performance management, virtual machines consolidation, scaling of monolithic applications (e.g., [6], [7], [12]); however, to the best of our knowledge, they have never been applied in a microservices-based scenario. Fuzzy logic approaches use pre-defined collections of if-then rules that represent how to take decisions and control a system according to the human knowledge (e.g., [13], [14]). When the scale of the controlled system increases, determining robust rules to combine conjunctive or disjunctive clauses becomes hard, also because we cannot give different importance to the factors to be combined. Queuing theory models an application as a queuing network. This allows to predict the application performance under different conditions of load and replication, and accordingly drive the scaling operations (e.g., [3], [10], [15]). Queuing models return only approximated behavior; moreover, parameterizing them correctly requires an extensive application profiling that can be time-consuming and costly.

Recently, machine learning policies are becoming appealing to manage and adapt complex systems also in a fully decentralized manner (e.g., [2], [5], [16]). In this field, reinforcement learning is a special technique by which an agent can learn how to make good decisions through a sequence of interactions with the environment. Most of the works consider the model-free RL (e.g., Q-learning) algorithms (e.g., [7], [14], [17]) which, however, suffer from slow learning rate. As such, the auto-scaler performs poorly during the learning period. To overcome the slow convergence rate of these solutions, model-based RL approaches have been proposed. Tesauro et al. [18] use queuing network to model the application performance. In [16], we present a model-based solution that empirically estimates the system model and scales monolithic applications.

In this paper, we resort to a threshold-based policy where we use RL to automatically learn and adapt the scaling thresholds at run-time. The work by Horovitz et al. [7] is the most closely related to ours. We differ from their solution from both the architectural and methodological standpoints.

First, Horovitz et al. propose a centralized heuristic based on a model-free Q-learning approach to dynamically scale monolithic applications. Conversely, we design a two-layered hierarchical solution to adapt at run-time the deployment of multi-component applications. Exploiting a full system view, the high-level control entity estimates at run-time each microservice contribution to the application response time and accordingly notifies the low-level per-microservice managers. Moreover, we propose a model-based RL solution that can speed-up the learning phase of the RL agents by exploiting the knowledge on the system dynamics. As such, we do not rely on additional heuristics to determine whether to activate the RL agent (as done in [7]).

III. SYSTEM ARCHITECTURE

A. Problem Definition

A microservice-based application results by the cooperation of different independently deployable services. Nevertheless, the overall application performance results by the smooth integration and cooperation between its microservices. Without loss of generality, we can model a microservice-based application as directed acyclic graph (DAG), where the vertices represent the application microservices and the edges the logical links or dependencies between them [19]. Two services are interconnected if they directly communicate to reach a common goal (i.e., to satisfy an external request). Let $M$ be the set of all the application microservices. We define the vertices without incoming (internal) links as sources and those without outgoing links as sinks. A front-end service (e.g., a gateway) is a source, because it can forward the user requests to the other microservices; a sink only returns a response, without

\[2\] The application DAG can be manually defined, e.g., in the deployment specification, or can be estimated at run-time using, e.g., a service mesh. Dependency cycles in microservice-based applications should be removed for data integrity and to reduce the risk of outages [20].
invoking other services. We define the set of all source-sink paths as $\Pi$. Note that a single microservice can be a member of multiple paths. Given a microservice $m \in M$, we denote the set of all paths that include $m$ as $\Pi_m \subseteq \Pi$.

In this work, we consider latency-sensitive applications that expose QoS requirements in terms of a target response time that should not be exceeded (i.e., $T_{max}$). Since the application workload usually fluctuates over time, the number of replicas of each microservice should be accordingly scaled at run-time so to meet the $T_{max}$ requirement avoiding resource wastage. Multiple microservice replicas can process incoming requests in parallel, thus reducing the per-replica load and, in turn, the processing latency.

**B. Hierarchical Control Architecture**

To manage and coordinate the microservices auto-scaling so to obtain desirable application performance, we need a deployment controller that provides self-adaptation mechanisms and can be equipped with deployment policies. The MAPE loop represents a prominent and well-know architectural pattern to organize the deployment controllers, where four components (Monitor, Analyze, Plan, and Execute) are responsible of self-adaptation actions. As described in [21], different patterns have been used in practice to decentralize the MAPE control loop. Among them, the hierarchical control pattern structures the adaptation logic as a hierarchy of MAPE control loops, promising to exploit the benefits of both centralized and decentralized architectures. In [3], we designed a hierarchical approach where the centralized controller issues reconfiguration requests to decentralized managers. In this paper, we consider a different approach, where the centralized controller only provides a feedback to the decentralized managers, which autonomously perform the adaptation actions.

Figure 1 illustrates the deployment controller architecture, highlighting the two-layered approach. The Application Manager and Microservice Managers can work at different time scales. Importantly, the Application Manager provides feedback to each Microservice Manager, which is then taken into account by the Analyze and Plan components of its MAPE loop cycle. At lower-level, we define multiple, decentralized, and autonomous Microservice Managers, each controlling a single microservice using what we call a local policy. The Monitor collects data about the monitored microservice (i.e., response time and resource utilization). Then, the local Analyzer processes the monitored data and determines whether an adaptation action is needed. If an updated is required, the Planner identifies which adaptation action is beneficial according to the local policy. Finally, the Executor enacts the deployment changes. Exploiting a broader system view, the high-level Application Manager steers the overall adaptation by providing guidelines to the lower levels through a global policy. First, it monitors the application performance (i.e., response time) and retrieves the microservice QoS metrics. After their analysis, it uses its global policy to estimate the relative contribution of each microservice to the overall application performance. This information is then forwarded to the Microservice Managers, which can accordingly update the microservices deployment in parallel.

**IV. LOCAL THRESHOLD-BASED SCALING POLICY**

At the local control level, our goal is to rely on dynamic thresholds and establish a method for automatically adapting their value at run-time, so to efficiently scale each microservice. We use reinforcement learning to learn the scaling threshold adaptation strategy. A RL agent learns what to do (i.e., how to map situations to actions) through direct interaction with the system [22]. It aims to learn an optimal adaptation strategy, so to minimize a numerical cost signal. To minimize the obtained cost, a RL agent must prefer actions that it found to be effective in the past (exploitation). However, to discover such actions, it has to explore new actions (exploration). One of the main challenges in RL is to find at run-time a good trade-off between the exploration and exploitation phases.

The Microservice Manager local policy implements the Analyze and Plan steps of the decentralized MAPE loops. For each Microservice Manager, we consider a RL agent in charge of adapting at run-time the scale-out threshold for the controlled microservice, aiming to minimize a long-term cost. In this work, we do not dynamically update scale-in thresholds; nevertheless, the proposed methodology can be easily extended to account also for these thresholds. The RL agent interacts with the microservice in discrete time steps. At each time step, the agent observes the microservice state and performs an action. One time step later, the microservice transits in a new state, causing the payment of an immediate cost. Both the paid cost and the next state transition usually depend on external unknown factors, hence are stochastic. To minimize the expected long-term cost, the agent estimates the so-called Q-function. It consists in $Q(s, a)$ terms, which represent the expected long-term cost that follows the execution of action $a$ in state $s$. To update the scale-out threshold, given the system state $s$, the agent performs the action $a$ that minimizes $Q(s, a)$. By observing the incurred immediate costs, $Q(s, a)$ is updated over time, thus improving the threshold update policy.

**State.** For each microservice $m$, we define its state at time $i$ as $s_i = (\theta_i, u_i)$, where $\theta_i$ is the scale-out threshold, and $u_i$ is the average CPU utilization of the microservice. We denote by $S$ the set of all the microservice states. Being CPU utilization $(u_i)$ a real number, we discretize it by assuming that $u_i \in \{0, \bar{u}, ..., \bar{L}\}$, where $\bar{u}$ is a suitable quantum and $\bar{L} \in \mathbb{N}$ s.t. $\bar{L} \bar{u} = 1$. We also assume that the scale-out threshold $\theta_i$ ranges in the interval $[\theta_{\min}, \theta_{\max}]$, where $0 < \theta_{\min} \leq \theta_{\max} < 1$. 
Action. According to an action selection policy (e.g., \epsilon-greedy), the RL agent identifies the threshold adaptation action to be perform. For each state \( s \in S \), we have a set of feasible adaptation actions \( A(s) \subseteq A \), where \( A \) is the set of all actions. Formally, the action model consists of \( A = \{ -\delta, 0, \delta \} \), where \( \delta \in (0, 1) \) is a suitable threshold quantum. In particular, \( \pm \delta \) represents a threshold adaptation action (i.e., \( +\delta \) to add a threshold quantum and \( -\delta \) to remove a threshold quantum), and \( a = 0 \) is the do nothing decision. Obviously, not all the actions are available in any microservice state: an action \( a \) is valid in a state \( s = (\theta, u) \) if \( \Theta_{\text{min}} \leq \theta + a \leq \Theta_{\text{max}} \).

Cost Function. We define an immediate cost function \( c(s, a, s') \) to capture the cost of carrying out action \( a \) when the microservice state transits from \( s \) to \( s' \). The RL agent wants to minimize the cost so to jointly satisfy application performance and limit resource wastage. For this purpose, the cost function includes two different contributions:

- the performance penalty \( c_{\text{perf}} \), paid whenever the microservice response time \( t_m \) is approaching (or exceeds) the response time bound \( T_{m,\text{max}} \). This latter parameter is provided by the Application Manager as a function of the overall application response time \( T_{\text{max}} \).
- the resource cost \( c_{\text{res}} \) for running the microservice. We can reasonable assume that the resource cost increases when the scale-out threshold decreases, because the lower the scale-out threshold, the higher the number of used resources.

We combine the two cost contributions into a single weighted cost function, where the distinct weights allow us to express the relative importance of each cost term. Formally, we define the immediate cost function \( c(s, a, s') \) as the weighted sum of the costs, normalized in the interval \([0, 1]\):

\[
c(s, a, s') = w_{\text{perf}} \cdot c_{\text{perf}}(s, a, s') + w_{\text{res}} \cdot c_{\text{res}}(s, a, s') \tag{1}
\]

where:

\[
c_{\text{perf}}(s, a, s') = \begin{cases} 
\xi \left( \frac{t_m - T_{m,\text{max}}}{T_{m,\text{max}}} \right)^2 & t_m \leq T_{m,\text{max}} \\
1 & \text{otherwise}
\end{cases}
\]

\[
c_{\text{res}}(s, a, s') = (1 - \theta')
\]

with \( \xi \) is a parameter determining the exponential function steepness, and \( t'_m \) is the microservice response time in \( s' \).

Intuitively, the cost function allows us to instruct the Microservice Manager to discriminate between the good system configurations and actions and the bad configurations and actions. As the Microservice Manager aims to minimize the incurred cost, it is encouraged to (i) keep the response time within the given bound and (ii) limit the resource usage. The different weights allow us to express the relative importance of each cost term. We note that this policy only indirectly optimizes the microservice performance: it updates the scaling threshold, which is then used by the Microservice Manager to scale the microservice based on its average CPU utilization.

We also observe that the response time bounds \( T_{m,\text{max}} \) grant a share of the global application response time bound \( T_{\text{max}} \) to each microservice, accordingly to its relative contribution. The \( T_{m,\text{max}} \) terms could be set either statically after preliminary profiling, or dynamically estimated and adapted at run-time by the Application Manager. In Section V, we describe a simple criterium to set these bounds for our reference application.

Q-function Update. To update the Q-function, we consider two RL approaches that differ for the actual learning algorithm adopted and on the assumptions about the system. We first consider the simple model-free Q-learning algorithm that requires no knowledge of the system dynamics. Then, we propose a model-based approach, which exploits what is known (or can be estimated) about system dynamics to accordingly update the Q-function and speed-up the learning phase.

1) Q-learning Threshold (QL Threshold): Q-learning is a model-free RL algorithm that does not require a knowledge of the system dynamics. At time \( i \), the Q-learning agent observes the microservice \( m \) state \( s_i \) and selects \( a_i \) using an \( \epsilon \)-greedy policy on \( Q(s_i, a_i) \); the microservice transits in \( s_{i+1} \) and experiences an immediate cost \( c_i \). The \( \epsilon \)-greedy policy selects the best known action for a particular state (i.e., \( a_i = \arg \min_{a \in A(s_i)} Q(s_i, a) \)) with probability \( 1 - \epsilon \), whereas it favors the exploration of sub-optimal actions with low probability. At the end of each time slot \( i \), \( Q(s_i, a_i) \) is updated using a simple weighted average:

\[
Q(s_i, a_i) \leftarrow (1 - \alpha) Q(s_i, a_i) + \alpha \left[ c_i + \gamma \min_{a' \in A(s_{i+1})} Q(s_{i+1}, a') \right] \tag{2}
\]

where \( \alpha \in [0, 1] \) is the learning rate parameter and \( \gamma \in [0, 1) \) is the discount factor.

2) Model-Based Threshold (MB Threshold): The RL agent identifies the threshold adaptation action \( a_i \) to perform for microservice \( m \) in state \( s_i \) relying on a possibly approximated system model. Differently from model-free solutions, the model-based RL approach does not use an action selection policy, but it always selects the best action in term of Q-values, i.e., \( a_i = \arg \min_{a \in A(s)} Q(s_i, a) \). Moreover, in the model-based RL approach we directly use the Bellman equation to update the Q-function:

\[
Q(s, a) = \sum_{s' \in S} p(s'|s, a) \left[ c(s, a, s') + \gamma \min_{a' \in A(s')} Q(s', a') \right] \quad \forall s \in S, \forall a \in A(s) \tag{3}
\]

where we use estimates for the unknown or partially unknown transition probabilities \( p(s'|s, a) \) and/or the cost function \( c(s, a, s'), \forall s, s' \in S \).

For the estimates of \( p(s'|s, a) \), it is sufficient to compute the CPU utilization transition probabilities \( P[u_{i+1} = u'|u_i = u] \). Formally:

\[
p(s'|s) = \begin{cases} 
P[u_{i+1} = \{\theta', u\} | u_i = \{\theta, u\}, \alpha_i = \delta] & \theta' = \theta + \delta \\
0 & \text{otherwise}
\end{cases} \tag{4}
\]

\(^3\)Intuitively, the model-based approach boils down to replacing the model-free equation (2) with one step of the value iteration algorithm using estimates for the unknown parameters.
Since $u$ takes value in a discrete set, we will write $P_{j,j'} = P[j_{i+1} = j' | j_i = j, j, j' \in \{0, \ldots, L\}$ for short. We estimate $p(s'|s, a)$ as the relative number of times the CPU utilization changes from state $j\tilde{u}$ to $j'\tilde{u}$ in the time interval $\{1, \ldots, i\}$.

For the estimates of the immediate cost $c(s, a, s')$, we observe that it can be written as the sum of two terms, respectively named as the known and the unknown cost:

$$c(s, a, s') = c_k(s, a) + c_u(s') \quad (5)$$

The known cost $c_k(s, a)$ depends on the current state and action; in our case, it accounts for resource costs. The unknown cost $c_u(s')$ depends on the next state $s'$. As in (1), $c_u(s')$ accounts for the performance penalty. As we assume that the application model is not known, we have to estimate $c_u(s')$ at run-time. Therefore, at time $i$, the RL agent observes the immediate cost $c_i$, computes $c_{u,i}(s') = c_i - c_{k,i}(s, a)$, and updates the estimate of the unknown cost $\hat{c}_{u,i}(s')$, as follows:

$$\hat{c}_{u,i}(s') \leftarrow (1 - \beta)\hat{c}_{u,i-1}(s') + \beta c_{u,i}(s') \quad (6)$$

where $\beta \in [0, 1]$ is the smoothing factor. $\hat{c}_{u,i}(s')$ is used to compute the cost of applying $a$ in $s$ according to (5). Given a state $s = (\theta, u)$, we observe that in the next state $s' = (\theta', u')$ the expected cost due to $T_{m,\max}$ violation is not lower when the scale-out threshold and/or the CPU utilization increases. Vice versa is also true. Therefore, to speed-up the learning phase, we can heuristically enforce the following properties while updating $\hat{c}_{u,i}(s'), \forall s \in S$:

$$\hat{c}_{u,i}(s) \leq \hat{c}_{u,i}(s'), \forall \theta \leq \theta', u \leq u'$$

$$\hat{c}_{u,i}(s) \geq \hat{c}_{u,i}(s'), \forall \theta \geq \theta', u \geq u'$$

V. GLOBAL SCALING POLICY

Hierarchical policies can scale well in the face of applications composed by a high number of microservices, because of the clear separation of concerns and distribution. Exploiting a system-wide view of the application execution, the Application Manager can easily provide a feedback or proactively notify the different Microservice Managers to improve their cooperation and meet the application requirements.

The Application Manager global policy implements the Analyze and Plan steps of the centralized MAPE loop. Its main goal is to conveniently pinpoint the bottleneck microservices so to not exceed the target application response time $T_{\max}$. To avoid over-complicating the hierarchical policy design, we resort to a simple global policy that dynamically estimates and adapts at run-time the relative microservice contribution to the overall application response time. At each iteration of the MAPE loop and for each microservice $m$, the Application Manager estimates its target response time $T_{m,\max}$ as $T_{m,\max} = \nu_m \cdot T_{\max}$, where $\nu_m$ is the (average) contribution of microservice $m$ to the overall application response time. To determine $\nu_m$ for each microservice, the Application Manager uses the application DAG. Hence, $\nu_m$ is updated using a simple exponential weighted average:

$$\nu_m \leftarrow (1 - \phi)\nu_m + \phi \frac{\hat{t}_m}{I_{1m}} \quad (7)$$

where $\phi \in [0, 1]$ is the smoothing factor, $\hat{t}_m$ is the average microservice response time and $I_{1m} = \sum_{p \in \Pi_m} p_m \cdot \hat{t}_m$ being $\hat{t}_m$ the weighted average of the response times of all sink-source paths including $m$ with $\Pi_m \subseteq \Pi$, and $p_m \in [0, 1]$ the probability that a service request invokes path $\pi$. The Application Manager then sends the $T_{m,\max}$ value to the Microservice Manager in charge of controlling $m$. Relying on its local policy, the Microservice Manager can accordingly update its scaling strategy.

VI. EXPERIMENTAL RESULTS

We evaluate the proposed deployment adaptation solutions by means of simulations. To capture the variability of microservice-based applications, we consider three different types of application DAG, namely sequential, diamond, and complex, as shown in Figure 2. They have the same number of microservices. Within each microservice $m$, Figure 2 reports its service rate $\mu_m$; on top of each microservice, we show the ratio between the overall outgoing request rate and the incoming request rate; and on the outgoing edges of $m$, we show the invocation probability of the successor microservices of $m$, considering a probabilistic microservice invocation [23]. Moreover, we compare the proposed dynamic-threshold approaches (i.e., MB Threshold and QL Threshold) against a static threshold-based policy (i.e., Static Threshold), which represents the most widely adopted auto-scaling solution in container orchestration frameworks (e.g., Kubernetes).

Without lack of generality, at each discrete time step $i$, we model each microservice as an $M/M/k_i$ queue, where $k_i$ is the number of microservice replicas.

We set the basic service rate $\mu$ shown in Figure 2 to 140 requests/s. Each application requires its overall response time to be below $T_{\max} = |\Pi|/\mu$ ms, where $|\Pi|$ is the length of the longest application path and $\bar{\mu}$ is the lowest components’ service rate; thus, the complex application requires $T_{\max} = 59.5$ ms, the sequential one $T_{\max} = 83.3$ ms, and the diamond one $T_{\max} = 35.7$ ms. We consider that the application receives an incoming request rate that changes over time according to the workload pattern shown in Figure 3. The RL algorithms use the following parameters: $\Theta_{\min} = 0.5$, $\Theta_{\max} = 0.9$, $\xi = 10$, discount factor $\gamma = 0.99$; QL Threshold also uses $\alpha = 0.1$ and $\epsilon = 0.1$ and MB Threshold uses $\beta = 0.1$. For the global policy, we set $\phi = 0.1$. We use small values of the smoothing factors (i.e., $\alpha$, $\beta$, and $\phi$) so to weigh more recent samples and improve the agents ability to react to system changes.

To discretize the application state, we use $\bar{u} = 0.1$. The scale-in threshold is set to 20% of CPU utilization. Our simulator is written in Java and uses one class for each microservice that, in turn, is modeled as an $M/M/k$ queue. Two main classes, Application Manager and Microservice Manager, implement the layered MAPE loop. At each time step, the simulator calls the MAPE components of the Application Manager and, then, of the Microservice Managers. The Application Manager
TABLE I: Application performance using different threshold-based scaling policies.

<table>
<thead>
<tr>
<th>Topology</th>
<th>Policy</th>
<th>Configuration</th>
<th>Average Threshold value (%)</th>
<th>Standard deviation of the threshold value</th>
<th>Average replicas per service</th>
<th>Average CPUs allocation (%)</th>
<th>T_{max} Violations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compact</td>
<td>MB Threshold</td>
<td>w_{perf} = 1, w_{res} = 0</td>
<td>30.44</td>
<td>1.80</td>
<td>2.26</td>
<td>38.17</td>
<td>0.01</td>
</tr>
<tr>
<td>QL Threshold</td>
<td>w_{perf} = 0.50, w_{res} = 0.50</td>
<td>89.44</td>
<td>2.64</td>
<td>4.39</td>
<td>38.90</td>
<td>2.62</td>
<td>40.70</td>
</tr>
<tr>
<td>Static Threshold</td>
<td>w_{perf} = 0.50, w_{res} = 0.50</td>
<td>76.72</td>
<td>11.51</td>
<td>62.22</td>
<td>35.42</td>
<td>1.34</td>
<td>35</td>
</tr>
</tbody>
</table>

**Divergence of the thresholds per application components**

Due to space limitations, we mainly focus the discussion on the complex application; nevertheless, Table I reports the results for the other topologies, as well. Overall, we can see that, for a specific topology, the application performances change under the different scale-out threshold policies. The Static Threshold policy is application-unaware and not flexible, meaning that it is not easy to satisfy QoS requirements of latency-sensitive applications by setting a threshold on CPU utilization. From Table I, we can observe that small threshold changes may lead to a significant performance deterioration. Conversely, the dynamic thresholds can be trained to optimize different deployment objectives, e.g., see the response time median, T_{max} Violations, or the average number of replicas. The model-based RL solution can successfully learn a different strategy to update the scaling thresholds for the different application topologies. We note that, although the thresholds have similar value for all the topologies, their standard deviation changes, especially for multi-objective optimizations. This indicates a diversification of the thresholds per application components, whose value has been empirically determined by the RL agents. Figure 4 shows the application performance during the whole experiment, when the model-based solution updates the scaling thresholds for each application microservice (i.e., MB Threshold is used). We can see that the application has a different performance when different weights for the cost function are used (Eq. 1). When the cost function penalizes response time violations (i.e., with w_{perf} = 1), the median of the application response time is 33 ms (6% of T_{max} violations) and, on average, each microservice runs with 2.4 replicas. Conversely, when we aim to save resources (i.e., w_{res} = 1), the application response time median grows to 42 ms (18% of T_{max} violations) and, on average, each microservice runs with 1.7 replicas. The different behavior is also clear by comparing the overall number of microservices replicas during the experiment.
experiment (see Figures 4a and 4c). Figure 4c also shows that the application response time follows the incoming workload, with different response time peaks when CPU utilization is approaching 75%. On the other hand, Figure 4a shows the benefits of replication: since the application is readily scaled, the resulting response time is below $T_{\text{max}}$ and has also a reduced variance.

Besides the weight configurations at the opposite ends, we can obtain a wide set of adaptation strategies that differ by the relative importance of the two deployment goals. Here, we propose a simple case, where we set $w_{\text{perf}} = 0.50$ and $w_{\text{res}} = 0.50$. The median application response time is 39 ms, with about 3% of $T_{\text{max}}$ violations. In this case, we obtain an average threshold value that is rather close to the case of $w_{\text{perf}} = 1$, even though there is a higher variance due to the different thresholds computed for the different application microservices.

The MB Threshold policy is flexible enough to host different sets of weights, thus allowing to explore diverse trade-offs between improving performance and saving resource. Importantly, the RL approach can automatically learn the most suitable strategy to satisfy the user preferences, estimating the mapping between user- and system-oriented metrics.

B. Comparing QL and MB Thresholds

We now compare the model-based RL approach against the simple and model-free Q-learning solution. Both the RL strategies are used to update the scale-out thresholds in a distributed manner, for each application microservice. The two approaches resort on the cost function (Eq. 1) to receive a feedback of the performed action and update their Q-value estimations. To visualize the update of the scaling threshold by the two RL policies, we report in Figure 5 the threshold value for the first microservice (with 3$\mu$) of the complex application, when we want to optimize the performance ($w_{\text{perf}} = 1$). Intuitively, the best threshold should be the lowest possible, so to use as many replicas as possible. The model-based RL solution benefits from the system model to quickly learn how to update the scaling thresholds. This holds true for all the application topologies and cost function configurations (see Table I). Conversely, Q-learning continuously updates the scaling thresholds, meaning that it is still exploring the best actions to perform. This behavior is also reflected on the application response time, whose median value does not change under the different cost function configurations.

C. Hierarchical Application Control

In this section, we investigate the global policy, used to provide an adaptation feedback to the decentralized RL agents.

When a static threshold policy is used to scale a complex application, we usually set a single threshold value for all the application components. Tuning different thresholds for the different components is costly, so it is not usually done in practice. Conversely, the cooperation between the Application Manager and the decentralized Microservice Managers allows to automatically adapt the thresholds for the different application components, according to their run-time behavior. Figure 6 reports the utilization, scale-out threshold, and number of replicas of two components of the diamond application, microservice 1 (with 0.6$\mu$) and microservice 5 (with 1.4$\mu$). In this case, the MB Threshold policy computes the thresholds under the weights configuration $w_{\text{perf}} = w_{\text{res}} = 0.5$. We observe that the two microservices have a different degree of replication, with the bottleneck component, microservice 1, running with more replicas than microservice 5 (on average, 2 and 1, respectively). Apparently, the RL agent prefers to use a high value as the threshold rest value, which is promptly updated when scaling actions are needed or the application runs under heavy workload conditions (see the time interval between 2000 and 3000 time units). To further investigate the
global policy, we run another experiment where we turned off the Application Manager and statically set $T_{m,\text{max}} = T_{\text{max}} / 3$, $\forall m \in M$. As a result, the median application response time is 31 ms, violating $T_{\text{max}}$ 37% of the time (instead of 25 ms and 3% of bound violations as in the previous setting). We observe that the distributed RL agents more slowly learn to distinguish the bottleneck components and diversify the thresholds: e.g., the bottleneck component runs with 1.6 replicas (instead of 2). The Application Manager helps to capture the heterogeneity of the application microservices, leading to the definition of different thresholds that better optimize performance and avoid resource over-/under-provisioning.

VII. CONCLUSIONS

In this paper, we presented a novel self-adaptive threshold-based policy for scaling microservice-based applications. Specifically, we designed a two-layered hierarchical control architecture where, at the lower level, decentralized controllers scale microservices using dynamic thresholds and, at the higher level, a centralized controller analyzes the relative microservice contribution to the overall application performance. To update the scaling thresholds at run-time, we rely on model-free and model-based RL algorithms, obtaining QL Threshold and MB Threshold, respectively. As regards the global policy, we proposed a simple yet effective heuristic to empirically estimate the microservice contribution to the application performance. Using simulation, we evaluated the proposed solutions and compared them against a static threshold-based policy, which is the most widely adopted scaling strategy. While the QL Threshold policy suffers from slow learning rate, our MB Threshold clearly outperforms all the other approaches. Differently from a static threshold-based approach, the MB solution not only differentiates the scaling threshold for the different microservices, but can also improves the policy flexibility, because it can learn different threshold update strategies according to the deployment goals. As future work, we will integrate the proposed solutions in Kubernetes, so to evaluate them in a real environment.

Moreover, we plan to design novel hierarchical policies that can jointly control the scaling and placement of microservice-based applications in a geo-distributed computing environment.

REFERENCES


COCOA: Cold Start Aware Capacity Planning for Function-as-a-Service Platforms

Alim Ul Gias  
Department of Computing  
Imperial College London  
London, UK  
a.gias17@imperial.ac.uk

Giuliano Casale  
Department of Computing  
Imperial College London  
London, UK  
g.casale@imperial.ac.uk

Abstract—Function-as-a-Service (FaaS) has become increasingly popular in the software industry due to the implied cost-savings in event-driven workloads and its synergy with DevOps. To size an on-premise FaaS platform, it is important to estimate the required CPU and memory capacity to serve the expected loads. Given the service-level agreements, it is however challenging to take the cold start issue into account during the sizing process. We have investigated the similarity of this problem with the hit rate improvement problem in Time to Live (TTL) caches and concluded that solutions for TTL cache, although potentially applicable, lead to over-provisioning in FaaS. Thus, we propose a novel approach, COCOA, to solve this issue. COCOA uses a queueing-based approach to assess the effect of cold starts on FaaS response times. It also considers different memory consumption values depending on whether the function is idle or in execution. Using an event-driven FaaS simulator, FaasSim, that we have developed, we show that COCOA can reduce over-provisioning by over 70% under some of the workloads we have considered, while satisfying the service-level agreements.

Index Terms—Function-as-a-service, serverless computing, cold start, sizing, layered queueing network

I. INTRODUCTION

Function-as-a-Service (FaaS) platforms, based on the serverless execution model [1], allow to deploy the codes as individual functions without having to manage the underlying infrastructure. This facilitates DevOps practices [2] by providing more flexibility to each development team and increasing the pace of delivery of code updates. The availability of open source platforms, like OpenFaaS and OpenLambda, has made it possible to install an on-premise FaaS platform, which calls for dedicated sizing and resource allocation methods in order to meeting service-level agreements (SLAs).

FaaS platforms are designed to implement event-driven applications, which react to a change of state as a result of events generated by the environment and execute associated business logic. In a FaaS platform, this logic is termed as a function, which is usually packaged as a container. To reduce resource wastage, FaaS containers are offloaded from the memory given that they remain idle for specific time period. When a new request for an offloaded function arrives, the request is blocked until the function is loaded again. This issue is known as the cold start issue [1], [3].

During the capacity planning process, the cold start issue can pose a significant trade-off between latency and memory allocation optimization. A cold start occurs when a function is invoked while the corresponding container is not yet loaded in memory, which adds a delay in sending the response needed to spin-up the container and the function runtime dependencies. Despite hurting performance, this mechanism aims at reducing memory consumption by offloading functions that are idle for a sufficiently long time. This poses a trade-off between response time SLAs and available memory to support concurrent execution of more functions, which needs to be considered upon sizing an on-premise installation.

To address this issue, we can draw parallels between a FaaS platform and a Time to Live (TTL) cache. Similar to FaaS, a TTL caching system periodically offloads its cached objects. Due to this, the cold start issue resembles the object hit rate improvement problem in a TTL cache, in which we determine the optimal time to keep objects in cache [4]. Thus, analysis methods from TTL cache research, such as characteristic time approximation [5], may be in principle also applicable to FaaS sizing for estimating the required memory capacity. However, from our study we have identified two limitations of such an approach. First, contrary to TTL cache misses, the latency incurred by function cold start times can vary widely. Next, while a large fraction of TTL research considers fixed-sized objects, a function consumes different amount of memory depending on whether it is idle or in execution.

In this paper, we present COCOA (COld start aware Capacity planner fOr function-As-a-service), a sizing method that leverages a stochastic modeling approach based on layered queueing networks (LQN) [6] and M/G/1-type queueing systems for capacity prediction. To consider the effect of cold starts, we have incorporated the probability of experiencing cold starts, by each function, with the LQN model. These probabilities are estimated from an M/M/1/setup/delayedoff model - a variant of M/M/k setup class of models [7] - which we solve using matrix-analytic methods as a special case of M/G/1-type system. Setup models can approximate the cold start probability for a function, taking into account the cold starts. To predict the required capacity, COCOA follows an iterative process. It repeatedly solves the LQN model to find a set of function idle times and a CPU configuration such that the function response times are just below the SLA. To accelerate the searching process, we have designed a parallel algorithm,
where each parallel branch utilizes binary search. Once the idle times and CPU configuration are obtained, COCOA estimates the CPU utilization value for each function. These estimations are integrated with a capacity estimation method for TTL cache [8] to predict the required memory capacity.

Overall, we summarize our contributions as follows:

- We investigate in Section II the similarity between a FaaS platform and TTL cache from the cold start perspective and illustrate that TTL cache analysis, despite promising, is alone insufficient for FaaS capacity estimation.
- We present in Section III an LQN-based performance modeling technique for FaaS platform that captures the effect of cold starts over function response times, correcting the limitations of TTL cache analysis when applied to this setting.
- In Section IV we propose COCOA, a sizing method for on-premise FaaS platforms leveraging our LQN model and demonstrate its effectiveness in reducing resource over-provisioning while ensuring response time SLAs.

Last, in Section V we validate our framework against data from simulation. Sections VI and VII respectively position the work against the state-of-the-art and conclude the paper.

II. SIMILARITY BETWEEN FaaS AND TTL CACHES

A. Analogy

TTL caches in Content Delivery Networks (CDNs) facilitate faster page loading and reduce the load at the origin server. In such caching systems, each cache object is associated with a TTL, after which the object is evicted [4]. If the cache can serve the request for an object, it is termed as a cache hit. The fraction of requests served, for a particular period, is called the hit rate. Longer TTL values can improve the hit rates but are more costly as they require a larger cache size.

Similar to TTL caches, to reduce the number of cold starts, a possible solution, from the point of view of the end user, is to keep the functions in the memory for longer periods. However, this significantly increases the required memory capacity since most of the functions always remain loaded. A way around to this problem is to determine an optimal idle time that ensures a certain degree of availability and reduces the number of cold starts. Consequently, this will help to satisfy the response time constraints.

B. Example

To illustrate the concept, we have developed a discrete-event simulator for a FaaS platform, referred to in the rest of the paper as FaasSim1. Developing FaasSim was necessary since popular performance modeling tools, like JMT [10], cannot model the cases we need to consider for FaaS - the cold starts and modeling both CPU and memory consumption.

In the simulation, we have considered an open workload model where the requests arrive following a Poisson process. To introduce popularity among the functions, meaning their invocation probabilities will be different, we have used the Zipf distribution [11]. The function service times are set such that they are at-most half of the SLA value of 2 seconds, when there is no resource contention. The cold start times are chosen from a recent study on popular FaaS platforms [12] that, apart from the platform, also considered factors like programming languages and deployment sizes, which can affect the magnitude of cold starts.

We have run the simulation in three settings with 16, 32 and 48 functions and observed the effect of different hit rates over the function response times. The function idle times are set by solving (1) for the specific hit rate. This hit rate have also been used to estimate the memory capacity. The hit rate is related to the average runtime memory consumption \( m \) as \( m = \sum_i h_i \theta_i \), where \( \theta_i \) is the memory requirement of each function [8]. We have used this value of \( m \) as the memory capacity and compared it with the actual memory consumption value obtained from FaasSim. The findings from the simulation are presented in Fig. 1.

C. Observations

In Fig. 1a, we plot the response times of each of the 48 functions for different hit rates. We see that even with 95% hit rate, there are response times that violate the SLA. However,

\[
h_i = 1 - e^{-\lambda_i T_i} \tag{1}
\]

1The simulator is available at - https://github.com/alimulgias/FaasSim
for 95% hit rate, more than half of the function response times are much lower than the SLA. This indicates that all the functions do not require the same hit rate to ensure the SLA. In Fig. 1b, we present a comparison between the estimated memory capacity and maximum consumption for 95% hit rate. Although the capacity notably increases with the number of functions, the consumption is less sensitive to it. This is because memory consumption is primarily dependent on the workload parameters. In addition, the consumption is not very high since most of the functions remain idle while resident in memory, which is not considered during the estimation.

From these observations, it is clear that an availability-aware approach is not adequate for optimal capacity estimation that ensures the SLA for response time. Such an approach only considers the volume of cold starts, whereas we also need to consider its effect on the response time. For a particular workload, firstly, we should know the cold start probabilities of the functions for different idle times. Subsequently, depending on these probabilities and the severity of cold starts, we need to approximate the function response times. Thus, we need a performance model incorporating all these factors. The model will also help in fine-grained capacity estimation by providing the resource utilization estimates. In the following section we present our performance model.

III. MODELING COLD STARTS IN FAAS

A. Estimating Cold Start Probabilities

Unlike commercial FaaS platforms, open source platforms, like OpenFaaS, allow concurrent function execution in same container [13]. We focus on this function concurrency approach. We propose to consider, from a modeling standpoint, the function as a server of a queueing model, representing the admission control buffer to the function, and the cold start delay as the initial setup time of the server before beginning service. The functions also have an idle time which is equivalent to the idle server waiting time before it is shut down. Considering these similarities, a cold start may be modeled as a M/M/1/setup/delayedoff model, which is a variant of M/M/k/setup class of models [7]. The M/M/k/setup models consider a setup cost, usually in the form of a time delay, when turning the server on. Its “delayedoff” variant considers an idle time before turning the server off.

Although in [7], the exact solution is provided for an M/M/k/setup/delayedoff model, this applies when the number of servers is $k \geq 2$. However, in our case we need to model each function separately. Thus, we have a function representing a single server, which can be either on or off. To get different performance indices for such a model, we may directly solve its underlying Continuous Time Markov Chain (CTMC).

The CTMC transitions are presented in Fig. 2. Each CTMC state $(i, j)$ has two parameters: $i$ tracks whether the function resides in the memory or not ($i \in \{0, 1\}$), while $j$ tracks the number of jobs ($j \in \mathbb{Z}^+$) in the admission queue to enter service in the function. A transition from $(i, n)$ to $(i, n + 1)$ occurs with rate $\lambda$, transition from $(i, n + 1)$ to $(i, n)$ occurs with rate $\mu$, and transition from $(0, j)$ to $(1, j)$ occurs with rate $\alpha$. These rates describe the mean inter-arrival time ($\frac{1}{\mu}$), mean service time ($\frac{1}{\mu}$) and mean cold start time ($\frac{1}{\alpha}$) respectively. There is a special transition from $(1, 0)$ to the initial state $(0, 0)$ with rate $\beta$, which describes the function idle time ($\frac{1}{\beta}$). In a CTMC, all holding times are considered to be exponentially distributed. However, in a real system the idle time of a function is set to a deterministic value. To address this issue, we can use the method of phases and make this transition Erlang-$k$ distributed with rate $k\beta$. To realize this, we introduce $k - 1$ extra states between $(1, 0)$ and $(0, 0)$. The transitions between all these states occur with a rate $k\beta$. This keeps the mean identical to the original exponential, $\frac{1}{\beta}$, but reduces the variance by $k$ times. Thus, for large enough $k$, the transition will display a behavior close to deterministic.

The effects of cold starts vary depending on the sequence of request arrivals. If a request arrives when the function is being loaded into memory due to a recent request, its response time will be affected to some extent. The severity of the queuing overhead will depend on the residual cold start time of the previous request. However, this does not need to be modeled explicitly thanks to the memoryless property of the exponential distribution. Considering this, as shown in Fig. 2, it is clear that the cold start states are $(0, j)$, $\forall j$. We can calculate the cold start probability of the functions from the stationary distribution ($\pi$) of their CTMC. We indicate with $\pi_{i,j}$ the probability of state $(i, j)$, then the cold start probability is defined as $\sum_{j} \pi_{0,j}$. We can get the stationary distribution by solving the CTMC. This can be done efficiently using the matrix-analytic method, since the CTMC sparsity structure makes it equivalent to a M/G/1-type process [14]. The latter is analyzed using the MAMSolver [15], [16].

B. Predicting Response Time

Solving the CTMC we can get the cold start probabilities for each of the functions. However, our eventual goal is to predict the response time of each function considering the cold starts. For that purpose, beside the cold start probabilities, we need a performance model of the functions, typically running in containers, contending for the CPU. Each of these functions contend for CPU times to execute two types of jobs, the regular tasks when the function is warm and service restarting when the function is cold. To ensure scalability of the model analysis [17], we use LQNs as reference modeling formalism.
The proposed LQN model\(^2\) is presented in Fig. 3. The model has two main building blocks - the tasks and the processors. In LQN models, tasks translate into different system resources, usually the software resources. They carry out different operations which are defined by their entries. The tasks are executed on the processors, which represent the physical entity, like the CPU, that carries out the physical executions. Although each of the functions is a software resource, we defined them by the entries rather than the tasks. The reason behind this choice is twofold. Firstly, it makes the LQN model more compact and manageable. Secondly, it reduces the model solving delay as the number of function increases.

Since each function has two types of jobs, we use two tasks, \textit{ColdPool} and \textit{WarmPool}. The entries in the \textit{ColdPool} define the cold jobs for all the functions. Similarly, the entries in the \textit{WarmPool} define the warm jobs. Since every cold job is followed by a warm job, there is a call from the cold entries to the warm entries. The proportion of cold and warm jobs is controlled by the \textit{Dispatcher} task based on the cold start probabilities. This is done by setting the cold start probability of each function to the call mean value from its \textit{Dispatcher} entry to the \textit{ColdPool} entry. The percentage of calls to each function, based on their popularity, is modeled using the reference task \textit{Client} by setting the percentage value as the call mean from the \textit{Client} entry to the \textit{Dispatcher} entry.

The LQN model requires two parameters, namely the service demands of the activities and the multiplicities of the modeling constructs. The service demand for a job is the total service time across all visits when there is no resource contention. Each function has different service times for its cold and warm jobs. The service times of the cold and warm jobs are defined by the cold start rate (\(\alpha\)) and the service rate (\(\mu\)) respectively. The service demands should be set in the activities of the corresponding entries. The multiplicities translate into different system entities depending on the modeling constructs. The multiplicity of the reference task indicates the number of clients present in the system, considering the system as a closed network \cite{18}. However, we can also consider the system as open like our \textit{FaasSim} simulator. To do so, we have adapted the think time \(Z\) as \(K/\lambda\), where \(K\) and \(\lambda\) represents the total number of clients and the open arrival rate \cite{19}.

The multiplicities of the processors indicate the number of available CPU cores. Since we do not consider the \textit{Dispatcher} as a bottleneck, we assume it executes separately from the functions, on a single CPU core. The multiplicities of the \textit{ColdPool} and \textit{WarmPool} indicate the number of process threads available for the function containers. Container platforms like Docker allow this on a container basis, which means that we can put a limit on how many threads a container can create\(^3\). However, in LQNs entries do not have a multiplicity property, which we are using to model the functions. Thus, in the model, we consider that the functions share two thread pools for cold and warm jobs. This assumption does not significantly affect the performance estimates if the number of threads, in both pools, are sufficiently large to start processing a job immediately.

### C. Model Validation

We have used the LINE modeling language \cite{20} to build our models, which are solved by the LQNS solver \cite{6}. The results are validated with the \textit{FaasSim} simulator. The simulation parameters are presented in Table I. Here, we consider more large-scale settings, compared to Section II, that include up to 128 functions. The function popularity is controlled by different values of the Zipf parameter that are common in cache based studies \cite{21}. We consider all combinations of number of functions (\(N\)), Zipf parameter (\(\eta\)) and arrival rate (\(\lambda\)). For each of those combinations, we have generated 30 models. In each of the models, we have chosen the service (\(\mu\)), cold start (\(\alpha\)) and idle lifetime (\(\beta\)) rates for the functions randomly from the given range. The range for service and cold start rates are same as Section II. The idle lifetime rates are chosen from \cite{12} such that it can trigger cold starts. The service demand of the \textit{Dispatcher} is set to a negligible value since we do not consider it as a bottleneck.

Since cold start affects the response time, we are concerned about how accurately our model captures that affect and estimate the response time of each function. Thus, we have considered

\(^2\)For details about the notation, please see the LQN user manual available at \url{http://www.sce.carleton.ca/rads/lqns/LQNSUserMan-jan13.pdf}

\(^3\)Such limits are put to prevent unnecessary thread creation causing memory leaks. However, the limits are never too small to affect the concurrency.
TABLE II
PERCENT ERROR IN ESTIMATING THE RESPONSE TIME OF EACH FUNCTION—ACROSS ALL THE ZIPF PARAMETERS IN TABLE I

<table>
<thead>
<tr>
<th>N</th>
<th>( \lambda = 0.2 ) max avg 95p</th>
<th>( \lambda = 0.5 ) max avg 95p</th>
<th>( \lambda = 0.8 ) max avg 95p</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>1.24 2.08 3.38</td>
<td>1.07 1.57 2.2</td>
<td>0.91 1.48 1.65</td>
</tr>
<tr>
<td>32</td>
<td>1.25 2.01 2.3</td>
<td>1.17 1.57 2.21</td>
<td>1.15 1.75 2.16</td>
</tr>
<tr>
<td>64</td>
<td>0.85 1.7 2.06</td>
<td>1.21 1.57 2.15</td>
<td>1.27 1.72 2.09</td>
</tr>
<tr>
<td>96</td>
<td>0.82 1.36 1.98</td>
<td>1.11 1.57 1.83</td>
<td>1.25 1.7 2.05</td>
</tr>
<tr>
<td>128</td>
<td>0.92 1.31 1.71</td>
<td>1.11 1.51 1.95</td>
<td>1.29 1.7 2.1</td>
</tr>
</tbody>
</table>

Fig. 4. An overview of the COCOA approach

the percent error in estimating the function response times. We present the results in Table II. From the table, we see that the increase of the number of functions has a negligible effect over the error. The maximum error across all the parameters is 2.38%. The maximum average error and 95\textsuperscript{th} percentile of the error is 1.29% and 2.08% respectively. Such errors are not significant and thus we conclude that the LQN model can accurately estimate the response time for each function considering cold starts. We leverage this model in our method, COCOA, which we present in the following section.

IV. COLD START AWARE CAPACITY PLANNING

A. Overview

COCOA provides resource allocation decisions for a FaaS platform in terms of its memory and CPU configurations. It also provides the function idle times that can ensure the SLA given the suggested configuration is applied. These idle times allow to control the magnitude of cold starts, which in turn aids in governing the response times. The configurations can be applied both on the hardware level or on the software level. This means that the memory and CPU constraints can be applied on the physical server or the container platform like Docker. COCOA has multiple components each focusing on a particular tasks and expecting different inputs, which are illustrated in Fig. 4.

Since COCOA is a model-based approach, we need a set of parameters to instantiate its model. Firstly, we need the service demands of the functions. These can be estimated in a test environment, when the functions are being developed, using state-of-the-art methods [22]. The workload parameters like the arrival rate and function popularity can be estimated from performance requirements or historical data. Once the parameters are estimated they are passed to the component LQN Model Generator. The Model Generator also requires the architecture of the FaaS platform, particularly containing the information about how the functions communicate.

Based on these inputs, the CTMC and LQN model is generated and forwarded to the next component, the Optimal Strategy Generator. It utilizes the models to provide memory and CPU configurations. It needs the SLAs for function response times and both the memory requirements when they are idle and in execution. The generator then searches for the idle times, under different CPU configurations, that do not violate the SLA with minimal memory consumption. These idle times are used to estimate the maximum memory consumption, based on which the memory capacity is suggested.

B. Problem Statement

We consider a system of \( N \) functions. These functions are executed on a multi-core CPU with \( C \) cores. Every function \( f_i \) has two different memory usage, one is while in execution (\( \theta_{\text{gen}}^m \)) and the other is while being idle (\( \theta_{\text{off}}^m \)). Considering a request for function \( f_i \), if the function is not in the memory, a cold start occurs. Due to this cold start, a request experiences an extra delay. This extra delay is incurred to load the function in the memory. When a function is loaded in the memory, it is associated with a timeout value \( T_i \). While the function is still in the memory, for each new request, the timeout is reset to the original value. A function is removed from the memory if it reaches the timeout limit.

We focus on two specific costs, the cost of CPU and cost of the memory. We define the per unit CPU and memory cost as \( \tau_c \) and \( \tau_m \), respectively. Thus, the cost for the CPU will be \( B = \tau_c \times C \). The memory cost is calculated based on maximum memory consumption. To estimate this, we incorporate the idea of a different memory usage, when the function is idle, with the estimator for TTL cache [8]. Based on this, given the function CPU utilization is \( \rho_i \), the average memory consumption (\( m \)) may be estimated as \( m = \sum_i h_i(\rho_i \theta_{\text{gen}}^m + (1-\rho_i) \theta_{\text{off}}^m) \).

The system’s memory capacity should be adequate when there is a spike in memory consumption. This occurs when there is a surge in requests within a short period. This increases the memory consumption because more functions starts execution, for which the memory requirement is much higher than being idle. It is sufficient to consider this increase in memory consumption by the functions in execution. Considering the memory consumption by the functions in execution is \( U \), the expectation is \( E[U] = \sum_i h_i \rho_i \theta_{\text{gen}}^m \). We can approximate the maximum consumption as \( v = \kappa E[U] \). The value of \( \kappa \) is calculated, using Markov’s inequality, [23] such that the upper-bound of \( P(U \geq v) \) is a negligible value \( \epsilon \). Based on this, we define the approximation for maximum memory consumption \( m_{\text{max}}^m \) as \( m_{\text{max}}^m = \sum_i h_i(\kappa \rho_i \theta_{\text{gen}}^m + (1-\rho_i) \theta_{\text{off}}^m) \), so that the memory cost may be defined as \( A = \tau_m m_{\text{max}}^m \).

Considering these cost functions \( A \) and \( B \), our objective function, \( z \), is defined in (2). Here, our goal is to find \( T \), a
vector including the idle times \( T_i \) of all the functions, and \( C \), the number of CPU cores, that minimizes a weighted sum of normalized memory and CPU cost.

\[
z = \min_{(T,C)} \omega_A \hat{A} + \omega_B \hat{B}
\]

subject to:

\[
C \leq C_{\text{max}}
\]

\[
W_i(T, C) \leq W^*, \forall i
\]

\[
T \in \mathbb{R}_+, C \in \mathbb{Z}_+
\]

The constraints for the objective function are provided in (3) and (4). The first constraint in (3) is regarding the maximum number of allowed CPU cores. This applies when the CPU constraint is imposed on the software level and the total physical CPU capacity is not accessible. The second constraint in (4) addresses the SLA for response time. Here, \( W_i \) is a function of \( T \) and \( C \) which returns the response time for a platform function \( f_i \). This response time should be less than the limit \( W^* \) mentioned in the SLA.

C. Optimal Strategy Generation

Using the objective function in (2), COCOA suggests an optimal strategy that includes the required memory and CPU capacity and the function idle times \( (T) \). It starts searching for an optimal strategy with an initial instance of \( T \). This is obtained by a characteristic time approximation technique for CDN cache [5]. It requires to solve \( m = \sum_i h_i \), where \( h_i \) is defined in (1), for a particular value of \( m \). However, as we have a large pool of functions, it is sufficient to estimate a single value \( T^* \) for all the functions instead of approximating \( T_i \in T, \forall i \) [24]. Thus, here we have used a second definition of \( h_i \), replacing \( T_i \) with \( T^* \) in (1).

After the initialization, COCOA fine-tunes the idle times, such that the function response times are just under the SLA limit, to ensure minimal memory consumption. For this, it solves the LQN model in iteration, upon adjusting the idle times, to observe its effect on the response time. The idle times are adjusted using the concept of binary search. It starts with an initial searching interval, \( (0, T^*) \), for each \( T_i \) and reduces the length of the interval by half on each iteration. The endpoints of the intervals are adjusted depending on whether the response time constraint is satisfied or not. The value of \( T_i \) is updated with the midpoint of the searching interval. For this process to work, the initial value, \( T^* \), should be sufficiently large so that there is no cold starts and thus the response time is not affected. For this purpose, we have solved \( m = \sum_i h_i \) by setting \( m \) to a value close to \( N \).

COCOA runs this fine-tuning process for different CPU configurations (CPU cores). Although the number of CPU cores can be any integer, practically we only need to consider some common options, like multiples of 2 with 32 as the limit. This accelerates the analysis process. In addition, for each configuration, this process is run in parallel, making it even faster. For each run, if a \( T \) is found, that does not violate the SLA, it is considered as a candidate solution.

After completing the process, the optimal solution is selected by comparing the memory and CPU cost. Its corresponding CPU configuration and idle times are suggested just the same. However, the memory capacity is suggested by considering the value \( \min(n_{\text{max}}, \sum_i h_i) \) as an upper-bound and calculating the aggregated size of required number of RAM modules.

V. Evaluation

A. Experimental Setup

We have evaluated COCOA using the FaasSim simulator. Here, we consider 64, 96 and 128 functions varying in terms of their popularity (\( \eta \)), service time (\( \frac{1}{2} \)) and cold start time (\( \frac{1}{2} \)) according to the parameters in Table I. However, the idle times (\( \frac{1}{2} \)) from Table I are not used because we want to estimate these with COCOA such that the response times constraints are satisfied with minimal memory and CPU capacity. We have set the functions memory requirement following the limits in AWS Lambda [25]. The percentage (0-1) of idle function memory consumption is considered to be log-normally distributed with a desired mean of 0.2. From the experiments, we aim to answer the following research questions:

- **RQ1:** Can COCOA reduce memory over-provisioning compared to availability-aware approaches?
- **RQ2:** Can COCOA predict the required memory capacity that meets the maximum demand?
- **RQ3:** Can COCOA predict the memory and CPU capacity to satisfy the SLA for response time?

B. Results

To answer RQ1, we have compared COCOA and the availability-aware approach with two hit rates, 0.8 and 0.95. We present the result for a single experiment in Fig. 5. We can see that the required memory capacity estimated by COCOA is much lower than the other two approaches. The results for all the parameters are presented in Table III. In all the cases, the estimates from COCOA is much less compared to the other two approaches. Considering the 95% hit rate, the capacity estimated by COCOA is 51-74% less. The reason is easy to understand - COCOA can take "well-informed" decisions by leveraging its performance model, which is not possible for the availability-aware approaches.

\footnote{We have considered that each of the memory module is 8GB but this is configurable depending on the availability of RAM modules.}
TABLE III
COMPARING THE PREDICTED MEMORY CAPACITY OF DIFFERENT APPROACHES - AVERAGED ACROSS THE ZIPF PARAMETERS FROM TABLE I

<table>
<thead>
<tr>
<th>N</th>
<th>COCOA 80% h.r.</th>
<th>95% h.r.</th>
<th>COCOA 80% h.r.</th>
<th>95% h.r.</th>
<th>COCOA 80% h.r.</th>
<th>95% h.r.</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>32</td>
<td>88</td>
<td>104</td>
<td>45.3</td>
<td>80</td>
<td>93.3</td>
</tr>
<tr>
<td>96</td>
<td>40</td>
<td>128</td>
<td>152</td>
<td>58.7</td>
<td>125.3</td>
<td>146.7</td>
</tr>
<tr>
<td>128</td>
<td>48</td>
<td>157.3</td>
<td>184</td>
<td>58.7</td>
<td>157.3</td>
<td>184</td>
</tr>
</tbody>
</table>

(a) Avg. Memory Consumption
(b) Capacity vs. Consumption

Fig. 6. Comparing the runtime memory consumption obtained from the analytical approximation and simulation for \( \lambda = 0.5 \) and \( \eta = 1.0 \)

Fig. 7. Illustrating that COCOA can meet the maximum memory demand for different percentage of memory consumption while the functions are idle

From Table III, we see that, in two cases, COCOA predicts a higher capacity for \( \lambda = 0.5 \) than \( \lambda = 0.8 \), which is counter-intuitive. This is because we have used a different upper-bound of \( P(U > v) \) to estimate \( \kappa \) for \( \lambda = 0.8 \). For \( \lambda = 0.8 \) it is 0.1 but for \( \lambda = 0.2 \) and 0.5 it is 0.05. The reason is, for high arrival rates, the spike in memory consumption from the expectation is less than low arrival rates. Here, \( \kappa \) is the coefficient to represent this extent. A larger upper-bound of \( P(U \geq v) \) will result in a smaller value of \( \kappa \). So for higher arrival rates, to reduce over-provisioning, \( \kappa \) should be approximated with a larger upper-bound of \( P(U \geq v) \).

To answer RQ2, we have compared the memory consumption values from the simulation with the values from COCOA. From Fig. 6a, we can see that the average memory consumption values from the simulation agrees with analytical approximation from COCOA. From Fig. 6b, we see that the memory capacity also meets the maximum demand. From all the experiments, we observe only 5 cases where there is a memory deficit greater than 0.5 GB with a maximum value of 3.2 GB. We have also done a sensitivity analysis changing the desired mean of percentage of memory consumption by idle functions. We used two settings with 64 and 128 functions with \( \lambda = 0.8 \) and \( \eta = 1.0 \). As seen from Fig. 7, in both cases, COCOA can satisfy the maximum demand.

To answer RQ3, we have investigated the response time of each of the functions. We have seen that across all the parameters, COCOA can ensure the SLA for response time. We present the response time of each function, for a single experiment, in Fig. 8. The SLA in this case is 2 seconds and COCOA satisfies it for all the functions with hardly any variance. On the other hand, even 95% hit rate has violations. The violations are even more, 45% or 57 out of 128 functions, when the SLA is 1.5 seconds.

To illustrate how COCOA ensures the SLA without over-provisioning, we have investigated the hit rates of each function obtained from the simulation. In Fig. 8a and 9a, we have plotted the hit rates, which correspond the experiments from Fig. 8a and 9a. As expected, we see that the hit rates are fixed for 80% and 95% hit rates. However, COCOA adjusts the idle times of the functions such that the hit rates are just sufficient to satisfy the SLA. This reduces the memory consumption when the functions are idle and thus COCOA suggests a much lower memory capacity. For a 2 seconds SLA, the lowest hit rate a function has is 46%. However, COCOA can also increase the hit rates, if required, as seen in Fig. 9b. Here the hit rates for some functions are even higher than 95% to satisfy a stricter SLA of 1.5 seconds.

VI. RELATED WORK

FaaS platforms, leveraging serverless computing, has gained the attention of many researchers. Here, we particularly focus on the works involving cost, resource management or cold
starts as such works are more relevant in our context. From the perspective of cost, researchers have focused on various issues. In [26], the authors present a technique that predicts the cost of function workflows. The authors in [27] propose a method for optimizing the cost of function workflows through function fusion and placement. In [28] the authors have identified different operation regimes that optimizes the cost of both customer and provider. From the perspective of resource management, researchers have mainly focused on runtime CPU allocation considering the QoS [29], [30].

The authors in [1] and [3] are among the firsts to investigate function latency considering cold and warm states. In recent works, researchers are also proposing different solutions to this problem. In [31], the authors have addressed cold starts from the end user perspective and mitigated it by periodically sending low cost service requests. The authors in [32] have pre-initialized resources, like networking elements, and associated them with containers as required. In [33], the authors provisioned containers in advance by leveraging function composition knowledge. The authors in [34] propose a window based approach to load or unload functions by analyzing their invocation patterns. However, none of these works modeled function memory consumption and response time, making them inapplicable in capacity planning.

VII. CONCLUSION AND FUTURE WORK

We have presented COCOA, a cold start aware sizing method for on-premise FaaS platforms. COCOA leverages an LQN model and M/M/k setup models to obtain different performance estimates and consequently, predict the required system capacity. We have illustrated the improvements yielded by COCOA with multiple experiments, showing that COCOA can help in provisioning FaaS systems that satisfy SLAs.

A future research direction could be incorporating burstiness in the workload that triggers more resource intensive actions and dealing with autoscaling scenario where multiple function replicas need to be instantiated.

ACKNOWLEDGMENT

A. Gias is a commonwealth scholar, funded by the UK government. The work of G. Casale is partially supported by RADON, funded by the EC Horizon 2020 research and innovation program under grant agreement No. 825040. The data referenced in this paper is available at https://doi.org/10.5281/zenodo.4046716, released under the CC BY 4.0 license.

REFERENCES

Evaluating the Performance of a State-of-the-Art Group-oriented Encryption Scheme for Dynamic Groups in an IoT Scenario

Thomas Prantl, Peter Ten, Lukas Iffländer, Alexandra Dmitrenko, Samuel Kounev, Christian Krupitzer
University of Würzburg
{thomas.prantl|peter.ten|lukas.ifflaender|alexandra.dmitrenko|samuel.kounev|christian.krupitzer}@uni-wuerzburg.de

Abstract—New emerging technologies, such as autonomous driving, intelligent buildings, and smart cities, are promising to revolutionize user experience and offer new services. The world has to undergo large scale deployment of billions of things — cost-efficient intelligent sensors that will be interconnected into extensive networks and will collect and supply data to intelligent algorithms — to make it happen. To date, however, it is challenging to secure such an infrastructure for many-folds reasons, such as resource constraints of things, large scale deployment, many-to-many communication patterns, and dynamically changing communication groups. All these factors rule out most of the state-of-the-art encryption and key-management techniques.

Group encryption algorithms are well-suitable for many-to-many communication patterns typical for IoT networks, and many of them can deal with dynamic groups. There are, however, very few constructions that could potentially fulfill the computational and storage constraints of IoT devices while providing sufficient scalability for large networks. The promising candidates, such as construction by Nishat et al. [1], were not evaluated using IoT platforms and under constraints typical for IoT networks.

In this paper, we aim to fill this gap and present the evaluation of a state-of-the-art group-oriented encryption scheme by Nishat et al. to identify its applicability to IoT systems. In detail, we provide a measurement workflow, a revised version of the approach, and describe a reproducible hardware testbed. Using this evaluation environment, we analyze the performance of the encryption scheme in a typical IoT scenario from a group member perspective. The results show that all calculation times can be assumed to be constant and are always below 2 seconds. The memory requirement for permanent parameters can also be considered to be constant and are below 8.5 kbit in each case. However, the information that has to be stored temporarily for group updates has turned out to be the bottleneck of the scheme, since their memory requirements increase linearly with the group size.

Index Terms—Security, Group Encryption, Performance Evaluation, Internet of Things

I. INTRODUCTION

In today’s era of digital transformation, we observe the synergy of physical and digital worlds through the integration of billions of things — cost-effective Internet-of-Things (IoT) platforms — into physical objects. This synergy has the potential to improve our daily environment through data collection, analysis, and automation of routine tasks and laying the foundations for additional intelligence in many use cases, such as smart factories, smart cities, smart homes, or smart health. The additional “smartness” comes from many IoT devices collecting and sharing data, and intelligent algorithms processing it.

The number of deployed IoT devices continuously increases, and — according to the estimation of a Gartner report [2] — this year, there will be 20 billion IoT devices connected to the Internet. The amount of data generated and shared by these devices increases immeasurably, considering that all connected IoT devices in 2018 have already generated five quintillion bytes of data [3]. However, not only the pure volume of data increases but also the complexity of communication patterns, which typically involve groups of devices interested in sharing data with constantly changing group membership dynamics. For instance, various types of wearable devices can monitor the health state of humans and send the collected data to patients, health insurance, or doctors [4]. Thereby, it is essential that the users are always in control of their data and can determine who can read it. Unauthorized control to transmitted data can be achieved through encryption so that only authorized parties that have a corresponding decryption key can decrypt it. As the amount of shared data with different groups of people will grow enormously in the future, it will become more and more important that the corresponding encryption methods work as efficiently as possible. Especially in the context of IoT systems, where IoT devices are limited in storage capacity, have little computing power, and often have only a limited power supply, efficiency becomes a key factor.

Many group encryption methods emerged requiring only one encryption for all the (e.g. [5], [6]) to boost efficiency in systems that feature one-to-many or many-to-many communication patterns. These schemes enable the efficient encryption of messages for fixed groups but become costly when modifying the group’s members. The necessary modifications caused by group membership changes often affect not only a single group member but several or even all group members [1]. Accordingly, for scenarios with dynamic groups, such as IoT, these modifications must be as efficient as possible, since they may occur very frequently. A new promising method was presented in [1] to deal efficiently with dynamic groups, which enables efficient group changes with constant overhead, in contrast to existing schemes [1] that typically feature more
significant overhead (linear or logarithmic).

In this work, we complement [1] through an evaluation of the approach in an IoT setting with real devices. In [1], the authors focused on computation time. We further analyze the efficiency of the scheme concerning energy consumption, energy efficiency, and data overhead. Especially for the last aspect, we focus on real values in terms of bytes rather than providing estimation using the Landau notation, as done on the original publication.

Specifically, this paper provides the following contributions:

- the description of a workflow, how to use the scheme in practice,
- a revised version of the approach from [1] eliminating an error in determining the sent information size in case of group changes,
- the design of a reproducible hardware testbed for the performance measurements of the scheme in an IoT scenario,
- the definition of performance metrics, including error measures, to rate the performance of the scheme, and
- an analysis of the general performance of the scheme using our testbed.

The remainder of this paper is structured as follows. In Section II, we present the scheme [1] in detail. In Section III, we highlight an error in the original publication, which affects the amount of transferred information in case of group changes.

In the course of this section, we introduce the most important components and processes of the scheme developed by Nishat et al. [1]. Specifically, we present (i) the involved actors, including their tasks and connections, (ii) the initial creation of a group consisting of $n$ members, (iii) the addition and (iv) the revocation of members from this group, and (v) the encryption and decryption of messages by group members. Figure 1 illustrates the approach.

As we already mentioned, the original paper has an error, which we will explain in more detail in Section III. Here, we use the corrected notation. We also describe a procedure for the use of the scheme, which includes practical recommendations such as updating parameters only once after a group change.

(1) Involved Actors: Those consist of the actual group members, who should be able to communicate securely with each other, and a central instance (CI), which is responsible for creating and managing groups. In general, each member of the group could take over this rule, but in the following, we assume a dedicated CI. In a smart home scenario, for example, the corresponding smart devices could form a group, and the home server could act as the CI. Since the CI manages the group memberships, it generates the corresponding keys and parameters for decryption and encryption and distributes them to the group members and can read all group messages. Thus, the CI must be a trustworthy party. Secure communication must be available once between the CI and each group member for the inclusion in a group. For subsequent communication between a group member and the CI, an insecure connection suffices as long as it guarantees all messages reaching their destination without artificial delay. Otherwise, it could happen that, e.g., when removing a member, another member misses this updated and continues to encrypt messages so that the excluded member can read them. For the rest of this paper, we assume that no messages are delayed or intercepted. For the sake of simplicity, we assume an integrated group management approach, i.e., the CI knows all initial group members, and when to add or remove members.

(2) Initial Group Creation: Figure 1a shows the initial group creation, which requires the CI to determine the master secret key $MSK$, the system parameter $\Gamma$ and mathematical implementation details $I$. These parameters are used by the CI to determine the corresponding encryption and decryption information (namely the public key $PK$ and respectively the public parameter $\gamma$ and the secret keys $S_i$) used by the group members.

For creating a new group, the CI first determines a prime number $p$, to generate two cyclic groups $G_1$ and $G_2$, each of order $p$. Thereby, $g$ is the generator of $G_1$. Then the CI randomly chooses $\alpha, \beta \in \mathbb{Z}_p^*$ and $h \in G_1$. These values already determine the $MSK \in (\alpha, \beta)$ and the system parameter $\Gamma = (h, g, g^\alpha, g^\beta)$. Next, the CI determines the public key $PK$ and the secret keys of the $n \in \mathbb{N}$ initial group members. For this purpose, the CI first assigns each initial group member a prime number $r_i \in \mathbb{Z}_p^*$. Thereby, $r_i$ is the assigned prime number of the $i$-th group member. These prime numbers are now used by the CI to select randomly $k \in \mathbb{Z}_p^*$, such that $k < \min\{r_1, ..., r_n\}$. Using the above-introduced parameters and Equations 1 and 2, the CI can now calculate $PK$ and the secret keys of the initial group members. Equation 2 shows the calculation of the secret key $S_i$ of the $i$-th user.

$$PK = \left( g^{\alpha r_i} \right)_{PK_1}, \left( g^{\beta r_i} \right)_{PK_2} $$

$$S_i = \left( h^{r_i} \right)_{s_1}, \left( g^{\alpha r_i} \right)_{s_2}, \left( g^{\beta r_i} \right)_{s_3}, \left( h^{\alpha r_i} \right)_{s_4}, \left( g^{\beta r_i} \right)_{s_5}, \left( gh^{\gamma r_i} \right)_{s_6}, \left( r_i \right)_{s_7} $$

The public parameter $\gamma$ results from using equation 3, for which $a \in \mathbb{Z}_p^*$ must be chosen randomly.

$$\gamma = (a * r_1 * ... * r_n) - k$$

In accordance with [1], a bilinear map $e : G_1 \times G_1 \rightarrow G_2$ must also be defined by the CI, which is used by all group members for later encryption and decryption. The bilinear map $e$ and the parameters $g$ and $h$ form the mathematical implementation information $I$. Thereby, $I$ is required by the group members for encryption and decryption.

CI must send the members their respective secret key $S_i$ and the mathematical implementation details $I$ via a secure
channel to complete the group creation. The CI must also send
\( \gamma \) and \( PK \) to all group members, but this does not require a
secure channel. Using Equation 4, 5 and 6, the individual
group members can then calculate \( k, \zeta_{i_1}, \) and \( \zeta_{i_2} \), which they
require for decryption.

\[
\begin{align*}
 k &= (r_i - \gamma) \mod r_i \\
 \zeta_{i_1} &= s_{i_1} s_{i_2}^k \\
 \zeta_{i_2} &= s_{i_1} s_{i_2}^{k_i} \\
\end{align*}
\]

(3) Addition of Group Members: As shown in Figure 1b,
adding group members requires the CI to provide a secret key
\( S_{n+1} \) for the new member and to update the \( PK \) and \( \gamma \) for all
group members, so that they can recalculate \( k, \zeta_{i_1}, \) and \( \zeta_{i_2} \).
Thereby, updating \( PK \) is necessary to encrypt messages to the
new group while updating \( \gamma \) is necessary to decrypt messages
in the new group. \( S_{n+1} \) is calculated analogously as for the old
group members using Equation 2, which requires the selection
of a random prime number \( r_{n+1} \in \mathbb{Z}_p \). The parameters \( \gamma \) and \( PK \) are updated according to Equation 7 and Equation 8,
wherefore a new random \( a' \in \mathbb{Z}_p \) and \( k' \in \mathbb{Z}_p \), where
\( k' < \min\{r_1, ..., r_n, r_{n+1}\} \), must be selected.

\[
\begin{align*}
 \gamma' &= (\gamma + k) \ast a'^{-1} \ast a' \ast r_i - k' \\
 PK &= (g^{a'k'} \ast g^{b'k'}) \\
\end{align*}
\]

(4) Revocation of Group Members: As shown in Figure 1c,
revoking group members requires the CI to update both the
\( PK \) and \( \gamma \), so that all remaining members can recalculate
\( k, \zeta_{i_1}, \) and \( \zeta_{i_2} \). This is done by means of Equation 9 and
Equation 10, for which a random \( a' \in \mathbb{Z}_p \) and \( k' \in \mathbb{Z}_p \), where
\( k' \) is smaller than each \( r_i \) of the remaining members, must be
selected. The factor \( r_i \) in Equation 9 is the assigned prime
number of the group member who should be excluded.

\[
\begin{align*}
 \gamma' &= (\gamma + k) \ast a'^{-1} \ast a' \ast (r_R)^{-1} - k' \\
 PK &= (g^{a'k'} \ast g^{b'k'}) \\
\end{align*}
\]

IV. THEORETICAL EVALUATION

While implementing the procedure proposed by Nishat et
al. [1], we noticed a small notation error in the calculation of
the public parameter. In this section, we show (i) this error
by presenting a counterexample, (ii) how this error can be
corrected, and (iii) how it changes the theoretical performance
of the method.

The error in the original publication [1] occurs when
calculating \( \gamma \), which each member uses to calculate back \( k \).
Equation 13 shows the original equation from [1]. Consider the
following counterexample to illustrate the error: We choose
\( p \) as 13, \( a \) as 3, \( r_1 \) as 5, \( r_2 \) as 7, and \( k \) as 2. In this case, the CI
would calculate \( \gamma = 12 \) (with accordance to the Equation 14)
and distribute it to the members. If the first group member tries
to calculate \( k \), he gets the value 3 (cf. Equation 15), instead
of 2. Thus the transmission of \( k \) by sending \( \gamma \) to the group
members did not work.

\[
\begin{align*}
 \gamma &= a \times r_1 \times ... \times r_n - k \\
 \gamma &= (3 \ast 5 \ast 7 \mod 13 - 2) \mod 13 = 12 \\
 k &= (5 - 12 \mod 5) \mod 13 = 3 \\
\end{align*}
\]

This error originates from Equation 1 in the original
publication, which assumes that
\( (a \times r_1 \times \ldots \times r_n, \mod r_i) \)
equals 0, which is only the case when using the “ordinary” multiplication * and not the group operation ×. Therefore, it should be possible to correct this error by using * instead of ×.

In the original publication, the authors estimated that the size of \( \gamma \in O(1) \), since \( \gamma \in Z_p^* \). To correct this estimation, we investigate the corrected Equation 3. For this equation, we first analyze the size of the minuend and subtrahend individually. Since the subtrahend \( \in Z_p^* \) its size is \( \in O(1) \). The minuend is the product of \( n + 1 \) positive numbers, all smaller than \( p \), whereby \( n \) is the number of group members. Therefore, the size of the minuend is \( \in O(n) \). Thus, the size of \( \gamma \) is \( \in O(n) \) in total.

The remaining theoretical estimates of the original paper are not affected by the changes made, as for them only the size of the affected parameter changes, but not the general approach. Nevertheless, we refuted the claim that the scheme presented in [1] only has fixed costs, since the size of \( \gamma \) grows linearly with the group size.

IV. PROPOSED EVALUATION ENVIRONMENT

As not all costs are theoretically constant anymore as claimed by the original publication [1] — because the size of \( \gamma \) is \( \in O(n) \) — it is even more critical to determine the costs for an application of the approach in practice. The scheme [1] no longer supports arbitrarily large groups — since the group creation requires the temporary storing of \( \gamma \), which grows linearly with the group size. However, it could still support group sizes that may be sufficiently large for many practical applications in the IoT context. For this reason, we analyze the performance of the scheme in a suitable environment. The procedure focuses on large groups perfectly suiting the requirements of IoT systems with increasing group sizes. Hence, we carry out our analysis in an IoT typical environment. Thereby, we assume that small low-power IoT devices form the members of a group, and a powerful PC or server acts as the CI. In terms of performance analysis, we focus on four specific aspects relevant in the context of resource-scarce IoT devices: storage requirements, computation time, power consumption, and energy efficiency. Throughout this section, we introduce (i) the measurement setup used for performance measurements, (ii) the used workload patterns, and (iii) the applied metrics to evaluate performance.

A. Workload Pattern

Since we are interested in the performance of the scheme on a typical IoT device, all workload patterns describe the behavior of an IoT device. In the following, we describe four different workload patterns, which describe the different scenarios of an IoT device we take into account for evaluation.

1. **Encryption** and 2. **Decryption**: The IoT device encrypts/decrypts a message consisting of \( B \) bytes \( N \) times in a row. Thus, a total of \( B \times N \) bytes are encrypted/decrypted.

3. **Updating Decryption Information**: The IoT device re-calculates the parameters \( k, \zeta_1 \), and \( \zeta_2 \) from a stored fixed \( \gamma \) in total \( N \) times.

4. **IDLE**: The IoT device is running in an idle state.

B. Metrics

Since IoT devices typically have not only limited hardware resources but also limited power supply, it is especially critical to use the available energy as efficiently as possible. Therefore, we use power consumption and energy efficiency, among other things, as comparative metrics to evaluate the measurement results.

For power consumption, we introduce the abbreviation \( W \) and define it in Equation 16 as the average power consumption per second. In this equation, \( n \) stands for the duration of the measurement in seconds and \( W_i \) for the power consumption during the \( i \)-th second. In Equation 17, we determine the accuracy of \( W \) using Gaussian error propagation, to be able to indicate an error range in which the actual value of \( W \) is located. Thereby \( \Delta W_i \) is the accuracy of the measured power consumption during the \( i \)-th second. All power measurements use a Yokogawa WT310, whose power measurement error is \( \pm(0.1\% \text{ of reading} + 0.2\% \text{ of range}) \), according to the manufacturer [7]. The range error in our case is 0.0006 Watt because we have set the measuring ranges to 3V and 100mA. These considerations result in the final calculation of \( \Delta W \) according to Equation 18.

\[
W = \frac{1}{n} \sum_{i=1}^{n} W_i \quad \text{(16)}
\]

\[
\Delta W = \frac{1}{n} \sqrt{\sum_{i=1}^{n} \Delta W_i^2} \quad \text{(17)}
\]

\[
\Delta W = \frac{1}{n} \sqrt{\sum_{i=1}^{n} (0.1\% \times W_i + 0.0006 \times W)^2} \quad \text{(18)}
\]

Under SPEC specifications [8], we define in Equation 19 the energy efficiency \( E \) as the ratio of the throughput \( T \) (see Equation 21) to the power consumption \( W \). We calculate the accuracy of the energy efficiency again using Gaussian error propagation, see Equation 20.

\[
E = \frac{\text{Throughput}}{\text{Power Consumption}} = \frac{T}{W} \quad \text{(19)}
\]

\[
\Delta E = \sqrt{\frac{\Delta T^2}{W^2} + \frac{T^2 \times \Delta W^2}{W^4}} \quad \text{(20)}
\]

As throughput, we consider the operations performed during a specified period \( t \). We distinguish several operations: (i) Encrypting \( B_e \) bytes, (ii) decrypting \( B_d \) bytes. For example, the throughput for the encryption of messages alone results from Equation 21. The throughput accuracy is obtained by Gaussian error propagation according to Equation 22. Thereby we assume that each encryption process has been successful. Thereby, \( \Delta t \) is the accuracy of the observed time frame \( t \).

\[
T_e = \frac{B_e}{t} \quad \text{(21)}
\]

\[
\Delta T_e = \frac{B_e \times \Delta t}{t^2} \quad \text{(22)}
\]
The throughput for decryption can be calculated analogously by exchanging the index of $B_e$ accordingly. A third operation, not yet mentioned, is to perform group update operations. The corresponding throughput can also be calculated analogously but in this case $B_e$ must be replaced by the number of the performed group update operations $N_u$. Since the update process for the encryption information consists only of storing $PK$, we only consider the update process of the decryption information for group updates here and in the following.

In addition to energy efficiency and power consumption, we also consider as metrics the average time it takes to (i) encrypt messages $t_c$, (ii) decrypt messages $t_d$, and (iii) perform group update operations $t_u$. $t_c$ can be calculated using Equation 23, where $n$ stands for the number of encryption operations performed and $t_{c_i}$ for the time of the $i$-th encryption operation. The error can be determined by Gaussian error propagation using Equation 24. Here, $t_{c_i}$ stands for the accuracy of the determination of the time duration for the $i$-th encryption process. $t_d$ and $t_u$ can be calculated analogously.

$$t_c = \frac{1}{n} \sum_{i=1}^{n} t_{c_i} \quad (23)$$
$$\Delta t_c = \frac{1}{n} \sqrt{\sum_{i=1}^{n} \Delta t_{c_i}^2} \quad (24)$$

Besides the metrics mentioned so far, we also consider the average size of the ciphertexts $S_c$, the parameters to be stored permanently for encryption $S_{p,e}$ or decryption $S_{p,d}$ and temporarily for group updates $S_t$. Here, $S_{p,e}$ consists of the required memory for $PK$, and $S_{p,d}$ for the required memory for $S_t$, $\zeta_1$, and $\zeta_2$. $S_t$ consists of the required memory to store $\gamma$. Since we can accurately determine the required storage space, we assume the standard deviation of the measured sizes as an error.

C. Measurement Setup

The objective of this paper is to analyze the performance of the group-oriented encryption scheme for dynamic groups from [1] in an IoT context. Hence, we apply an evaluation in an IoT-typical scenario using microcontrollers as hardware for the group members. As a typical example of IoT hardware, we choose the ESP32 microcontroller, since it is wide-spread and used in various IoT systems, for example, automated solar water pumping systems [9], smart surveillance [10], or smart saline level monitoring [11]. The ESP32 is a 32-bit microcontroller from Espressif Systems and a so-called System-on-a-Chip. It has a 240 MHz dual-core CPU, 512 kB RAM, and the ability to establish a 2.4 GHz Wi-Fi connection. For ease of use, we used the developer board version of the ESP32 because it can be connected conveniently via USB, allowing for conveniently flashing and reading out the serial pin of the ESP32 using a computer. For simplicity, we see the serial pin as an external interface to the console output of the ESP32.

In the following, we explain the energy efficiency measurement setup used in the scenario of encrypting messages (workload pattern (1)) and then show how we modified it for the other scenarios.

Figure 2 illustrates the setup for power measurements. In this setup, we use the Elegoo Power Supply Module 1PC to power the ESP32, since it can directly provide the required 3.3V for the ESP32. We use a Yokogawa WT310 power meter to measure the power consumption of the ESP32.

For implementing the actual schema, we require a library that allows the calculation of bilinear mappings. For this purpose, we decided to use the pbc library in version 0.5.14, because it is one of the few standard libraries for this purpose and is present in other applications like Boneh-Lynn-Shacham short signatures or Hess identity-based signatures [12]. Since the pbc library bases on the gmp library, we also use the gmp library in version 6.1.2 [13]. As a concrete pairing, we used the Type A pairings from the pbc library.

For measurement setups for workloads that do not require power measurements, the ESP32 was connected directly to the laptop via USB for convenient access to its serial port. Besides, we measured all metrics describing only the required storage space of specific parameters directly on the laptop performing the corresponding necessary group operations.

V. Practical Evaluation

In the course of this section, we first evaluate the measured memory requirements and then present the calculation times and energy efficiency in more detail.

A. Storage Analysis

We performed all storage measurements on a PC, as we are only interested in the particular memory requirements that are platform-independent. For this reason, the measurements have been made on the PC to allow more measurements.

A group member must permanently store the parameter $S_{p,e}$, consisting of $PK$ to encrypt messages and respectively $S_{p,d}$ to decrypt messages, consisting of $\zeta_1$, $\zeta_2$ and $S_l$. Thereby for storing $S_{p,e}$ exactly 2048 bits of memory are required and for $S_{p,d}$ a total of 6302 ± 2 bits (rounded up to whole bits). Furthermore, the required memory space for $S_{p,e}$ has the fluctuation of 2 bits due to the non-static memory requirement of $S_{l,i}$. The corresponding memory requirements were analyzed based on 1000 groups consisting of only one member since the observed parameters are independent of the actual group size.
A group member must temporarily store the information $S_t$ consisting of $\gamma$ to initially join a group or to perform group update operations. Thereby, the amount of memory required for storing $\gamma$ depends on the size of the group, as illustrated in Figure 3. The size of $\gamma$ for a concrete group size $s_g$ was determined by analyzing the size of $\gamma$ for 1000 groups of size $s_g$. Figure 3 shows practically that the theoretical assumption from Section III, that the size of $\gamma$ increases linearly with group size, is correct in the considered measuring range. The practical evaluation also allows the statement that on average $\gamma$ increases by 159 bits (rounded up) per group member in the considered measuring range.

As the last parameter regarding memory requirements, we analyzed the size of ciphertexts. Thereby we have encrypted messages consisting of up to 50 bytes and received within this range, independent of the concrete chosen message size, fixed ciphertext sites of precisely 3072 bits. The reason for the chosen measuring range is that the majority of messages in IoT are small and often less than 40 bytes [14]. At this point, it is noteworthy that constant chipper texts are not a matter of course in group encryption methods, and their length is often related to the number of group members, for example, in [6].

Overall, the evaluation of the considered memory requirements shows that the memory requirements, in the considered measuring range, are to be regarded as constant, apart from the linearly increasing memory requirement of $\gamma$.

### B. Computation Time Analysis

In the following, we present the measured calculation times for the pure encryption $t_e$ and decryption $t_d$ and the execution of group update operations $t_u$. We performed the corresponding workloads (1), (2), and (3) on the ESP32.

We measured $t_e$ and $t_d$ for different message sizes, which is illustrated in Figure 4. Thereby the measuring range was selected as before. The measurement values illustrated in Figure 4 allow us to conclude that for the considered measurement range (i) it is acceptable to regard the times for encrypting and decrypting as constant, (ii) that decryption needs $1.389 \pm 0.004$ seconds and encryption needs $0.569 \pm 0.058$ seconds, and (iii) that decrypting messages takes more than twice as long as encrypting messages.

In addition to the times required for encryption and decryption, we measured the times required by the ESP32 to perform group update operations. These measured times are shown in Figure 5 for group sizes between 50 and 550. The measurement range was limited because too large group sizes require larger contiguous memory areas than the ESP32 with the used libraries, and our implementation could not provide. Therefore, we decided to start with a group size of 50 measurements and to increase it in steps of 50 as long as the group update operation was successful. Figure 5 allows the conclusion that it is acceptable for the considered measuring range, to assume that the execution of group operations requires constant time independent of the group size. Thereby $0.669 \pm 0.039$ seconds are to be expected for the execution of an update operation.

Overall, our measurements confirm the assumption of constant calculation costs in the measuring range under consideration.

### C. Energy Efficiency Analysis

In the following, we present the measured energy efficiency and power consumption of the scheme. We executed all measurements on the ESP32.

The general energy efficiency of the encryption and decryption procedures are determined using the workload patterns (1) and (2), and the corresponding energy efficiency values are illustrated in Figure 6. The measuring range was selected analogously to the previous subsection. Figure 6 allows us to derive two statements for the considered measuring range. First, decrypting messages is more energy consuming than encrypting messages. One might have already expected this statement from the different calculation times for decryption and encryption. However, different execution times alone would not have allowed this conclusion since, in general, a program that requires more time does not automatically require
more energy [15]. Second, the energy efficiency of encryption and decryption increases linearly with the message length. For this reason, from an energy point of view, it does not bring any benefits in practice to, e.g., divide a message into two messages and to encrypt and decrypt them separately.

In addition to decryption and encryption, we also measured the energy efficiency for group update operations separately on the ESP32 using workload (3). The measuring range was again selected analogously to the previous section, and the measured efficiency is depicted in Figure 7. This figure allows us to state that for the considered measuring range and within our measuring accuracy, it is acceptable to assume that the energy consumption for group update operations is constant and independent from the actual group size. Also, the figure allows us to make statements about the maximum number of possible group operations that would be possible in this measurement range for a battery-powered ESP32. For example, if the ESP32 would be powered by a CR123 A Lithium battery (which provides 4.65 kJ) [16], then (if we overestimate the group update operations per Joule to be 7.6), not more than 35340 update operations would be possible.

Finally, considering the energy usage of ESP32 in IDLE mode (workload pattern (4)) compared to continuous group updates (workload pattern (3)) with 550 users yields the overhead caused by keeping group information update to date. Since the throughput of IDLE is 0, we take the average energy consumption per second as a measure of comparison. The measurements resulted in 0.20504 ± 0.00028 Joule for IDLE and 0.21803 ± 0.00027 joules for group updates. Within the scope of the measurement accuracy, the overhead creates an extra energy consumption of less than 0.02 Joule.

VI. RELATED WORK

In the domain of encryption procedures for efficient communication with more than one communication partner, many schemes emerged in recent years, e.g., [1], [5], [6], [17]–[19] to name some. The works differ significantly in their performance evaluation — some papers provide theoretical analysis [5], [6], [19], while others offer analysis based on empirical measurements with real hardware [1], [17] or simulated measurements [18]. Theoretically evaluated encryption schemes often originate from the crypto community that commonly use Landau notation in evaluation. While such a comparison helps estimating the complexity of developed algorithms, they are less useful for practitioners. Practical approaches to evaluation, however, often lack unified evaluation scenarios and metrics, which makes it difficult to compare them with one another. For instance, some evaluation scenarios target conventional computers such as desktop PCs or laptops (as in [1]), while others aim at IoT specific platforms and scenarios [17], [18]. We also chose an IoT setting because (i) IoT gradually becomes an integral part of our daily environment, (ii) IoT systems typically represent the target group of the considered encryption schemes by consisting of a large number of communication participants, and (iii) for the resource-limited IoT devices the performance of the methods is much more critical compared to scenarios with powerful computers. The considered performance aspects — such as the required storage space, calculation times, and energy efficiency — are also already used in the literature [17], [18]. However, our evaluation differs in (i) how we power the observed IoT device and (ii) how to determine the efficiency of energy usage. As a common practice in related work, IoT devices run on battery power, and the battery lifetime, battery status and the average current determine energy efficiency (e.g., [17], [18]). However, these approaches some disadvantages. For example, in the battery datasheets (e.g., see the datasheet of a CR123 A Lithium battery [16] used in [17]) the discharge curve typically shows a non-linear behavior, so only the current cannot be used as a measure of the consumed energy. Besides, the use of a battery thus leads to the fluctuating power supply, whose resistance also changes (e.g., for a changing internal battery resistance of a Lithium-Ion battery see [20]) and thus also influences the current. Although battery life is a metric that is of great practical interest, this metric requires additional information to interpret it. Thus, it is essential to know the complete history of the IoT device’s behavior to interpret its battery lifetime.

To avoid the limitations of using battery lifetime and the average current for determining the energy efficiency, we decided to power our chosen IoT device with a constant power supply, as this allows us to determine the energy efficiency in a time stable measurement environment. This approach also allows us to make statements about individual operations and to indicate their actual energy consumption in Joules. That way, we can directly compare the energy consumption of two different operations, like how much energy the encryption of...
a message of 10 Bytes consumes in contrast to the encryption of a 100 Bytes message. In addition to the modified test setup, we have also specified the corresponding error formula for the metrics used to be able to make statements about the accuracy of the values determined.

VII. CONCLUSION

In this paper, we present the evaluation of the state-of-the-art group-oriented encryption scheme from [1]. Therefore, we describe a revised version of the approach, a measurement workflow, and a design of a reproducible hardware testbed. Using this evaluation environment, we analyze the performance of the encryption scheme in a typical IoT scenario. The results show that that encryption, decryption, and group updates need constant time and complete in less than two seconds. In terms of memory requirement for the permanently stored parameters, the storage requirement remains below 8.5 kbit. However, we identify a bottleneck of the scheme since the temporary parameter stored for group updates are increasing linearly with the group size.

For future work, we plan to extend our experiments in terms of network communication and, therefore, combine the scheme with IoT typical communication protocols like MQTT or OPC UA. Further, we already used real hardware in a controlled environment. For future work, it would be interesting to apply experiments in a real-world IoT system as this would integrate additional challenges, such as mobility of devices or fluctuation of the network connection. Additionally, in this paper, we analyzed the encryption scheme from Nishat et al. [1]. For future work, it would be interesting to compare other approaches, such as those mentioned in the related work section: [5], [6], [17]–[19]. Using such a knowledge base of experiments might help to derive general guidelines for the use of group-oriented encryption schemes for IoT systems. Those guidelines allow us to derive the best encryption scheme; moreover, those can be encoded in a self-aware computing approach [21] to automatically choose the best approach dynamically at runtime depending on characteristics of the group as well as the requirements for communication.

ACKNOWLEDGMENT

This research has been funded by the Federal Ministry of Education and Research of Germany in the framework KMU-innovativ - Verbundprojekt: Secure Internet of Things Management Platform - SIMPL (project number 16KIS0852) [22].

REFERENCES

Performance Prediction for Data-driven Workflows on Apache Spark

Andrea Gulino, Arif Canakoglu, Stefano Ceri and Danilo Ardagna
Politecnico di Milano, name.surname@polimi.it.

Abstract—Spark is an in-memory framework for implementing distributed applications of various types. Predicting the execution time of Spark applications is an important but challenging problem that has been tackled in the past few years by several studies; most of them achieving good prediction accuracy on simple applications (e.g. known ML algorithms or SQL-based applications). In this work, we consider complex data-driven workflow applications, in which the execution and data flow can be modeled by Directly Acyclic Graphs (DAGs). Workflows can be made of an arbitrary combination of known tasks, each applying a set of Spark operations to their input data. By adopting a hybrid approach, combining analytical and machine learning (ML) models, trained on small DAGs, we can predict, with good accuracy, the execution time of unseen workflows of higher complexity and size.

We validate our approach through an extensive experimentation on real-world complex applications, comparing different ML models and choices of feature sets.

Index Terms—performance prediction, workflow applications, Spark, machine learning

I. INTRODUCTION

In the past decade, we have witnessed an increasing spread of big data applications in several domains, such as business analytics[1], social media analysis[2], healthcare[3], and natural language processing[4]. Big data applications are characterized by data-intensive and computationally intensive tasks which are typically implemented on top of parallel algorithms and distributed computing frameworks. Among such frameworks, Apache Spark has emerged as the de-facto platform for analytical processing, with broad adoption in both industry and academia, due to its simplicity, fault tolerance, and scalability[4].

At the same time, users and enterprises have started moving their big data applications from traditional local server architectures to cloud computing platforms (e.g. Amazon EC2, Google Cloud, Microsoft Azure1), which provide configurable environments suiting the needs of big data applications. These systems usually offer services at the Platform level, where big data frameworks are already installed, and allow their users to choose among several configurations, e.g. specifying the number of instances in a cluster and their characteristics (CPUs, memory, . . . ). Each choice might drastically affect the application execution time and the monetary cost of using the cloud computing service.

Predicting the performance of an application is therefore useful for a proper allocation of the available resources, aimed at reducing resource wasting and extra costs.

The problem of performance prediction for big data applications on the cloud has been tackled in several studies. Some of them rely on traditional techniques, such as analytical models[5]–[7] and simulation[8]. More recently, machine learning (ML) has been used to predict the performance of large systems[9]–[12]. The idea is to collect training data offline and use ML models, such as regression, to predict the runtime performance of future executions. Those studies mainly differ for the chosen set of features (black-box vs. gray-box approaches), i.e. capturing more or fewer details of the system, and for the applied ML technique (simple regression vs. more complex ML techniques). In [12] authors show how black-box models (specifically Ernest, an approach proposed by Spark inventors [9]) fail when applications exhibit irregular patterns and/or when extrapolating on bigger data set sizes. Despite achieving good results in terms of accuracy, those studies are performed on simple monolithic applications (e.g. static programs, known ML algorithms), which complexity is far from modern data analytics pipelines. Indeed, nowadays, scientific jobs are rather represented as workflow applications that consist of many complex tasks, with logical or data dependencies, that can be implemented on top of several parallel frameworks[13]–[15].

A workflow application can be represented by a Directed Acyclic Graph (DAG), i.e. a directed graph with no cycles, in which vertices and their connecting edges are used to represent, respectively, application tasks and their dependencies. Each task cannot be executed until all its parent tasks have completed their execution and moved their results to their child tasks. Specifically, we target workflow applications implemented on Spark, i.e. workflows in which each task of the workflow applies a set of Spark operations to the task inputs. Moreover, a workflow can be potentially implemented by multiple Spark applications.

A simple way of predicting the execution time of a workflow could consider the workflow as a monolithic application with known inputs. In this case, a ML model should be trained for each feasible workflow, i.e. for each possible combination of tasks that could appear in the workflow. Besides being a solution that does not scale, the complexity of large workflows could be hardly captured by machine learning.

The solution proposed in this paper, instead, builds a separate ML model for each task allowed in a workflow application. The workflow execution time is eventually estimated as a combination of the individual tasks execution time predictions. This type of solution allows training ML...
models on minimal workflows and uses them to predict the performance of unseen workflows of arbitrary complexity. Moreover, since the input data of intermediate tasks is not known offline, we estimate their characteristics (profiles) by mixing ML and analytical models.

The overall solution is a hybrid approach to estimate the execution time of arbitrary complex workflow applications based on Spark. To the best of our knowledge, no previous work focused on performance prediction for Spark applications presenting this level of complexity and no similar approaches, explicitly tackling the problem of intermediate result estimation, were used for workflow performance prediction.

We validate our approach on a real-world complex system, comparing different ML techniques and choices of feature sets (black-box vs. gray-box). Eventually, an extrapolation analysis compares the robustness of different ML models against the variation of the input data size and of the cluster computational power.

This paper is organized as follows: in Section II we describe data-driven workflow applications and introduce the real-world system used in our experimental evaluation; in Section III we describe our three-phase approach for performance prediction, validating the approach in section IV. A discussion of related literature proposals is reported in Section V. Conclusions are finally drawn in Section VI.

II. DATA-DRIVEN WORKFLOW APPLICATIONS

A data-driven workflow application can be represented as a Directly Acyclyc Graph (DAG), i.e. a directed graph with no cycles. Each vertex in this DAG represents a task, while edges represent the data and control dependencies between tasks. A task is executed only when all its input data have been computed, i.e. when all parent tasks have completed their execution. Formally, a DAG representing a workflow application can be described as a tuple $G = (V, E)$, where $V$ is the set of vertices (tasks), $E$ the set of directed edges (dependencies) s.t. $E \subseteq V^2$. We call entry task (exit task) a task with no incoming (outgoing) edges. For simplicity, we assume that workflows have a unique exit task. While entry tasks represent the reading of the workflow input data, the exit task stores back its final result. We further assume that each task produces a single output, which can be the input to multiple child tasks (through multiple edges). Moreover, we characterized each task defining:

- **Output Profile**: a set of features quantitatively describing its result.
- **Task Arguments**: any input parameter given to the task, in addition to the input data produced by its parent tasks. Examples can be strings, flags or other options that might change the behaviour of the task.
- **Environment Parameters**: a set of features describing the execution environment in which the task will run. Examples are the number of cores and amount of memory of a cluster.
- **Execution Time**: time required for processing the input data and produce the task result.

Once a query is submitted, the ScQL processing system translates the query into a ScQL-Workflow, similar to the one depicted on the left side of Fig. 1. Each task of this workflow applies a set of Spark operations on input data to implement the semantics of the corresponding ScQL operator. In this specific example, the entire workflow corresponds to a single Spark application, which DAG structure is depicted on the right side of Fig. 1.

In the ScQL Data Model, an interval-dataset $D$ is made of several files (which in memory are called samples). Each entry in a file represents an interval, by means of its start and stop values. Although ScQL includes several operators, in our experimental evaluation we will only mention two of its most complex and peculiar operators, namely ScQL-Map and ScQL-Join, that are here briefly described:

- **ScQL Map**: given two input datasets, namely a reference and an experiment dataset, this operator computes, for each interval in the reference dataset, aggregates on the overlapping experiment intervals.

- **ScQL Join**: given two input datasets, namely a reference and an experiment dataset, this operator computes, for each couple of reference-experiment files, the couples of overlapping reference-experiment intervals. A distance parameter (dist) can be provided to match also experiment intervals which are at a maximum distance from a reference interval.

Lastly, we mention a partitioning technique called interval-binning [18]. The bin-size parameter determines the amount

![Image](image-url)
of intervals that end up in the same partition. Depending on this parameter, some intervals of the initial dataset might be replicated in several partitions.

III. WORKFLOW PERFORMANCE PREDICTION

Our three-phase solution for predicting the execution time of a workflow application performs the following three steps:

1) Intermediate profile estimation: for each task in the workflow, its output profile is estimated. This implies that, at the end of this phase, the input data profile of every task in the workflow will be available. This phase is detailed in Section III-C.

2) Task execution time prediction: for each task of the workflow, its execution time is predicted. This is discussed in section III-A.

3) Workflow execution time prediction: the overall execution time is computed by combining the predictions of the individual tasks execution times. Generally, the workflow execution time depends on the underlying engine and its scheduling algorithm. In this work we target workflows that are mapped to Spark applications. By assuming that: i) every task of a workflow is executed within the same Spark context; ii) every workflow task corresponds to a number of Spark tasks that is greater or equal to the number of available cores (in other words, assuming that each workflow task will keep busy all the available cores) then we can reasonably approximate the workflow execution time as the sum of the individual tasks execution times. The validity of these assumptions is proven by our experimental results (see Section IV).

A. Task execution time prediction

The goal of task execution time prediction is to build, for each different type of task that can be present in a workflow, a machine learning model that, with adequate accuracy, is able to predict the task execution time. A ML model is trained on a set of features characterizing the task, such as a quantitative description of its input data, its arguments, and features describing the execution environment, with the objective of predicting its execution time.

Identifying and extracting a complete set of features is hard to achieve in this context. First of all, the knowledge on the application and on the execution environment might be limited; secondly, extracting some features might be non-trivial and time-consuming, hence not convenient. Third, a super-set of features usually leads to overfitting.

State-of-the-art ML solutions for performance prediction are mostly based on simple feature sets that provide an high-level description of the input data and of the execution environment. These are usually called black-box features, e.g. the input size and the number of cores can be considered a complete set of black-box features for the performance prediction of a generic parallel application. Although it has been proven that black-box features can be powerful enough to get good prediction accuracy (e.g. in Ernest [9]), choosing a simple feature set might become limiting when trying to describe the performance of complex workflows.

In this work, we considered different feature sets, showing how different choices can impact the prediction accuracy of the produced models. Without constraining the choice of features, we propose a categorization of the possible feature sets. The first distinction we make is between:

- **Black-box** features, that do not require detailed knowledge on the application, data model and execution environment.
- **Gray-box** features, that extend black-box features with application, data-model and environment-specific features. We assume that gray-box feature sets include also black-box features.

Orthogonality to the previous distinction, we discriminate among:

- **Basic features**, organized into:
  - **Input Data Profiles**: features quantitatively describing the input data of a task (e.g. data size, number of files, number or entries ...)
  - **Task Arguments**: any input parameter given to the task.
  - **Execution Environment**: a set of features describing the environment in which the task is executed, e.g. number of cores and amount of memory, or more advanced features characterizing the Spark environment.
- **Composite features**. Even though several machine learning methods are able to capture non-linear dependencies on the features, it is sometimes "helpful" to include non-linear combinations of some basic features in the final feature set. For example, Ernest [9] introduced the logarithm of the number of cores, which encodes the cost of reducing operations in parallel frameworks, and the ratio between data size and number of cores, which approximates the time spent in parallel processing. We define composite features as linear or non-linear combinations of basic features.

Given the aforementioned distinctions, we organize the possible feature sets in four categories, summarized in Table I: basic black-box features (BB-Basic); basic and composite black-box features (BB-Full); basic gray-box features (GB-Basic); basic and composite gray-box features (GB-Full).

According to the proposed categorization, Ernest, i.e. the state-of-the-art for Spark application performance prediction, uses a black-box full (BB-Full) feature set. In the experimental evaluation, our choice of features for the BB-Full feature set included all the Ernest features and will be considered a baseline to evaluate the benefits of our approach.

For each category, we defined a set of candidate features on which we applied Sequential Forward Selection (SFS) [19] to remove non-relevant ones.
B. Model Techniques

In the previous works, Venkararaman et al. [9] (Ernest’s authors) and Maros et al. [12] have applied ML to estimate the execution of simple Spark applications. While Ernest is based on linear regression, with coefficients estimation based on non-negative least squares (NNLS), in Maros et al. authors took into account more complex ML techniques such as Decision Tree and Random Forest.

In Section IV we compare the performance of three suitable ML techniques: (i) Linear regression (LR), which produces models that can be easily interpreted, but hardly captures complex interactions between features, (ii) Decision Trees (DT) and (iii) Random Forest (RF), having the ability to capture non-linear relationships, still allowing a good interpretability of the model.

Given a target task type, for which we want to predict the execution time, the training set provided to the chosen machine learning technique is made of several executions of that task. As proven in the experimental section, depending on the feature set and on the model technique, it might not be necessary to run on big inputs and extremely powerful environments. Even running small executions on medium-size clusters can still produce prediction models with good prediction error on larger inputs and clusters.

C. Predicting intermediate input features

In order to perform the task execution time prediction step, all the input features, including the input data profiles, must be available offline. However, this does not hold for intermediate tasks, which input data have not been computed yet. To address this problem, we perform, as the first step, an estimation of all such data profiles.

We assume that entry tasks import data which have already been profiled. This assumption is in general acceptable, since application users typically download a limited number of datasets which are then re-used for several workflow runs and can therefore be profiled once for all.

The goal is to estimate, for every task \( v_i \in V \), which is not the exit task, its output profile \( P_i \). Specifically, an estimation model should be built for each feature in \( P_i \). (e.g. data size, number of entries, etc.).

Similarly to the execution time prediction, the feature set used for this type of estimation includes: i) the input profiles of task \( v_i \), ii) the task arguments. Moreover, the same feature set categorization proposed in section III-A has been applied, reasonably excluding any environment-related feature.

Each feature in \( P_i \) can be estimated:

- **Analytically**: there is a known combination of features of the chosen feature set which allows to exactly compute the output feature.
- **Heuristically**: the exact formula for computing the output feature is unknown or there is not enough information to compute it; in this case, heuristics on the available feature set can be defined.
- **by Machine Learning**: applying ML to estimate the output feature, similarly to what was has been said for task execution time prediction.

The intermediate profile estimation phase estimates each task output profile, starting from entry nodes, which input data profile was computed offline (previous assumption), up to the exit node.

In the experimental section (IV) we compare different ways of predicting the output profiles, showing the advantage of using heuristics/analytical models w.r.t. ML. Even though heuristics and analytical models require more knowledge on the application, they are able to guarantee lower prediction error.

**IV. EXPERIMENTAL EVALUATION**

We applied the proposed performance prediction model to ScQL DAGs, introduced in II-A, focusing on two of its most complex and peculiar operators. In this section we:

- describe the experimental setup (IV-A), the evaluation metric and how hyper-parameter tuning was performed (IV-B)
- assess the accuracy of task execution time prediction for different choices of feature sets, different ML techniques and execution environment configurations (IV-C)
- assess the accuracy of output profile estimation, either using analytical models/heuristics, and ML (IV-D)
- measure the error in predicting the workflow execution time, comparing gray-box models to the state of the art black-box based (IV-E) and validating on workflows that were much more complex than the workflows used for collecting training data.
- present an extrapolation analysis, in which we compare the robustness of different ML models against the increase of the input data size and of the cluster computational power (IV-F).

Specifically, we prove that models trained on small input data (number of cores) are good at predicting execution with larger input data (higher number of cores). This is particularly beneficial, given the nature of big data analyses, in which the size of processed data gets bigger and bigger, as well as the computational resources required for processing.

A. Experimental setup

All the experiments in this section were run on Amazon AWS (EMR service) choosing a variable number of r5d.2xlarge worker nodes (each having 8 vCores, 64 GiB RAM). Specifically, we used clusters with 6,8,10 and 12 worker nodes, running the emr-5.19.1 release with Spark 2.3.2 and the Amazon 2.8.5 Hadoop distribution. Training sets were built running more than 4500 executions, collecting all the information required for execution time prediction and output profile estimation, using an in-house developed profiling solution.

B. Evaluation metric and hyper-parameter tuning

**a-MLLibrary**, an open source library built on top of scikit-learn 0.19.1, was used to test different values for the hyper-

\[^{1}\]https://github.com/eubr-atmosphere/a-MLLibrary
\[^{2}\]https://scikit-learn.org
parameters characterizing each learning methods along the lines of the work in [12]. Several combinations of values for the hyper-parameters were tested, and the best combination, corresponding to the lowest MAPE obtained with 5-fold cross-validation, was selected.

The most frequently used hyper-parameters are reported in tables II and III. Some parameters, e.g. the alpha penalty and the maximum three depth for Random Forest and Decision Tree help to prevent overfitting. Minimum Samples to Split/per Leaf represent, respectively, the minimum number of samples required to split a node and the minimum number of samples required to be a leaf. More details on the hyper-parameters are provided in the scikit-learn documentation. For each feature set category, we first defined a set of candidate features and then applied SFS to exclude non-relevant features.

C. Task execution time prediction

Tables IV and V report the measured MAPE for different ScQL operators, i.e. ScQL-Map (Table IV) and ScQL-Join (Table V). Within each table, we report the prediction error for:

- different choices of the feature set, i.e. black-box vs gray-box and basic vs full, as described in Section III-A;
- different machine learning methods, including DTs, RFs, and LR. We excluded methods for which the best MAPE was higher than 20%, like XGBoost and Neural Networks.

Tables VI and VII, show, respectively, the relevant features for ScQL-Map and ScQL-Join provided by the ML technique giving the lowest MAPE (RF), for a given feature set choice (black-box vs. gray-box, basic vs. composite). The feature called \textit{binned-result-size} represents the (known-by-semantics) result size, accounting for the replication of some intervals due to binning. The definition of this composite feature includes gray-box features such as the \textit{bin-size} and the \textit{average-interval-length}. Similarly applies to the feature called \textit{binned-total-size}, which definition includes also a task parameter distance.

Results show that:

- RF is able to give good predictions even with black-box basic features. Therefore, even without having detailed knowledge on the application and on the environment, and without "helping" the model by defining composite (i.e.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
Hyper-parameter & DT & RF \\
\hline
Max Depth & 5 & 8 \\
Max Features & auto & auto \\
Min samples to split & 4 & 2 \\
Min samples per leaf & 1 & 2 \\
Criterion & MSE & MAE \\
\hline
\end{tabular}
\caption{Most used hyper-parameters values (RF and DT)}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Hyper-parameter & LR \\
\hline
Penalty \(\alpha\) & 0.01 \\
Fit-intercept & True \\
\hline
\end{tabular}
\caption{Most used hyper-parameters values (LR)}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
 & Basic & Full \\
\hline
BB & ref-total-size (0.5) & ref-total-size (0.5) & \textit{input-total-size/cores} (0.3) \\
& exp-num-files (0.3) & exp-num-files (0.3) & \textit{input-total-size/cores} (0.3) \\
& cores (0.2) & cores (0.2) & \textit{input-total-size/cores} (0.3) \\
& ... & ... & ... \\
& \textit{binned-result-size / cores} (0.8) & \textit{binned-result-size} (0.1) & ... \\
& \textit{bin-size} (<0.1) & \textit{bin-size} (<0.1) & ... \\
\hline
GB & ref-total-size (0.5) & \textit{input-total-size} (0.3) & \textit{input-total-size/cores} (0.5) \\
& exp-num-files (0.3) & exp-num-files (0.3) & \textit{input-total-size/cores} (0.5) \\
& ref-num-entries (0.1) & ref-num-entries (0.1) & \textit{input-total-size/cores} (0.5) \\
& cores (0.1) & cores (0.1) & \textit{input-total-size/cores} (0.5) \\
& distance (<0.1) & distance (<0.1) & \textit{input-total-size/cores} (0.5) \\
& \textit{binned-total-size / cores} (0.1) & \textit{binned-total-size} (0.5) \\
& \textit{binned-total-size / cores} (0.1) & \textit{binned-total-size} (0.5) \\
& \textit{binned-total-size} (0.1) & \textit{binned-total-size} (0.5) \\
\hline
\end{tabular}
\caption{SciQL-Map Relevant Features (RF)}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
 & Basic & Full \\
\hline
BB & input-total-size (0.3) & \textit{input-total-size} (0.3) & \textit{input-total-size/cores} (0.5) \\
& exp-num-files (0.3) & exp-num-files (0.3) & \textit{input-total-size/cores} (0.5) \\
& ref-num-entries (0.1) & ref-num-entries (0.1) & \textit{input-total-size/cores} (0.5) \\
& cores (0.1) & cores (0.1) & \textit{input-total-size/cores} (0.5) \\
& \textit{distance} (<0.1) & \textit{distance} (<0.1) & \textit{input-total-size/cores} (0.5) \\
& \textit{binned-total-size / cores} (0.1) & \textit{binned-total-size} (0.5) \\
& \textit{binned-total-size / cores} (0.1) & \textit{binned-total-size} (0.5) \\
& \textit{binned-total-size} (0.1) & \textit{binned-total-size} (0.5) \\
\hline
\end{tabular}
\caption{SciQL-JOIN Relevant Features (RF)}
\end{table}

non-linear) features, it is still possible to get an acceptable error.

- LR performs well only when all the complex features are in place (gray-box), including the user-provided non-linear features (composite).

- Basic gray-box features do not significantly reduce the prediction error unless they are properly combined into composite features.

- The higher complexity of the ScQL-Join operation w.r.t. the ScQL-Map is reflected in the higher prediction error.

D. Intermediate profiles estimation

In Table VIII we report the prediction error (MAPE) for the main features describing the result of a ScQL-Join operation. While the output number of files can be analytically determined, the output total size and the output average interval length cannot be known a priori, since they depend on the number of intersections between input intervals. For those features, we defined two heuristics which give a low prediction error. For instance, the number of output samples (files) of a ScQL-Join operation is simply estimated as the product between the number of samples in the reference dataset and the number of samples in the
experiment dataset. For all the three features we reported the prediction error obtained using ML models (specifically, Decision Tree). Although the output total size is a black box feature, the prediction error reported in the table was obtained using a model which took into account gray box features; using only black-box features we were not able to obtain a MAPE lower than 22%. We do not report ScQL-Map prediction errors since all the output features can be easily estimated, both analytically and by applying ML, with a MAPE lower than 1%.

### E. Workflow execution time prediction

In order to measure the workflow execution time prediction error we automatically generated 168 workflows with different topologies including ScQL-Map and ScQL-Join, together with other mandatory entry/exit tasks which, for simplicity, are not discussed in this paper. Each ScQL workflow was mapped to a Spark Application. On average, each generated Spark application contained 23 jobs, 117 stages and more than 16K tasks per critical stage. Note that those workflows have higher complexity w.r.t. to the workflows used to build the training sets, mostly containing only the task for which the ML model was going to be build. In other words, here we demonstrate the ability of our modular approach to predict the execution time of workflows of unseen complexity, although minimal workflows were used for training the ML models.

As described in Section III, we first estimated the input profiles for all the tasks in the DAG and then predicted each task execution time using ML models that we previously built. We compared the workflow execution time prediction error (MAPE) for BB-FULL, i.e. the state of the art Ernest approach, and GB-FULL feature sets. The error distribution is depicted in Fig. 2. While using BB-FULL we observed an average MAPE of 28%, using GB-FULL features the average MAPE dropped to 17%. Moreover, while with BB-FULL (Ernest) the error variance is higher, reaching more than 120% prediction error in one case, the highest error measured for GB-FULL is slightly higher than 50%.

#### F. Extrapolation analysis

A desirable way to build a prediction model for performance estimation would consist in learning with small input data and few computational resources and expect a low prediction error even when the input data is much bigger and the execution environment is more powerful. In this way, the training dataset could be built in a short-time and renting expensive resources would not be necessary. If a ML model is able to guarantee a stable prediction error for unseen value ranges of a given feature, we can say that the model is robust against the scaling of that feature. In this section, we compare the robustness of models, built with different ML techniques and trained on different feature sets, w.r.t. the scaling of the number of cores and of the input data size. Since we observed that DT models are as robust as the models produced by RFs, we only show the comparison between RF and LR.

In the first two rows of Fig. 3, we tested the robustness against the scaling of the input data size for ScQL-MAP (first row) and ScQL-Join (second row). The first plot in each row shows on the x-axis the input data size and on the y-axis the number of executions in our dataset performed with an input of that size. We trained the models on small executions (with input size lower than $x_0 = 20\%$ of the maximum size) and measured the validation MAPE on executions belonging to unseen input size ranges: 20-40%, 20-60%, 20-80% and 20-100% of the maximum input size. The scaling of the MAPE is depicted in the plots belonging to the second and third columns of Fig. 3. The first point in each plot (positioned at $x_0$) represents the validation MAPE computed on some executions randomly selected from the 0-20% range that were not used for training.

Results show that RF guarantees a good scaling, while LR fails (overfits in 0-20% range), unless gray-box full features are provided. Similarly, we tested the robustness against the scaling of the number of cores, which, for our experiments, corresponds to the scaling of the number of worker nodes in the EMR cluster. We trained our models on executions with 16 and 32 cores (i.e. 2 and 4 worker nodes), and measured the validation MAPE on executions using 48 cores, (48 & 64) cores and (48 & 64 & 96) cores. Again, results show that RF guarantees a good scaling, while LR fails (overfits), unless composite features are used. In this case, LR is robust even using black-box features, given that non-linear features involving the number of cores are defined (in our feature sets input-total-size/cores has been selected by SFS).

### V. RELATED WORK

To the best of our knowledge, this is the first comprehensive work on performance estimation for scientific data-driven workflows, implemented on Spark, based on machine learning and analytical models. Moreover, we did not find any modular workflow performance prediction solutions which explicitly address the problem of estimating

<table>
<thead>
<tr>
<th>Feature</th>
<th>Analytical Heuristics</th>
<th>ML</th>
</tr>
</thead>
<tbody>
<tr>
<td>out-num-files</td>
<td>0%</td>
<td>0.004%</td>
</tr>
<tr>
<td>out-total-size</td>
<td>0.2%</td>
<td>0.11%</td>
</tr>
<tr>
<td>out-interval-length</td>
<td>0.005%</td>
<td>0.03%</td>
</tr>
</tbody>
</table>
intermediate tasks input data profiles, which are not known offline.

Performance prediction for cloud-based applications has been tackled in several ways. Some studies apply traditional techniques, such as analytical models [6], [20]–[22] and simulation [23]. These techniques usually require detailed knowledge of the system, which is available only at runtime, and make simplifying assumptions that produce models unable to capture the complexity of cloud-based big data applications, losing in accuracy. Some studies, e.g. [24], propose a change in the Spark architecture that makes the job completion time estimation easier. More recently, there have been studies exploiting machine learning (ML) models for performance prediction of large systems [9]–[12], [25]. Most of these works use a black-box approach, in which historical data is used to predict future execution time, without knowledge of the system internals. Some works, instead, try to apply gray-box approaches, taking into account some system detail [12], [26]. Ernest [9], using simple features (functions of the input size and of the number of cores) and non-negative least squares (NNLS) regression, is considered the state-of-the-art in using ML. In [12], several ML models and several approaches (black-box vs. gray-box) are compared, showing better accuracy w.r.t. Ernest.

Fig. 3: Validation error (MAPE) variation w.r.t. the scaling of the input data size and the number of cores.
In the area concerning workflow performance prediction, most of the works focus on the individual task execution time prediction [27]–[29], highlighting the important role that this prediction plays in the context of optimal workflow scheduling [30]. In those papers, machine learning techniques are applied to predict the execution time of tasks contained in different real-world static workflows, i.e. workflows with a fixed structure, executed on the cloud. Feature sets mostly account for environment parameters and only the workflow input data size, not the individual input to each task, is used to describe the input data.

Compared to previous work, we extend performance prediction to complex Spark applications, showing the limitation of using black-box features, and we consider complex workflows with arbitrary combinations of tasks, whose performance can be accurately predicted thanks to intermediate output estimation.

VI. CONCLUSIONS

In this paper, we presented a hybrid three-phase modular solution for predicting the performance of complex data-driven workflows which can be mapped to Apache Spark applications. The workflow performance is predicted by combining individual task execution time predictions. Compared to previous works in this area, we targeted dynamic workflows, with arbitrary tasks composition, introduced intermediate-profile estimation, essential for the proposed approach, and considered a real-world complex system as a benchmark. In the experimental evaluation we compared the accuracy of different ML methods using different feature sets, which depend on how much knowledge on the application is available. Results show that, even for complex systems, performance can be predicted with good accuracy, which improves when the semantics of tasks is known, i.e. when using gray-box features. Moreover, the produced models keep a low prediction error even for unseen input data size, amount of cluster resources and workflow dimensions.

ACKNOWLEDGMENT

This work was supported by the AWS Machine Learning Research Award (MLRA) and by the Data-Driven Genomic Computing (GeCo) project, funded by the European Research Center (ERC) (Advanced ERC Grant 693174).

REFERENCES


Voilà: Tail-Latency-Aware Fog Application Replicas Autoscaler

Ali J. Fahs1, Guillaume Pierre1, and Erik Elmroth2,3
1Univ Rennes, Inria, CNRS, IRISA
2Umeå University
3Elastisys AB

Abstract—Latency-sensitive fog computing applications may use replication both to scale their capacity and to place application instances as close as possible to their end users. In such geo-distributed environments, a good replica placement should maintain the tail network latency between end-user devices and their closest replica within acceptable bounds while avoiding overloaded replicas. When facing non-stationary workloads it is essential to dynamically adjust the number and locations of a fog application’s replicas. We propose Voilà, a tail-latency-aware auto-scaler integrated in the Kubernetes orchestration system. Voilà maintains a fine-grained view of the volumes of traffic generated from different user locations, and uses simple yet highly-effective procedures to maintain suitable application resources in terms of size and location.

I. INTRODUCTION

Humans and IoT devices produce ever-increasing volumes of data. It is expected that, by 2025, 75% of all enterprise data will be generated out of the data centers [1]. Transmitting these data over long-distance networks to the cloud before processing them is becoming increasingly undesirable and sometimes even infeasible. Instead, “fog computing” aims at processing data using resources within very low latency to the end users [2]. In contrast to cloud computing where large numbers of resources are co-located in a handful of datacenters far away from the users, fog computing scatters nodes on the network edge, in the immediate vicinity of the end users.

Although some fog computing applications are designed to serve the needs of a single end user, many others aim to serve requests from a population of end users located within a broad geographical area such as a city or a region [3]. To deliver low-latency processing of their requests, fog applications may be designed as a set of functionally-equivalent service replicas which can be placed in strategic network locations such that every user request can be served by a nearby replica.

The number of service replicas an application should deploy is mainly determined by two factors. First, the geographical distribution of the end users requires one to create enough replicas such that a nearby replica exists for every source of traffic. Second, any replica necessarily has a limited processing capacity, which may require one to create multiple replicas to serve workloads originating from major sources of traffic.

Fog computing resources are precious in a multi-tenant environment, so fog applications must carefully adjust their deployments so that they satisfy their QoS objectives while reducing their resource usage as much as possible. On the other hand, any user-produced workload may largely vary over time [4], which motivates the need for using an auto-scaler to dynamically adjust the number and locations of a fog application’s replicas.

A fog application replica auto-scaler aims to reach three objectives: (1) network proximity such that every request may be routed to a nearby replica with a network round-trip latency lower than some threshold \( l_o \); (2) processing capacity management such that no replica receives more requests than its processing capacity \( c_r \); and (3) high resource utilization such that the majority of the provisioned resources are actually being utilized according to their capacity. Following best practice from commercial content delivery networks [5], we aim to optimize the tail network latency rather than its mean, which practically requires minimizing the number of user requests which incur a network round-trip latency \( l > l_o \).

We propose Voilà, a tail-latency-aware fog application replica auto-scaler. Voilà integrates seamlessly with Kubernetes, the de-facto standard container orchestration framework in clusters and data centers [6]. Kubernetes is also a promising basis for designing future-generation fog computing platforms [7]–[11]. Voilà continuously monitors the request workload produced by all potential traffic sources in the system, and uses efficient algorithms to determine the number and location of replicas that are necessary to maintain the application’s QoS within its expected bounds despite potentially large variations in the request workload characteristics.

Our evaluations based on a 22-nodes cluster and a real traffic trace shows that Voilà guarantees 98% of the requests are routed toward a nearby and non-overloaded replica. The system also scales well to much larger system sizes.

Section II surveys the technical background, and Section III discusses the related work. Then Section IV presents the system and Section V evaluates it. Finally, Section VI concludes.

II. BACKGROUND

A. Kubernetes

Kubernetes is an open-source orchestration engine which automates the deployment, scaling and management of containerized applications [12]. As shown in Figure 1, a Kubernetes cluster consists of a master node responsible for the monitoring and the management of the deployed applications,
and any number of worker nodes that constitute the system’s computing, network, and storage resources.

In Kubernetes, an application deployment is composed of a set of pods, defined as a set of logically-related containers and data volumes to be deployed on a single machine. Application replication is ensured by deploying multiple identical pods. These pods can be then exposed to external end users as a single entity by creating a service which exposes a single IP address to the end users and acts as a front end which routes requests to one of the corresponding pods.

The Kubernetes scheduler is in charge of determining which pod will be placed in which worker node. By default Kubernetes uses latency-unaware filtering/scoring algorithms. The objective of this work is to design a new scheduler which controls both the number of replicas and their placement within the fog computing infrastructure to maintain QoS guarantees across large variations in the end-user request workload.

B. Network Proximity

In a geo-distributed system like a fog computing platform, every user request should be routed to a nearby node to be processed as quickly as possible. In Kubernetes, request routing works in two phases. First, some external routing mechanism must ensure that end user’s requests are routed to any of the Kubernetes’ worker nodes. Fog computing platforms typically make use of SDN/NFV technologies to route end user requests to a nearby node which then acts as a gateway to enter the Kubernetes system. In a second phase, Kubernetes further routes the request toward one of the worker nodes which execute a pod of the concerned application, where the request gets processed.

a) Estimating network latencies: To determine which worker node should receive each incoming request, and to reason about the necessary placement of an application’s pods within the available worker nodes, it is important to know the network latency between every pair of worker nodes in the system. Rather than relying on simple metrics like geographical distance (which delivers poor accuracy [13]) or frequent pairwise measurements (which would require \(O(N^2)\) measurements), we rely on Serf [14], a mature implementation of Vivaldi coordinates [15]. Serf relies on a lightweight gossip-based algorithm to maintain reasonably accurate latency estimates between every pair of nodes in the system.

Since the first part of a request route (from the end-user device to the first Kubernetes node) is independent from the Kubernetes system itself, in this work we consider only the network latencies between this gateway node and the worker nodes holding application replicas.

b) Routing requests to a nearby pod: Kubernetes’ internal routing is based on simple location-unaware load balancing techniques such as round robin between all available pods of the concerned application. However, this approach may dispatch requests to far-away replicas even if there exists nearby ones. To ensure that requests are received by nearby pods, Voilà relies on Proxy-mity, which overrides the routing rules to favorize replicas reachable through low-latency routes [7].

c) Pod placement and auto-scaling: Routing requests to the closest available pod is not sufficient to guarantee low latency for every end-user requests. Depending on the end-users’ locations within the system, it is necessary to carefully choose the number of pods and their node placement such that Proxy-mity can find a nearby pod to route requests to.

Choosing the number and placement of an application’s pods necessarily results from a tradeoff between using the smallest possible number of pods (to reduce costs), while maintaining enough of them to cover the regions where users are located, and without overloading any of the pod replicas. The number and placement of pods is therefore a complex function of the Quality-of-Service requirements defined by the application, and the request workload imposed by its users.

d) Optimizing the tail vs. mean network latency: In latency-sensitive environments such as fog computing applications, it is important that each individual request gets processed with very low latency. For instance, virtual reality applications usually require consistently low response times. In such applications “lag spikes and dropouts need to be kept to a minimum, or users will feel detached” [16]. Aiming to minimize the mean latency between the user devices and their closest replica does not allow one to provide such extremely demanding type of guarantees. Instead, for each application we define a round-trip latency threshold \(l_o\) (e.g., \(l_o = 20\ ms\)), and aim to minimize the number of “slow” requests which incur a round-trip network latency \(l > l_o\).

Likewise, requests addressed to an overloaded replica may incur delays due to the saturation of server resources. We define a replica load threshold \(c_o\) (e.g., \(c_o = 50\ req/s\)) and also flag requests addressed to overloaded replicas as “slow.”

C. Non-stationary traffic properties

Any online application which processes incoming requests from an unbounded user population notoriously experiences significant workload variations across time [4]. This also applies to fog applications. However, geo-distributed systems such as fog computing platforms must not only handle variations of the aggregate workload produced by their entire user
population, but also variations in the location from which the users generate their traffic.

To highlight the non-stationarity in time and in space experienced in the fog, we analyze a geo-distributed request workload derived from telecommunication traffic traces from the Trentino region in Italy, and emulated proportionally to the number of Internet requests per user found in the trace [17].

Figure 2 shows the aggregated requests of users in 10 different areas of the city, and highlights the traffic produced by three of them. Overall we see a typical day/night pattern where most of the workload is produced between 9am and 11pm. However, different zones observe workload peaks at different times of the day. Commercial districts with shopping malls and offices peak at 12pm and 4pm, whereas residential areas peak at 7pm and nightlife neighborhoods peak at 11pm.

For a fog replica auto-scaler such as Voilà, this means that application replicas should not only be created in the morning and removed in the evening to follow the aggregated traffic intensity. Also during the day, to maintain proximity between the users and the application, replicas must be created/deleted/relocated from one neighborhood to another.

III. STATE OF THE ART

Fog computing servers are considered as precious resources. Numerous resource management mechanisms have therefore been proposed to address various aspects of fog computing resource management [18], [19].

A number of systems do not vary the number of replicas but rather propose to offload selected tasks to a backend cloud to reserve precious fog resources only for the most demanding tasks. Mukherjee et al. reduce the overall task completion latency by solving Quadratically Constraint Quadratic Programming optimization problem [20]. Vu et al. offload services in fog radio access networks to optimize the energy consumption and offload latency [21]. Yousefpour et al. propose a delay-minimizing offloading policy for fog nodes to reduce service delay for the IoT nodes [22]. In all these works, the fog layer network latencies are either not considered [20] or defined as a constant between all the nodes [21], [22]. In contrast, we consider the fog as a set of dispersed nodes where placing a pod on one node or another strongly influences the resulting quality of service. Also, choosing some tasks to be processed in the backend cloud does not reduce the tail execution latency. We thus aim at processing all the requests in the fog layer, without offloading to a backend cloud.

Several papers are based on Kubernetes. Zheng et al. propose to vary the number of pods according to the load, but does not address the question of pod placement nor efficient routing between the end users and their closest pod [11]. Several other works propose location-aware pod placement mechanisms for Kubernetes [8]–[10], [23]. However, they consider the number of replicas as a constant and do not aim to vary it to accommodate non-stationary workloads.

Finally, few systems propose auto-scaling for the fog. ENORM aims to reduce latency between users and computing devices, and the network traffic to the cloud [24]. However, it chooses the resources regardless of their location, and essentially considers every fog node as equivalent to one another. ORCH proposes to dynamically add compute nodes in the system to resolve workload surges [25]. In contrast, we consider the set of worker nodes as a constant and aim to place the right number of replicas in the right set of nodes.

To our best knowledge, Voilà is the only dynamic fog resource manager which considers at the same time auto-scaling to adjust the necessary number of application replicas across any significant variation of the request workload, placement/replacement to choose where these replicas should execute, and routing of end-user request to nearby replicas. It aims at optimizing the tail request latency rather than its mean while avoiding replica overloads. This paper presents the autoscaling and placement algorithms, and evaluates their implementation integrated in Kubernetes on a testbed of 22 fog nodes. Request routing is addressed in a separate paper [7].

IV. SYSTEM DESIGN

The objective of this work is to dynamically scale and place an application’s replicas in a cluster of geo-distributed fog nodes to predominantly minimize the number of slow requests while maintaining efficient resource utilization. A request is said to be slow in two cases: (1) it encounters a network round-trip time between the Kubernetes gateway and the serving pod greater than the threshold latency $l_0$ defined by the application provider; or (ii) it is addressed to a pod whose current workload is greater than the specified pod capacity $c_0$. System administrators are also requested to define $\mathcal{E}_0$, the maximum acceptable percentage of slow requests.

A. System model and monitoring

Table I summarizes the main variables used to describe our system model. A fog computing cluster is defined as a set of $n$ server nodes $\Delta = \{\delta_1, \delta_2, \ldots, \delta_n\}$, where every $\delta_i$ is an object of class Node which holds the status of the server node and the list of pods it currently hosts. An application is defined as a set of $r$ replicas $\Phi = \{\varphi_1, \varphi_2, \ldots, \varphi_r\}$, where every $\varphi_i$ is an object of class Pod which holds the status of the pod and the identifier of the server node where it is running. All these variables are maintained by Kubernetes as part of its...
normal operations. They can be obtained through simple call to Kubernetes’ etcd service.

The latency matrix $L$ contains all round-trip latencies between pairs of nodes, where every $l_{ij}$ is the RTT latency between nodes $\delta_i$ and $\delta_j$ as estimated by Serf. We can obtain an up-to-date estimate of any such RTT latency with a simple call to Serf’s rtt API at the master node.

Every worker node in a Kubernetes cluster has two different roles. First, it may host a pod of the application which processes user requests. Second, it may act as a gateway. End user requests may be sent to any gateway node, which is then in charge of routing the request to the one of the application’s pods. For clarity, we distinguish these two roles as gateway $g_i$ and server node $\delta_i$.

Kubernetes routes network requests from the gateway nodes to the server nodes using IP-level routing. This means that we have access to precise kernel-level counters measuring exactly how many network packets have been routed from which gateway to which server node, and back. To ensure that gateways route incoming requests to nearby nodes, Proxy-mity defines a matrix $P$ where every $p_{ij}$ represents the probability a request received by gateway $g_i$ should be routed to server node $\delta_j$, defined using a monotonically decreasing function of the estimated latency between $g_i$ and each server node where pods may be located (so that requests have high probability of being routed to nearby server nodes). Being filled with probabilities, the matrix $P$ maintains the following properties:

$$0 \leq p_{ij} \leq 1 \quad \forall (i, j) \in [1, n]^2$$

$$\sum_{j=1}^{n} p_{ij} = 1 \quad \forall i \in [1, n]$$

The specific values $p_{ij}$ are defined by Proxy-mity for all gateways and server nodes, as if every node actually hosted a pod of the application.

B. Replica placement quality evaluation

We evaluate the quality of any potential replica placement decision as the percentage of slow requests among all received requests ($\mathcal{E}\%$). Any placement decision consists of a set of host nodes $\omega_i$. Voilà calculates $\mathcal{E}(\omega_i)$ of any potential placement according to the current load and latency distribution.

To allow Voilà to evaluate a large number of potential placements in reasonable time, $\mathcal{E}$ should be evaluated as efficiently as possible. Voilà defines the probability matrix $P$ once per placement cycle, and then exploits it to evaluate the quality of any replica placement.

Procedure 1 illustrates the computation of $\mathcal{E}(\omega_i)$. First, it removes the matrix’s rows which correspond to idle gateways. It also sets to 0 the columns which correspond to server nodes which do not host a replica in the evaluated placement. After normalizing the matrix such that the sum of probabilities per row equals 1, the resulting matrix $\hat{P}$ contains only the routing probabilities from active gateways to potential replicas.

Using $\hat{P}$ and the set of active gateways $\hat{G}$, Voilà calculates the number of slow requests due to high network latency:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Delta$</td>
<td>${\delta_1, \delta_2, \ldots, \delta_n}$ set of all server nodes.</td>
</tr>
<tr>
<td>$\delta_i$</td>
<td>A server node of index $i$.</td>
</tr>
<tr>
<td>$n$</td>
<td>$</td>
</tr>
<tr>
<td>$\Phi$</td>
<td>${\varphi_1, \varphi_2, \ldots, \varphi_n}$ set of application’s replicas.</td>
</tr>
<tr>
<td>$\varphi_j$</td>
<td>An application replica of index $j$.</td>
</tr>
<tr>
<td>$r$</td>
<td>$[0, n]$, number of application replicas.</td>
</tr>
<tr>
<td>$L$</td>
<td>$[l_{ij}]$, symmetric $n \times n$ matrix of inter-node RTT latencies.</td>
</tr>
<tr>
<td>$l_{ij}$</td>
<td>$l_{ij}$ RTT latency between nodes $\delta_i$ and $\delta_j$.</td>
</tr>
<tr>
<td>$G$</td>
<td>${g_1, g_2, \ldots, g_n}$, set of all end user’s gateways.</td>
</tr>
<tr>
<td>$g_i$</td>
<td>$g_i$, a gateway located at node $\delta_i$.</td>
</tr>
<tr>
<td>$g_i.load$</td>
<td>The number of requests redirected by gateway $g_i$.</td>
</tr>
<tr>
<td>$\varphi_j.load$</td>
<td>The number of requests received by server node $\varphi_j$.</td>
</tr>
<tr>
<td>$P$</td>
<td>$[p_{ij}]$, $n \times n$ matrix of the request route probabilities.</td>
</tr>
<tr>
<td>$\mathcal{E}$</td>
<td>${\omega_1, \omega_2, \ldots, \omega_r}$, set of all possible placements.</td>
</tr>
<tr>
<td>$\mathcal{E}_\tau$</td>
<td>$\mathcal{E}$ per cycle threshold in %.</td>
</tr>
<tr>
<td>$\tau$</td>
<td>Cycle duration in s.</td>
</tr>
<tr>
<td>$l_o$</td>
<td>RTT latency threshold in ms.</td>
</tr>
<tr>
<td>$c_o$</td>
<td>Pod capacity threshold in req/pod/s.</td>
</tr>
</tbody>
</table>

$$V_{\tau_o}(\hat{G}, \hat{P}, L) = \sum_{i=1}^{n} \sum_{j=1}^{n} \hat{p}_{ij} \times \hat{g}_{i}.load \times f_1(l_{ij})$$

where

$$f_1(l_{ij}) = \begin{cases} 1 & \text{if } l_{ij} > l_o \\ 0 & \text{else} \end{cases}$$

Similarly, function $V_{c_o}$ counts the requests which would be routed to an overloaded replica $\varphi_j$ hosted at $\delta_k$:

$$V_{c_o}(\Phi, \hat{P}) = \sum_{j=1}^{r} f_2(\varphi_j) \varphi_j.load = \sum_{i=1}^{\frac{|\hat{G}|}{r}} \hat{g}_{i}.load \times \hat{p}_{ik}$$

where

$$f_2(\varphi_j) = \begin{cases} \varphi_j.load - c_o \times \tau & \text{if } \varphi_j.load > c_o \times \tau \\ 0 & \text{else} \end{cases}$$

The variable $\mathcal{E}$ is then computed as the sum of $V_{l_o}$ and $V_{c_o}$ divided by the total load:

$$\mathcal{E}^\% = 100\% \times \frac{\text{Slow Total}}{\text{Total}} = 100\% \times \frac{(V_{l_o} + V_{c_o})}{\sum_{i=1}^{n} g_{i}.load}$$

Selecting a suitable replica placement consists in finding $\omega_i$ such that $\mathcal{E}(\omega_i) \leq \mathcal{E}_\tau$.

C. Initial replica placement

When a new application is deployed in the fog computing platform, no information is available yet about its traffic characteristics. Instead, Voilà uses the set of active gateways from other deployed applications (regardless of their actual workload) to define an initial set of replica locations.

Finding the optimal placement of $r$ replicas among $n$ worker nodes requires in principle one to fully explore the set of all
possible solutions $\Omega$. However, this set is extremely large even for a modest value of $n$ and $r$:

$$|\Omega| = \binom{r}{n} = \frac{n!}{r!(n-r)!}$$

For example, placing 10 replicas out of 50 server nodes yields a set of $10,272,278,170$ possible placements. Exploring them all is obviously infeasible. Instead, we explore only a small subset of promising placements, and choose the first one which satisfies that all active gateways have a nearby replica to which they may route incoming requests.

Procedure 2 takes the latency matrix $L$ as an input, and labels all the possible routes as suitable (with value 1 if $l_{ij} \leq l_0$) or unsuitable (with value 0 otherwise). The objective of the resulting Test matrix $T$ is to check whether every active gateway is covered by at least one nearby replica. This condition is determined by the fact that each row corresponding to an active gateway $\hat{g}_i$ has at least one $t_{ij} = 1$.

The algorithm to identify a suitable replica placement is illustrated in Procedure 3. Starting from an empty placement $\omega$, the set of active gateways $G$, and the Test Matrix $T$, the algorithm starts by removing the idle gateways from $T$, then iteratively adds new replicas until all active gateways are covered by at least one suitable nearby replica. Every new host node $\delta_x$ is chosen with a greedy heuristic as the one which covers the greatest number of gateways. The loop continues until $G$ becomes empty, which indicates that all active gateways are covered. The procedure finally returns $\omega$.

Note that this initial placement is only a starting point when a new application is deployed in the platform. It is determined based on latency requirements only. Depending on the request workload, it may or may not satisfy the processing capacity requirement as well. Also, user-generated traffic is expected to vary over time, which mandates the usage of re-placement and autoscaling techniques, as we discuss next.

### D. Replacement and autoscaling

After an application has started, Voilà periodically checks whether the latency and the processing capacity requirements are still met. In case of violation, it implements corrective actions to bring the QoS within its desired bounds. Voilà also periodically checks whether fewer replicas may be sufficient.

1) **Checking for potential violations**: Voilà periodically checks whether the QoS constraints are still respected. As shown in Procedure 4, the QoS check algorithm starts by calculating the percentage $E$ of slow requests in the last cycle for $\omega_0$. If $E(\omega_0) > \varepsilon_0$, a violation is declared and the violation type determines which corrective function must be called.

2) **Replica replacement**: When a QoS violation is detected, Voilà first tries to fix it by moving a replica from one server node to another. Procedure 5 starts by selecting a number of replicas To-Be-Replaced (TBR) and a number of server nodes To-Be-Tested (TBT). The TBR and TBT are chosen according to the nature of the QoS violation:

- In the case of a Proximity violation, TBR is the set of current replicas which participate the least to the gateways-to- replica
types of QoS violations. It iteratively chooses a pair of nodes

- In the case of a capacity saturation violation, TBR contains the list of nodes located in close proximity from the currently overloaded replicas.

Once the sets TBR and TBT have been defined, Procedure 5 takes replica re-placement decisions in the same way for both types of QoS violations. It iteratively chooses a pair of nodes \( \delta_1 \in TBR, \delta_2 \in TBT \), and evaluates \( E \) in case node \( \delta_1 \) was replaced with \( \delta_2 \) in the current replica placement \( \omega \). If at least one replacement decision delivers an acceptable QoS with \( E(\omega) < E_0 \) within some pre-defined computation time, then the procedure returns the best replacement decision it has found. Otherwise, it considers that replacing replicas is unlikely to address the QoS violation, so it calls the Scale Up procedure to create an additional replica.

3) Scaling up: To choose the node where an additional replica should be created, Procedure 6 first defines a set TBT in the same way as previously. It then iteratively considers every node from this set and tests whether adding it to \( \omega \) (without replacing the existing replicas) would solve the QoS violations. If no single new replica is found to be able to solve the violation, it tries to add two new replicas, and so on until the violation is solved or all nodes from TBT have been added.

**V. EVALUATION**

A. Experimental setup

We evaluate Voilà with both experimental measurements and simulations. The experimental setup consists of 22 Raspberry Pi (RPI) model 3B+ single-board computers acting as fog computing servers. Such machines are frequently used to prototype fog computing infrastructures [8], [26], [27]. They run HypriotOS Linux v1.9.0 with kernel 4.14.34, Docker v18.04.0 and Kubernetes v1.9.3. We implemented Voilà on top of Serf v0.8.2.dev and the development version of Proxy-mity.

The RPIs are organized with one master node and 21 worker nodes capable of hosting replicas. Every worker node also acts as a WiFi hotspot and a Kubernetes gateway so end users can connect to the WiFi network and send requests to the service.

We emulate a realistic workload based on a trace of geodistributed Internet requests in the province of Trentino in Italy [17]. Every request is tagged with a location at 1 km granularity of the base station it was addressed to. We randomly select 22 1 km² cells and inject the load of each cell in a different testbed gateway. The application is a simple web server which returns the IP address of the serving pod, such that the request processing time is almost zero.

We emulate realistic inter-node latencies using the Linux tc command. Latency values are defined as a linear function of the geographical distance between the cells. They range from 4 ms to 80 ms with a median of 26 ms, which arguably represents a typical situation for a fog computing infrastructure.

We also perform scalability analysis using a simulator which simulates up to 500 virtual nodes using the same latencies and workload distributions, as well as the same algorithm implementations as in Voilà to select replica placements.

B. Autoscaling behavior

We first evaluate Voilà on the testbed based on parameters shown in Table II. Figure 3 shows 28 hours of workload from the Trentino trace, and Voilà’s autoscaling behavior when confronted to this workload. We speed up the trace so every hour in the trace is replayed over 2 min in the experiment.
When the application is deployed at 12am on the first day, the initial replica placement algorithm creates two replicas. We however notice that, although the workload intensity is fairly low, about 5% of requests are being slow, mainly because of network latency between the gateways and the replicas. At the next cycle Voilà creates a third replica, which fixes this QoS violation. At 2am another violation occurs, but a replica replacement is sufficient to solve this issue. Between 7am and 9am we observe a strong workload increase. Voilà detects a capacity saturation violation and reacts by bringing the number of replicas to 5 such that the violation is resolved and the number of slow requests gets back to almost zero.

From 10am until 8pm the global load stabilizes close to its daily peak. However, as discussed in Section II-C this “stable” workload still observes many changes in the users’ locations. We observe that Voilà adjusts to these changes by issuing a number of replica relocation operations. Finally, when traffic decreases in the evening, Voilà scales the system down to three replicas after observing three cycles with no QoS violations.

We conclude that Voilà effectively controls the number and location of replicas. Only $E_T = 2.6\%$ of all requests were categorized as slow: 0.59% because of proximity violations, and 2.01% because of capacity saturation violations.

C. Scaling up before saturation violations take place

The main reason for saturation violations is that any replica creation takes a few dozen seconds before the new replica becomes operational. If Voilà triggers a scale-up only after observing a saturation violation, then many requests may get penalized in the mean time. A practical solution to mitigate this effect consists of defining a safety margin and triggering scale-up operations to handle potential capacity saturation issues before saturation actually takes place.

Figure 4 shows the resource utilization of the busiest pod and the cumulative fraction of slow requests among the trace with safety margins 0%, 10% and 20% of actual pod capacity, which respectively trigger adaptation when any pod’s workload reaches 100%, 90% and 80% capacity. Larger safety margins reduce the number of capacity saturation violations from $E_T = 2.6\%$ to $E_T = 1.7\%$ with safety = 20%. However, because replicas are created sooner, the resource utilization through the day reduces slightly, from 48% to 39%. Better QoS comes at a greater cost in terms of resource usage.

Further reducing the number of violations would require predictive traffic models capable of anticipating the 9am traffic surge sufficiently early. We leave this topic for future work.

D. Sensitivity analysis

We now explore Voilà’s performance by means of simulations in a 200-node system with 100 active gateways. We set $E_o = 1\%$, and define default parameter values $c_o = 100\text{ req/pod/s}$, $l_o = 20\text{ ms}$ and safety = 20%.

Figure 5 shows the system behavior when varying $c_o$, $l_o$ and safety. Each test was repeated 50 times using different load distributions of 28 cycles from the Trentino grid. Every plot displays the average and the 95%-confidence interval, except the deployment size plot where the error bars depict the minimum and maximum sizes reached during the tests.

1) Deployment size: When $c_o$ and $l_o$ have low values, Voilà compensates by adding pods. Similarly, larger safety margins imply that nodes are less utilized, which requires more pods.

2) Slow Requests: The number of slow requests decreases when we increase the value of $c_o$: a smaller number of high-capacity pods can better absorb traffic intensity variations. Similarly, increasing the safety margin reduces the saturation violations. Varying $l_o$ shows two effects: first, strict latency requirements with a low value of $l_o$ makes latency-aware placement more difficult, which results in greater numbers of proximity violations. More surprisingly, larger values of $l_o$ result in an increase in saturation violations. The reason is that when a gateway becomes active, its produced traffic increases over a short time period. When $l_o$ is high, Voilà creates less replicas, so the violation occurs at a higher load rate which leads to more slow requests.

<table>
<thead>
<tr>
<th>Table II TESTBED EVALUATION PARAMETERS.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable</td>
</tr>
<tr>
<td>$l_o$</td>
</tr>
<tr>
<td>$c_o$</td>
</tr>
<tr>
<td>$E_o$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>22 nodes</td>
</tr>
<tr>
<td>$</td>
<td>C</td>
</tr>
<tr>
<td>Cycle duration $\tau$</td>
<td>120 s</td>
</tr>
</tbody>
</table>
3) **Resource utilization**: When the capacity \( c_o \) of every pod increases, their placement becomes increasingly dictated by latency considerations. Their average utilization therefore decreases. We observe a similar effect with low latency thresholds \( l_o \) where the many replicas that are created to cover the relevant areas of the network actually receive a modest workload each. Finally, as previously observed, increasing the safety margin decreases resource utilization.

4) **Voilà under extreme values of \( l_o \)**: Voilà performs well even with very strict values of \( l_o \). For example a requirement of \( l_o = 10 \text{ ms} \) is very challenging because in our experiments only 7.5% of the inter-node latencies are below 10 ms. In this case Voilà still maintains \( E_T < 1.7\% \), yet at the expense of a large number of replicas with low resource utilization.

### E. Scalability

We finally evaluate the execution time for various system sizes. All measurements are done on a quad-core Intel Core i7-7600U @2.80GHz laptop, \( c_o = 100 \text{ req/pod/s} \), \( l_o = 20 \text{ ms} \), \( E_o = 1\% \), half of the gateways transmitting load and over 10 runs with 28 cycles for each test.

Figure 6 compares the average number of placement that can be studied per second for various system sizes with the average number of placements that must be evaluated to repair a latency or capacity violation. When the cluster size increases, the time needed to study any single placement also increases. However, even for a large system with 500 nodes, Voilà evaluates \( \approx 100 \) placements per second. Voilà’s algorithms typically need about 100-150 placement evaluations to repair a violation, regardless of system size.

VI. CONCLUSION

Fog computing platforms must carefully control the number and placement of application replicas to ensure guaranteed proximity between the users and the replicas serving their requests, while avoiding replica overload and reducing resource consumption as much as possible. To our best knowledge, Voilà is the only proposed system which satisfies these three objectives even in challenging situations, and has been integrated in a popular container orchestration platform.

### REFERENCES

Baloo: Measuring and Modeling the Performance Configurations of Distributed DBMS

Johannes Grohmann  
*University of Würzburg*  
Würzburg, Germany  
johannes.grohmann@uni-wuerzburg.de

Daniel Seybold  
*Ulm University*  
Ulm, Germany  
daniel.seybold@uni-ulm.de

Simon Eismann  
*University of Würzburg*  
Würzburg, Germany  
simon.eismann@uni-wuerzburg.de

Mark Leznik  
*Ulm University*  
Ulm, Germany  
mark.leznik@uni-ulm.de

Samuel Kounev  
*University of Würzburg*  
Würzburg, Germany  
samuel.kounev@uni-wuerzburg.de

Jörg Domaschka  
*Ulm University*  
Ulm, Germany  
joerg.domaschka@uni-ulm.de

Abstract—Correctly configuring a distributed database management system (DBMS) deployed in a cloud environment for maximizing performance poses many challenges to operators. Even if the entire configuration spectrum could be measured directly, which is often infeasible due to the multitude of parameters, single measurements are subject to random variations and need to be repeated multiple times.

In this work, we propose Baloo, a framework for systematically measuring and modeling different performance-relevant configurations of distributed DBMS in cloud environments. Baloo dynamically estimates the required number of measurement configurations, as well as the number of required measurement repetitions per configuration based on a desired target accuracy. We evaluate Baloo based on a data set consisting of 900 DBMS configuration measurements conducted in our private cloud setup. Our evaluation shows that the highly configurable framework is able to achieve a prediction error of up to 12%, while saving over 80% of the measurement effort. We also publish all code and the acquired data set to foster future research.

Index Terms—Performance Modeling, Distributed Database Management Systems, Parameter Optimization, Machine Learning, Cloud Computing

I. INTRODUCTION

Many software systems offer parameters to configure internal workings. This allows performance engineers to influence non-functional properties and with them application performance. Yet, these configurations are not always easy to interpret and interdependencies between different configuration options are hard to oversee. In consequence, engineers traditionally rely on rules of thumb and testing. Performance prediction of configurable systems [1]–[4] aims at automating their modeling and optimization and providing engineers with a scientifically approved toolbox for taking configuration decisions.

Database Management Systems (DBMS) are particularly challenging configurable software systems: Relational DBMS offer a wide range of configuration options that expose a multitude of interdependent knobs to tweak the DBMS performance for a specific workload. To address the complexity of such DBMS optimization problems, many different approaches for finding the best possible DBMS configuration exist [5]–[7].

Beyond relational DBMS, NoSQL and NewSQL DBMS exploit distributed architectures that provide more non-functional properties including horizontal scalability, elasticity, and availability [8], [9], which are determined by additional configuration options such as cluster size and replication factor.

For operating distributed DBMS cloud resources have become the preferred infrastructure, as they provide scalability and elasticity on resource level [10]. Alas, this operational model further increases the configuration space adding cloud-related dimensions such as resource type, storage backend, and others.

In addition to understanding the performance impact of each individual configuration, it is necessary to also understand the interdependencies between parameters in the overall configuration space. This is extremely challenging even for experts in the three domains and therefore demands for supportive methods covering the entire configuration space. Due to its size one cannot simply evaluate each and every configuration option, but instead need to improve decision-making by predicting the performance of all configurations using a subset of measurements and hence, decreasing overhead.

Related work on performance prediction of cloud-hosted DBMS focuses on single-node DBMS ignoring distribution [11], [12]; targets only specific DBMS technologies ignoring cloud resource characteristics [13]; or considers only cloud resources without DBMS characteristics [14].

A core challenge when dealing with performance models of distributed DBMS is the time-intensive and expensive generation of the underlying data set: (1) Measurements of single configuration points are costly, as it requires a cluster of cloud resources that need to be reserved during the entire measurement period. (2) Performance measurements of distributed DBMS have a high variability [15] and therefore need to be repeated multiple times to achieve statistical significance. (3) The thorough evaluation of a performance prediction approach for any configurable system requires measurements of every possible configuration; an exponentially growing space.

Our contributions in this paper are as follows:

1) We present a novel framework for the measurement and
modeling of arbitrarily complex configuration spaces of configurable software systems. The design of this approach is specifically targeted at distributed DBMS: Our approach (1) selects a suitable robust statistical measure for the given scenario, (2) determines the minimal required number of measurement repetitions for a given measurement point, (3) chooses the next required measurement point, and (4) constructs a model of the configured parameter space using Machine Learning.

2) We evaluate our approach using a data set with measurements of a distributed DBMS with various configuration options. We publish this reference data set¹ comprising a total of around 450 measurement hours and roughly 9,450 computing hours in a private cloud environment.

By modeling the whole configuration space, our approach can quickly extrapolate expected performance results for a given configuration without actually measuring it. This is a strong benefit over a naive black-box optimization search. Hence, the resulting DBMS performance configuration model provides the foundation for selecting a performance-optimal operations and deployment configuration of a configurable system. Besides finding the most performant configuration, it gives a better understanding of the entire configuration space, providing valuable insights for operators and architects when trading performance against other non-functional aspects such as security, reliability, and costs.

The generic nature of the proposed framework enables researchers and practitioners to configure, adapt, and modify our approach as well as to transfer it to other domains. Our publicly available data set supports other researchers analyzing the performance behavior of the investigated DBMS in detail and evaluate further approaches for performance prediction.

The rest of this paper is structured as follows. First, Section II clarifies requirements before we introduce the Baloo approach in Section III. Section IV presents an evaluation of Baloo including a description of our data set. Finally, we list the related work in Section V. Section VI concludes the paper.

II. PERFORMANCE ENGINEERING REQUIREMENTS

For operating a distributed DBMS in volatile, cloud-like environments, operators can choose between multiple DBMS, cloud providers, cloud resources, and several configuration options of the DBMS including cluster size, replication factor, and client consistency. A performance model takes as input a possible configuration from that configuration space and outputs the expected performance of this configuration. As such, a performance model helps operators to answer various questions on performance impact of changes to the configurations (adding/removing resources, changing the consistency model, changing the size of the resources), but also changes in use as caused by, e.g., a changed read/write ratio. For DBMS, performance is either measured in latency or throughput. In an ideal world, the operator knew the performance of all the different possible configurations, but in practice, this is not the case. In consequence, an operator can estimate the performance of a single configuration by benchmarking that configuration with a workload as close as possible to the expected workload. In the following, we refer to a single configuration as a configuration point and to an execution of a benchmark as a measurement run.

Requirement 1 (Comparability): In order to judge which configuration points are better, it is necessary to compare the outcomes of the measurement runs for different configuration points. Yet, this comparison is difficult, as the raw results of a measurement run is a time series of throughput and latency values. Instead, a higher-level metric is required that captures the quality of a measurement run for a configuration point.

Requirement 2 (Repeatability): Cloud-like environments are characterized by volatility in performance so that multiple independent measurement runs are required to conclude on the actual performance of a configuration point.

Requirement 3 (Prediction): Being able to predict the performance of unmeasured configuration points helps reducing required time and costs for creating the performance model.

Requirement 4 (Trade-off between quality and cost): Obviously, the more measurement runs per configuration point and the more configuration points are measured, the more precise the performance model will be. Yet, the more measurements, the higher the time and financial impact. Hence, an approach like Baloo needs to offer the capability to flexibly decide on the required quality and hence, costs.

Requirement 5 (Validation): In order to validate Baloo an extensive data set with DBMS performance measurements is required that needs to contain multiple measurements for different configuration points. Due to the fact that such data sets are hard to obtain from real-world applications such a data set needs to be based on synthetic data, but still make use of realistic configurations.

III. APPROACH

Figure 1 shows a graphical overview of our approach, Baloo. The overall goal of Baloo is to run as few measurements as possible for each configuration point and to use as few configuration points as possible to train an accurate performance model. This performance model then enables the performance prediction of arbitrary configuration points without explicitly measuring them.

The framework is initialized with the Robust Metric Selection (offline phase) that provides the means to summarize the results of individual measurement runs. The goal of this phase is to determine the robust metric that—based on a large data set of existing performance measurements—best reduces noise in the measurement data. Once the robust metric has been found, performance engineers can request the creation of a performance model from Baloo. For creating the performance model, Baloo generates a training data set in a workflow with three steps: (i) Baloo selects crucial configuration points from the configuration space; (ii) Baloo decides how often each of the configuration points needs to be measured; (iii) Baloo triggers the measurements using the Mowgli measurement
framework [16]. For all of these steps, Baloo applies statistical heuristics to decide how many measurements to conduct per configuration point and further to decide which and how many configuration points to chose.

In the following, we describe Robust Metric Selection, Distributed DBMS Performance Measurement, Measurement Repetition Determination, Measurement Point Selection, and Performance Model Construction in more detail.

A. Robust Metric Selection

This step addresses the Comparability Requirement (see Requirement 1 in Section II) by selecting a suitable metric for summarizing the performance time series of one performance measurement. This is required, since distributed DBMS operated in volatile environments are subject to performance variations and fluctuations in latency and throughput measurements during one experiment run. We therefore need metrics to summarize each run. Furthermore, those metrics need to enable a robust comparability between different configuration points. The most important characteristic of a metric in our scenario is its robustness to the given fluctuations. Hence, the name robust metric. The "ideal" robust metric reports the same values for two repetitions of the same configuration point.

For finding the best robust metric, we compare different robust metrics candidates by analyzing their coefficient of variation (CV), a measure of the standard deviation in relation to the sample mean, over all measurement runs available. Based on the resulting list of CVs for each individual measurement, we rate each metric using the mean and the standard deviation of their respective CV-scores. The metric observing the lowest mean CV is then defined to be the most suitable robust metric for the given scenario.

For potential robust metrics, we investigate common robust measures of central tendency from literature [17]: mean, median, different percentiles, trimmed mean, winsorized mean, trimmean, and the hodges-lehman estimator. A full analysis of all robust metrics is presented in Section IV-B.

B. Distributed DBMS Performance Measurement

This step lays the basis for the following steps as it generates the data for individual measurement runs (step 5 in Figure 1). For doing so, Baloo builds on the open source and extensible DBMS evaluation framework Mowgli2 [16] that supports the design and execution of DBMS evaluations. Mowgli allows to define relevant domain-specific properties and allow their specification based on the supported technologies such as the DBMS itself, cluster size, replication factor, cloud provider, resource capacity, and workload type; Furthermore, it fully automates allocation of cloud resources, deployment and configuration, workload generation, calculation of performance data, and processing of results. The resulting evaluation data (step 6) sets contain the benchmark metrics, monitoring data, resource metadata, and execution logs of the respective evaluation tasks.

C. Measurement Repetition Determination

This step addresses Requirement 2 and Requirement 4 of Section II as it decides on the number of required measurements for a specific configuration point \( P \). Here, the overall number of measurements is dependent on the desired confidence \( t_c \) and stability of the measurements, which is dependent on the volatility of the environment.

Baloo requires at least two measurements in order to judge on the stability of the results, and uses a configurable upper limit \( n_{\text{max}} \) in order to control execution time and costs. Details are presented in Algorithm 1.

First, the set \( M \) of performance measurements is filled by two consecutive calls of \( \text{triggerMeasurement}(P) \) (step 4)
Algorithm 1: Measurement Repetition Determination.

```
Input: Desired configuration point \( P \),
       target confidence threshold \( t_c \),
       maximum number of measurements \( n_{\text{max}} \).
Output: Obtained measurement value \( m \).
1 \( M = \text{triggerMeasurement}(P) \)
2 \( \text{while } c > t_c \text{ and } |M| < n_{\text{max}} \)
3 \( M = M \cup \text{triggerMeasurement}(P) \)
4 \( M' = \text{removeOutliers}(M) \)
5 \( m = \text{aggregate}(M') \)
6 \( c = \text{confidence}(M') \)
7 \( \text{return } m \)
```

Algorithm 2: Performance model construction.

```
Input: Configuration space definition \( S \),
       target score threshold \( t_s \),
       maximum configurations ratio \( r_{\text{max}} \).
Output: Performance model \( p \).
1 \( C = \text{getInitMeasurements}(S, t) \)
2 \( p = \text{constructPerformanceModel}(C) \)
3 \( s = \text{scorePerformanceModel}(C, p) \)
4 \( \text{while } s < t_s \text{ and } |C| \leq r_{\text{max}} \cdot |S| \)
5 \( C = C \cup \text{addMeasurements}(S, C, r) \)
6 \( p = \text{constructPerformanceModel}(C) \)
7 \( s = \text{scorePerformanceModel}(C, p) \)
8 \( \text{return } p \)
```

that wraps invocations to Mowgli as described in Section III-B. In lines 4–7, the obtained measurement values are analyzed and aggregated. If the calculated confidence \( c \) deceeds \( t_c \), or if \( n_{\text{max}} \) has been reached, the calculated aggregation \( m \) is returned (step 7).

Baloo allows different implementations of outlier detection, aggregation, and confidence estimations (cf. lines 4–7). Our implementation used in Section IV applies outlier detection based on isolation forests [18] through the Python version of scikit-learn [19] with two isolation trees. We use the mean as aggregation function and quantify confidence through the coefficient of variation (CV). The confidence threshold \( t_c \) is set to 0.02. Hence, the loop stops if \( \text{CV}(M') < 0.02 \). Note that both median and CV are not calculated on \( M \), but rather on \( M' \) that does not contain outliers.

D. Configuration Point Selection

This step selects the next configuration point (step 3) that needs to be added to the training set (cf. Requirement 4). This process is commonly referred to as sampling [20], while the underlying selection strategy is called sampling method.

While our framework supports any strategy for the selection of the next configuration point, the implementation used in Section IV is based on uniformly distributed random sampling that was found leading to the most accurate performance models in general [21], but not always [22]. Our random implementation relies on enumerating the entire configuration space, which does not pose a problem in our scenario. Otherwise, more sophisticated solutions are required, e.g., based on binary decision diagrams [23] or satisifiability solvers [24].

E. Performance Model Construction

This step is concerned with the construction of the actual performance model for a configuration space \( S \) and, therefore, addresses Requirements 3 and 4 of Section II. Its iterative approach determines the minimal required number of measurements for achieving a configurable target accuracy \( t_s \). A configurable maximum ratio \( r_{\text{max}} \) of \( S \) is used as configurable upper limit in order to control execution time and costs.

Here, \( S \) is the explorative part of the overall configuration space, i.e. the Cartesian product of all available feature values, while \( t_s \) is the targeted score.

Algorithm 2 shows that first a set of initial measurements is conducted (line 1) based on the given configuration space (step 2). All measurements (step 8) constitute the set of available measurements \( C \), from which a performance model \( p \) is built and scored using an internal scoring function. The \( i \) parameter to \( \text{getInitMeasurements} \) determines ratio of \( S \) used.

The algorithm keeps adding measurement points in line 5 until \( s \geq t_s \) or \( |C| \geq r_{\text{max}} \cdot |S| \). In each iteration, it recomputes \( p \) (line 6) and \( s \) (line 7). The \( r \) parameter to \( \text{addMeasurements} \) determines the ratio of \( S \) added in each iteration. After termination it returns the last trained performance model (step 9).

The algorithm is highly parameterizable by adapting \( t_s \) and the scoring function. For model construction, any regression or other performance modeling technique can be applied. The implementation used in Section IV applies a threefold cross-validation score on \( C \) to determine model accuracy and uses variance as score function. Furthermore, we compare and evaluate different machine learning algorithms for the model construction in Section IV-D.

IV. Evaluation

In this section, we evaluate our approach following a four-step method: in the first step, we use an external, third-party data set to determine the robust metric (cf. Section IV-B); in the second step, we use a newly generated data set to validate the chosen metric (cf. Section IV-B). The very same data set is then used to evaluate the quality of the measurement repetition determination in Section IV-C and the performance model construction in Section IV-D. The data sets are described in Section IV-A. The Baloo framework as well as all evaluation scripts are available for repetition as a CodeOcean capsule.

A. Validation Data Sets

For validating Baloo we make use of two different data sets. The configuration space they cover is depicted in Table I and comprises eleven dimensions. The first data set is an existing openly available data set [25], comprising 102 configuration points for Apache Cassandra and Couchbase. The Apache
Cassandra configuration points serve as a reference data set for determining the robust metric. The second data set has been created for this work and is published as OpenData⁴. For this data set, seven configuration dimensions are static for the sake of this evaluation: We select a private OpenStack-based cloud infrastructure as this gives us control over OpenStack-specific configurations such as the overcommitting factor and Virtual Machine placement. We select Apache Cassandra as representative cloud-hosted DBMS due to its widespread adoption⁴. Finally, we use write-heavy workload issued from the Yahoo Cloud Serving Benchmark (YCSB) [26] and four million records of a record size of 5 KB for comparability to other work [16]. We use SSD as storage as it is recommended for most DBMS. For the remaining four dimensions we use three (VM type, client consistency, replication factor) and five (cluster size) different configuration options. Due to the fact that client consistency and replication factor cannot be chosen fully independently, this yields a total of 90 different configuration points. The data set contains 10 measurements for each of those and for each measurement a time series of performance metrics, system metrics, and additional meta data.

### B. Robust Metric Selection

Here, we evaluate the different statistical measures that are robust metric candidates. We start with a publicly available data set [25] and then evaluate the transferability by comparing the results with our data set. Table II compares the average CV score of the different robust metrics. The lower the score, the less variation and hence the better the metric fits our needs. Although the two data sets are different in regard to the used cloud infrastructure, the VM-sizes, and the number of measurement repetitions, we observe that the performance of the different metrics is very comparable for both throughput and latency. For throughput, the 95th percentile achieves the best score across both data sets. When analyzing latency, we observe that both data sets have one minimum at for the trimmed mean of 30%. The first data set has an additional minimum using the 30%-winsorized metric, while the second data set performs slightly better using the median. However, both the 30%-winsorized and the median are on the third place for the respective other data set.

Summarizing, we can say that the results from the publicly available data set transfer very well to our own data set. Hence, we conclude that the 95%-percentile for throughput, as well as the trimmed mean or the winsorized mean for latency, are viable robust metrics that can be applied for comparing DBMS cloud performance. As DBMS are usually optimized for throughput, we concentrate on throughput for the remainder of this work. Based on our insights, we use the 95th percentile as robust metric.

### C. Measurement Repetition Determination

In this section, we evaluate the measurement repetition determination discussed in Section III-C. We do so by comparing estimations obtained by Baloo with the median overall ten measurement repetitions from the evaluation data set. As the true mean of the underlying distribution is unknown, we make the assumption that the median of all 10 measurements approximates the true mean of the underlying distribution and serves as a gold standard in this evaluation. We compare this gold standard against randomly selected 1, 2, 3, and 5 points as well as against the result obtained from Baloo.

As Baloo uses probabilistic elements and the selection of the next measurement point is non-deterministic as well and strongly influences the obtained results, we repeat the evaluation 100 times. Table III reports the average results.

Table III shows the mean absolute percentage error (MAPE) and the root mean squared error (RMSE) of Baloo and the five baselines together with the average measurement repetitions required. We observe that the error generally decreases as we increase the number of measurement points. This is expected as more measurement points reduce the impact of random measurement noise. Baloo outperforms 1-point, 2-point, and even 3-point in average MAPE and RMSE, while requiring only 2.59 measurement points on average. This shows that

---

⁴https://db-engines.com/en/ranking

---

### TABLE I

**EVALUATION CONFIGURATION SPACE**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Seybold et al. [25]</th>
<th>Baloo data set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Infrastructure</td>
<td>public Amazon EC2,</td>
<td>private OpenStack</td>
</tr>
<tr>
<td></td>
<td>private OpenStack</td>
<td></td>
</tr>
<tr>
<td>VM Type</td>
<td>small - t2.medium</td>
<td>tiny - small - large</td>
</tr>
<tr>
<td>DBMS</td>
<td>Apache Cassandra</td>
<td>Apache Cassandra</td>
</tr>
<tr>
<td>Cluster size</td>
<td>3 - 5 - 7 - 9</td>
<td>5 - 7 - 9 - 11</td>
</tr>
<tr>
<td>Client consistency</td>
<td>any - one - two</td>
<td>one - two - three</td>
</tr>
<tr>
<td>Replication factor</td>
<td>3</td>
<td>1 - 2 - 3</td>
</tr>
<tr>
<td>Benchmark</td>
<td>YCSB</td>
<td>YCSB</td>
</tr>
<tr>
<td>Workload</td>
<td>write-heavy</td>
<td>write-heavy</td>
</tr>
<tr>
<td>Records</td>
<td>4,000,000</td>
<td>4,000,000</td>
</tr>
<tr>
<td>Record size</td>
<td>5 KB</td>
<td>5 KB</td>
</tr>
<tr>
<td>Storage backend</td>
<td>SSD, HDD, remote</td>
<td>SSD</td>
</tr>
</tbody>
</table>

---

### TABLE II

**COMPARING THE AVERAGE CV FOR EACH ROBUSTNESS METRICS FOR THE EXTERNAL DATA SET (EXT) WITH OUR OWN DATA SET (OWN).**

<table>
<thead>
<tr>
<th>Metric</th>
<th>Throughput CV</th>
<th>Latency CV</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ext</td>
<td>Own</td>
</tr>
<tr>
<td>Mean</td>
<td>0.039</td>
<td>0.062</td>
</tr>
<tr>
<td>Median</td>
<td>0.044</td>
<td>0.063</td>
</tr>
<tr>
<td>95th percentile</td>
<td>0.028</td>
<td>0.039</td>
</tr>
<tr>
<td>90th percentile</td>
<td>0.029</td>
<td>0.042</td>
</tr>
<tr>
<td>80th percentile</td>
<td>0.032</td>
<td>0.047</td>
</tr>
<tr>
<td>Trimmed(5%) mean</td>
<td>0.039</td>
<td>0.062</td>
</tr>
<tr>
<td>Trimmed(10%) mean</td>
<td>0.039</td>
<td>0.062</td>
</tr>
<tr>
<td>Trimmed(30%) mean</td>
<td>0.042</td>
<td>0.082</td>
</tr>
<tr>
<td>Winsorized(5%) mean</td>
<td>0.039</td>
<td>0.062</td>
</tr>
<tr>
<td>Winsorized(10%) mean</td>
<td>0.039</td>
<td>0.062</td>
</tr>
<tr>
<td>Winsorized(30%) mean</td>
<td>0.041</td>
<td>0.063</td>
</tr>
<tr>
<td>Trimean</td>
<td>0.042</td>
<td>0.062</td>
</tr>
<tr>
<td>Hodges-Lehmann</td>
<td>0.039</td>
<td>0.063</td>
</tr>
</tbody>
</table>
TABLE III
ACCURACY AND MEASUREMENT REPETITIONS FOR DIFFERENT
APPROACHES.

<table>
<thead>
<tr>
<th>Approach</th>
<th>MAPE (%)</th>
<th>RMSE</th>
<th>Average # points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baloo</td>
<td>1.42</td>
<td>251.8</td>
<td>2.59</td>
</tr>
<tr>
<td>1-point</td>
<td>2.75</td>
<td>538.4</td>
<td>1.00</td>
</tr>
<tr>
<td>2-point</td>
<td>2.19</td>
<td>403.3</td>
<td>2.00</td>
</tr>
<tr>
<td>3-point</td>
<td>1.54</td>
<td>315.4</td>
<td>3.00</td>
</tr>
<tr>
<td>5-point</td>
<td>0.76</td>
<td>148.1</td>
<td>5.00</td>
</tr>
<tr>
<td>10-point</td>
<td>0.00</td>
<td>0.0</td>
<td>10.00</td>
</tr>
</tbody>
</table>

in most of the cases, two measurements are sufficient to accurately describe a performance measurement.

In some cases, more measurements are required, though: This insight can be supported by comparing the reported MAPEs with the RMSEs. For MAPE our approach is only slightly better than 3-point (≈ 8% decrease); the difference is larger when analyzing RMSE (≈ 20% decrease). Since the RMSE puts a stronger focus on outliers, we can conclude that our approach is able to correctly determine "critical" measurement points, while keeping the requested measurement to the minimum amount of two measurements, when not required. The two baselines 5-point and 10-point consistently achieve lower errors than our proposed approach; this is expected, as they also conduct significantly more measurements. If a higher accuracy is required, our approach could be tuned accordingly. Summarizing, our approach successfully handles the trade-off between required measurement repetitions and target accuracy.

It is worth noting that in this experiment, the average number of required measurement points is comparatively low, as all experiments were executed on a private cloud with relatively low load (see Section IV-A). Therefore, the number of required measurement repetitions as well as the gain achieved by our approach might be even higher for other environments [27].

D. Performance Model Construction

In this section, we evaluate the performance model construction techniques and the corresponding performance models. As our approach works with basically any regression technique, our analysis compares the performance of different machine learning algorithms. For this, we analyze the required amount of measurement configuration points together with the achieved target accuracy on the remaining validation set for the following algorithms: linear regression (LinReg), ridge regression (Ridge), elastic net regression (ElasticNet), bayesian ridge regression (BayesianRidge), and huber regression (HuberRegressor) as linear models, a gradient tree boosting regressor (GBDT), a random forest regressor (RandomForest), and a support vector machine (SVR). Additionally, to get a better comparison, we add a baseline regressor (ZeroR) which always predicts the mean of all seen samples.

In this experiment we vary the target accuracy threshold $t_s$ of the internal MAPE score $s$ based on a threefold cross-validation between 0.1, 0.15, 0.2, 0.25, and 0.3. The maximum ratio $r_{max}$ was set to 0.9, resulting in a cutoff at 81 (0.9-90 total) configuration points. As initial configuration measurement set, we set a ratio of 0.05, resulting in at least 5 ([0.05 · 90]) configurations for each approach. We only add one configuration per increment (i.e., $r = 0.01$), as for our comparatively small example set, the training time was significantly lower than the measurement time and could therefore be neglected. Furthermore, this enables a better analysis of the required configuration points. All algorithms implementations are based on scikit-learn [19] and use the defined default parameterization.

The comparison is shown in Figure 2. The x-axis depicts the number of configuration points that were measured, before the specific workflow terminated, i.e., $t_s$ was achieved. The y-axis shows the MAPE on those configurations that were not added to the training set. We repeat all experiments 25 times and report the average value as these processes are highly influenced by the random seed.

From Figure 2, we observe that multiple approaches perform comparably in terms of required configuration points and the achieved prediction error. However, SVR and ZeroR (baseline) are not able to capture the performance structure very well and, therefore, perform poorly in terms of prediction error and required configuration points. The baseline even runs out of measurement options and therefore retrieves the maximum number of measurements for $t_s = 0.1$.

All other approaches perform similarly well, but we can still identify small differences between them. Increasing the target accuracy has the expected effect of generally reducing the prediction error while increasing the measured configuration points. While GBDT achieves the overall lowest prediction error (11.63%), random forest is able to achieve slightly worse results using considerably fewer configuration points for $t_s = 0.15$ and $t_s = 0.2$. For bigger values of $t_s$, GBDT performs slightly better again.

Table IV shows more details on the performance of the individual algorithms for $t_s = 0.15$. In addition to the average achieved error (MAPE) and the required configurations (＃
Conf.), we see the average number of total measurement runs conducted (# Meas.) and the average time of the workflow execution excluding measurements (Time).

We conclude that even un-optimized machine learning approaches are able to achieve prediction errors of around 20% on a dataset consisting of 90 configuration points, while measuring just one fifth (<20) configurations and conducting even less than 5% (<45) of individual measurement runs.

Adding more measurement points increases the accuracy and reduces the error to up to 12%. However, the accuracy gain per added configuration decreases over time, which is consistent with our expectations. The additional computation effort introduced by our framework is negligible (run-times of less than 1s per training process), if we consider that one measurement run takes minutes or even hours to complete. Therefore, the achieved time savings are more than 95%, for an accuracy cost of just 20%. Even by reducing the accuracy cost to 12%, we can still achieve measurement time reductions of over 80%. We believe this result to be very impressive, considering that average performance in public clouds fluctuates by 5% [28].

V. RELATED WORK

Related approaches can be divided into the following three research areas.

A. Performance measurement of distributed DBMS

DBMS benchmarking is a common process to determine the optimal operational model for DBMS in general and as well for cloud-hosted DBMS. This process is supported by a multitude of benchmarks that support diverse workload models and evaluation objectives [29], [30]. In consequence, there are numerous supportive performance studies of cloud-hosted DBMS available [26], [27], [31]. Yet, each of these studies covers only a small part of the entire cloud resource and distributed DBMS scope.

B. Performance optimization of DBMS

DBMS performance optimization approaches such as iTuned [5], DBSherlock [6] and OtterTune [7] target single instance relational DBMS that are operated on dedicated resources. These approaches have a special focus on the workload by considering trace-based workloads [5], [6] or unknown workload types [7]. Rafiki [11] targets the performance optimization of single instance NoSQL DBMS for different workload types by automatically determining DBMS runtime parameters and deriving their optimal configuration. Performance models for distributed DBMS are presented by Farias et al. [32] and Dipietro et al. [33], considering the performance prediction impact of different cluster sizes [32] and DBMS-specific runtime parameters [33].

The URSA framework [12] targets the automated capacity planning of a single node DBMS operated on cloud resources. Thus, the focus of URSA lies on the resources, while the aspects of distributed DBMS are not considered. In summary, existing approaches provide comprehensive performance prediction mechanisms for single node DBMS on dedicated resources [5]–[7], [11], focus on distribution aspects [32], [33] without considering cloud resources or consider only cloud resources without considering DBMS distribution aspects [12].

C. Performance prediction of configurable systems

Zhang et al. propose the application of Fourier learning to predict the performance of configurable systems with theoretical accuracy guarantees. [1]. Siegmund et al. combine machine learning and sampling heuristics to build performance-influence models for highly configurable systems. [2]. Sarkar et al. compare different sampling techniques for CART-based performance models and introduce a novel heuristic for the selection of the initial samples [34]. Guo et al. improve CART-based performance model by resampling the training data to determine the accuracy of the resulting model and automated hyper-parameter tuning [35]. Ha et al. propose a deep sparse neural network architecture and hyper-parameter optimization approach for the performance prediction of configurable systems [3]. Westermann et al. [36] compare the accuracy of MARS, CART, Genetic programming, and Kriging for the construction of software performance models. Similarly, Noorshams et al. [37] evaluate the accuracy of linear regression, MARS, CART, MS Trees and Cubist Forests for the performance modeling of storage systems. To the best of our knowledge, no existing performance prediction approach targets distributed DBMS or addresses measurement variability in the sample generation.

VI. CONCLUSION

In this work, we presented Baloo, a framework for measuring, modeling, and predicting the performance of distributed database management systems (DBMS) in cloud environments for different configurations. Our approach builds upon the Mowgli framework and works by (1) measuring a performance configuration, (2) determining the number of measurement repetitions, (3) determining the next configuration point to be measured, and (4) building a performance model using all available measurement points to predict the remaining unavailable measurement points of the configuration space.

To evaluate our framework, we measured the distributed DBMS Apache Cassandra in our private cloud using 90 dif-

---

TABLE IV
DETAILED PERFORMANCE OF ALL ALGORITHMS FOR FIXED TARGET ACCURACY $\tau_r$ OF 0.15.

<table>
<thead>
<tr>
<th>Approach</th>
<th>MAPE</th>
<th># Meas.</th>
<th># Conf.</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LinReg</td>
<td>21.87</td>
<td>44.72</td>
<td>17.08</td>
<td>0.44</td>
</tr>
<tr>
<td>Ridge</td>
<td>23.02</td>
<td>38.52</td>
<td>14.80</td>
<td>0.38</td>
</tr>
<tr>
<td>ElasticNet</td>
<td>23.28</td>
<td>88.48</td>
<td>34.84</td>
<td>0.93</td>
</tr>
<tr>
<td>BayesianRidge</td>
<td>20.08</td>
<td>65.12</td>
<td>25.84</td>
<td>0.77</td>
</tr>
<tr>
<td>HuberRegressor</td>
<td>19.50</td>
<td>51.16</td>
<td>20.56</td>
<td>0.96</td>
</tr>
<tr>
<td>GBDT</td>
<td>19.93</td>
<td>47.28</td>
<td>17.44</td>
<td>1.02</td>
</tr>
<tr>
<td>RandomForest</td>
<td>17.13</td>
<td>65.04</td>
<td>25.60</td>
<td>4.95</td>
</tr>
<tr>
<td>SVR</td>
<td>27.03</td>
<td>175.52</td>
<td>68.84</td>
<td>1.90</td>
</tr>
<tr>
<td>ZeroR</td>
<td>29.24</td>
<td>200.36</td>
<td>77.92</td>
<td>2.03</td>
</tr>
</tbody>
</table>
fferent configurations and ten repetitions each, resulting in 900 measurement runs comprising of roughly 450 measurement hours and 9,450 compute hours. We also made this data set publicly available to foster future research towards this area. The evaluation shows that our highly configurable approach is able to save between 80% and 95% of measurement time for a respective accuracy cost of 12% to 20%.

In future work, we plan to include other DBMS and cloud providers (including measurements from public clouds) in our configuration model itself. This would enable to not only to providers (including measurements from public clouds) in our configuration model itself. This would enable to not only to

REFERENCES


Instability in Geo-Distributed Kubernetes Federation: Causes and Mitigation

Mulgeta Ayalew Tamiru
Elastisys AB
Univ Rennes, Inria, CNRS, IRISA
mulgeta.tamiru@elastisys.com

Guillaume Pierre
Univ Rennes, Inria, CNRS, IRISA
guillaume.pierre@irisa.fr

Johan Tordsson, Erik Elmroth
Elastisys AB
tordsson.elmroth@elastisys.com

Abstract—As resources in geo-distributed environments are typically located in remote sites characterized by high latency and intermittent network connectivity, delays and transient network failures are common between the management layer and the remote resources. In this paper, we show that delays and transient network failures coupled with static configuration, including the default configuration parameter values, can lead to instability of application deployments in Kubernetes Federation, making applications unavailable for long periods of time. Leveraging on the benefits of configuration tuning, we propose a feedback controller to dynamically adjust the concerned configuration parameter to improve the stability of application deployments without slowing down the detection of hard failures. We show the effectiveness of our approach in a geo-distributed setup across five sites of Grid’5000, bringing system stability from 83–92% with no controller to 99.5–100% using the controller.

Index Terms—Self-configuration, self-adaptation, Kubernetes Federation, Fog Computing, automatic configuration tuning.

I. INTRODUCTION

Fog computing extends cloud computing by harnessing geographically-distributed computing resources for moving computation closer to where data are generated (e.g., IoT devices). One of the main challenges in fog computing is the autonomous management of tens of thousands of remote nodes and clusters found in diverse locations. Several approaches based on modified container orchestration frameworks such as Kubernetes have been proposed [1]–[3]. More recently, Kubernetes introduced the notion of Federation (KubeFed) which provides abstractions to manage multiple geo-distributed Kubernetes clusters from a single control plane.

Since Kubernetes was designed for managing local clusters in public and private cloud settings, it assumes reliable network connectivity between the nodes with low latency, high bandwidth, and low packet loss. KubeFed makes a similar assumption: while it is designed to manage Kubernetes clusters located in different regions of the same cloud provider or multiple cloud providers, KubeFed assumes high reliability of the network connectivity between the control plane and the managed clusters. However, such assumptions are not met in many fog computing environments [4]. As a result, static configurations, including the default values of the configuration parameters for both Kubernetes and KubeFed are not necessarily well-suited to the case of geo-distributed fog computing infrastructures.

It is well known that configuration parameters may have a strong influence on the performance and availability of systems [5]. However, finding the optimal configuration settings that result in the best performance of the system is not easy because of the large parameter space and the complex interaction of multiple parameters. This is the case of Kubernetes and KubeFed which are composed of several embedded control loops [6] with numerous configuration parameters.

In this paper, we demonstrate that federated applications deployed on a geo-distributed KubeFed infrastructure with static configuration, including the default settings, may suffer from important instability where containers get repeatedly created and deleted before being able to provide a useful service. To our best knowledge, we are the first to report this undesirable behavior of KubeFed.

Our contribution is two-fold. First, we demonstrate the existence of the instability problem in a realistic geo-distributed fog computing infrastructure based on KubeFed and identify one configuration parameter (Cluster Health Check Timeout), whose value influences stability the most. We show that, for obtaining the best system behavior, the value of this parameter should be adjusted according to the characteristics of the execution environment. Second, we propose, implement, and experimentally evaluate a feedback controller that improves the stability of the system by dynamically adjusting this configuration parameter at runtime. We show that this controller is very effective for improving the system’s stability across a wide range of inter-cluster network latencies and packet loss rates, without losing the ability to detect actual cluster failures. In our evaluations, the system stability improves from 83–92% with no controller to 99.5–100% using the controller. By enabling self-configuration and self-adaptation, our solution helps make KubeFed more autonomous.
II. BACKGROUND

Kubernetes is a container orchestration platform which automates the deployment, scaling, and management of containerized applications in large-scale computing infrastructures such as a cluster and a datacenter [6]. To extend it to multisite deployments, Kubernetes Federation (KubeFed) supports resource management and application deployment on multiple Kubernetes clusters from a single control plane, thus making it suitable for managing geo-distributed resources [7].

KubeFed’s implementation builds upon the concept of custom resource definitions (CRDs) from Kubernetes. In KubeFed terminology, a single host cluster runs the federation control plane which controls any number of member clusters where applications may be deployed. The host cluster is also the central point where the federation’s configuration parameters are defined. The KubeFed controller manager is itself a Kubernetes deployment resource deployed on the host cluster. It runs several controllers to manage the member clusters, scheduling, deployments, services, and other resources.

KubeFed introduces three concepts for each resource:

- **Template** defines the specification of a resource common across all member clusters;
- **Placement** specifies which member cluster(s) will get the resource;
- **Override** defines per-cluster variations of the template.

Using these concepts, users can define their deployments and services and decide how many application containers of a deployment should appear in which cluster. Figure 1 shows the KubeFed architecture with a host cluster and three member clusters where an “nginx” federated service is configured to be deployed on only two of the three member clusters.

KubeFed also offers ReplicaSchedulingPreference (RSP) which is an automated mechanism to distribute and rebalance federated deployments across the member clusters. This is useful when scaling an application across several clusters. As shown in Figure 2, users only need to specify the target resource to be controlled by RSP and the total number of replicas to be distributed in the federation. By default, RSP distributes the replicas evenly across all member clusters if they have sufficient resources. RSP does this in multiple iterations proportional to the number of total replicas. After calculating how many replicas should go to each member cluster, RSP modifies the Federated Deployment object to update the number of replicas on each cluster, which in turn pushes or reconciles the changes to the member clusters via the sync controller. The sync controller is responsible for propagating changes from the Federation Control Plane on the host cluster to the member clusters and maintaining the desired state of resources across member clusters.

There are several configuration parameters that control the behavior of the Sync Controller. The most important for our work is the Cluster Health Check Timeout (CHCT), which has a default value of 3 seconds. This parameter determines the duration after which sync requests and cluster health check time out. In a geo-distributed deployment where large network latencies and packet losses are commonplace, it is important to choose the right value for this parameter to ensure the correct functioning of the Sync Controller and RSP. A low CHCT value ensures fast detection of cluster failures, allowing RSP to rebalance the deployment away from failed clusters onto healthy clusters. However, fast detection increases the probability of false positives due to delays and transient network failures. Therefore, it may be necessary to increase the value of CHCT to minimize false positives. However, increasing the CHCT value may lead to delayed cluster failure detection. It is, therefore, important to consider the trade-off between fast and slow detection of cluster failures when choosing the CHCT parameter value. Manually selecting an optimal value for CHCT is challenging as the choice may depend on the computing and networking environment as well as application workload dynamics.

III. RELATED WORK

Modern distributed systems are increasingly complex and difficult to manage manually. As a result, autonomic computing approaches have been proposed to simplify the difficult task of managing these systems by way of self-configuration, self-adaptation, self-tuning, and self-healing [8], [9].

One aspect of the complexity of modern distributed systems is the large number of configuration parameters with complex interactions that affect the performance and availability of systems. Even though good configuration settings can improve the performance of systems, finding these settings among hundreds of parameters is often far from being trivial [5].
In the past decades, several works have proposed to leverage the concepts from autonomic computing. Many of the works focus on optimizing the performance of systems by finding the best combination of configuration settings from all the possible combinations [5], [10], [11]. Some other works focus on specific type of distributed systems or frameworks such as Enterprise Java (J2EE) [12], big data management systems such as Apache Hadoop [13] and Apache Spark [14], database management systems [15], distributed message systems such as Apache Kafka [16], web servers such as Apache web server [17]–[19], Docker [20], or Kubernetes [21].

Selecting appropriate values for timeout parameters, which is the specific topic of this paper, has proven particularly challenging in a number of related works. Typically, large values result in slow failure detection whereas small values reduce the reliability of the failure detector. To address this challenge, some works have proposed delay predictors which determine at runtime values for detection timeout for fast detection while not reducing the reliability of detection [22]–[25]. In [26], the authors propose an autonomic failure detector based on feedback control theory that re-configures its timeout and monitoring period parameters at runtime in response to changes in the computing environment or application according to user-defined QoS requirements. Similarly, we show that small values for the CHCT timeout parameter of KubeFed lead to instability whereas large values lead to slow failure detection.

The focus of most of these works is to find the right trade-off between accuracy and responsiveness of failure detectors, whereas our main focus is to improve the stability of application deployments without impairing the responsiveness of failure detection. Moreover, unlike these works, our work addresses a problem in a geo-distributed fog computing environment. To the best of our knowledge, the problem of failure detection and dynamic adjustment of configuration parameters have not been widely studied in the context of geo-distributed computing environments such as fog computing.

### IV. PROBLEM ANALYSIS

When a replicated application is deployed in a Kubernetes federation, in certain settings, the application incurs significant instability where containers are repeatedly created and deleted, which in turn causes application unavailability and unacceptably long application response times. In this section we first experimentally demonstrate the existence of this undesirable behavior, and then analyze its causes.

#### A. Experimental setup

To highlight the unstable deployment problem, we set up an experimental testbed as close as possible to a realistic fog computing environment, depicted in Figure 3. We deploy six Kubernetes 1.14 clusters in five sites of the Grid’5000 experimental testbed [27]: two in Rennes, and the other four clusters in Nantes, Lille, Grenoble, and Luxembourg. Every cluster has one master node and five worker nodes. KubeFed v0.1.0-rc6 is deployed on the first cluster as the host cluster, and the remaining five member clusters are then joined to the federation. Each node in the host cluster has 4 CPU cores and 16 GB of memory allocated to it, whereas each node in the member clusters has 4 CPUs and 4 GB of RAM. This configuration emulates typical fog computing resources which are often composed of single-board machines such as Raspberry Pis [28], [29]. We control the network performance inside each cluster and between the host and member clusters using the “traffic control” (t.c) tool available in Linux systems. The internal network of each cluster has 1 Gbps bandwidth, whereas the network characteristics between the host cluster and the member clusters are defined in Table I. These values are based on a recent study [4] which highlights the characteristics of today’s networking technologies used in edge computing settings.

The application used for our tests is a simple federated deployment of nginx web server that scales progressively. We scale the total number of replicas from 75, 100, 500, 1500, 2500, to 3500 to be distributed equally among the five member clusters of the federation. The task of automatically balancing the pod replicas across the member clusters is handled by KubeFed’s Replica Scheduling Preference (RSP) controller.

We define three scenarios for our experiments:

- **Stationary scenario**: a federation with Network Setting 1, with no variation in the networking environment and no cluster failure;
- **Network variability scenario**: a federation where the networking environment varies between Network Setting 1 and Network Setting 2, with no cluster failure;
- **Cluster failure scenario**: a federation with Network Setting 1, with no change in the networking environment but with a failure and a recovery of one member cluster.
B. The instability problem

As the total number of replicas of the pods of the federated deployment increase, RSP calculates the number of replicas to be distributed to each member cluster. Unless other requirements such as the minimum number of replicas or weights per cluster are specified by the user, RSP opts for an even distribution across all member clusters. As per the source code of KubeFed, \( \text{clusterCount} \times \log_{10}(\text{replicas}) \) iterations are required to distribute all replicas among the member clusters. The time complexity of this algorithm is \( O(\text{clusterCount}^2 \times \log_{10}(\text{replicas})) \), where \( \text{clusterCount} \) is the number of member clusters and \( \text{replicas} \) the requested number of containers. After determining the number of replicas per cluster, RSP updates the Federated Deployment Object’s override field. The changes are then automatically pushed to each cluster by the Sync Controller as depicted in Figure 2.

In a geo-distributed federation setup, there are two ways in which instability may arise due to network delays or transient network failures:

1) **Reconciliation failure**: Push reconciliation requests to one or more member clusters may time out prematurely at the time of scheduling by RSP, in which case the sync controller tries to re-sync the resources until the desired state is achieved. If transient network failures continue to happen, it may take a long time for the reconciliation to terminate.

2) **Health check failure**: One or more member clusters may be declared unhealthy by the kube-controller-manager if health check requests time out, in which case RSP recalculates the distribution of replicas and rebalances them by moving replicas away from the now-unhealthy clusters to healthy ones. RSP re-syncs the resources to the unhealthy clusters, if they become healthy again. These actions may repeat over and over in network environments with a large number of transient failures.

The unstable behavior is manifested by the number of replicas on the affected member clusters being significantly fewer than the desired numbers, sometimes even reaching zero. Figure 4 shows the number of deployment replicas, which is the number of replicas pushed by the kubefed-controller-manager, and the actual number of running pods in one of the member clusters in our setup during a period of instability. As shown in the figure, the number of replicas that the kubefed-controller-manager pushes to the member cluster fluctuates widely over time, in turn affecting the number of pods actually running on the cluster.

To quantify the unstable behavior we introduce the stability metric \( \nu \) as follows:

\[
\nu[\%] := \frac{1}{n} \sum_{i=1}^{n} \left( 100 - \frac{100}{T} \sum_{t=1}^{T} \frac{d_i - p_{i,t}}{d_i} \right) \tag{1}
\]

where \( n \) is the total number of member clusters and \( i \in [1, n] \); \( T \) is the full experiment duration and time \( t \in [0, T] \); \( d_i \) is the desired number of pods in cluster \( i \); \( p_{i,t} \) is the number of running pods in cluster \( i \) at time \( t \). Stability is a measure of how much and for how long the number of replicas in the member clusters is close to the desired number of replicas: a system which fails to deploy any replica during the entire experiment will have \( \nu = 0\% \) whereas a perfectly working and stable system will have \( \nu = 100\% \).

C. The influence of configuration parameters

To identify which of KubeFed’s configuration parameters has the greatest influence on the stability of the deployments in the member clusters, we conduct principal component analysis on the data obtained from the measurement of stability by varying the values of several parameters. We identified eight configuration parameters which might influence the behavior and stability of the system: timeout durations, health check periods, numbers of retries, etc. We then measured the stability derived from 705 randomly-chosen sets of parameter values.

Our results show that the first source of stability variations between different configurations can be attributed to a single parameter. Specifically, we observe instability mainly when the Cluster Health Check Timeout (CHCT) parameter has too low values. Even the default value of 3 s for this parameter leads to significant instability. We also notice that increasing the value of the CHCT parameter significantly improves the stability of the system.

D. Trade-off between instability and failure detection delay

Although the stability of the system improves when the value of the CHCT parameter is increased, setting very large values to the CHCT parameter leads to slower failure detection as the system needs to wait until the CHCT timeout expiration before it updates the status of the failed cluster as “Offline.” As shown in Figure 5, increasing CHCT leads to greater system stability; however, it also increases the failure detection delay. The goal of our controller is to identify a sweet spot which implements the necessary trade-off between these two effects.

E. The influence of the networking environment

The last important factor which influences the occurrence of instability is the network performance between the clusters. Figure 6 depicts the stability of the system for the three scenarios. We see that the system is very unstable for the Stationary scenario. Moreover, stability gets worse as the network latency and packet loss is increased or one of the clusters fails in the Network variability and Cluster failure.
scenarios, respectively. Table II shows the system stability measures of the uncontrolled system under the three evaluation scenarios.

V. A CONTROL-BASED APPROACH FOR TUNING CHCT

Since the CHCT parameter value is the cause of most instability problems, a natural solution would consist of finding a better value for this parameter. Generally speaking, the CHCT value should be set as low as possible (to reduce the delay in detecting actual cluster failures), but not too low either (because this would generate instability). However, no single “best” value can be found, as the choice of a good value largely depends on the operational conditions such as the inter-cluster network characteristics and the application workload.

Instead, we propose to dynamically adjust the CHCT value at runtime using a feedback controller which reacts to changes in operational conditions. Unlike other configuration tuning methods, this adaptive approach does not require prior knowledge of the infrastructure and it is simple to implement. In this section, we present the details of our solution including our design decisions, controller design, and tuning of the controller parameters.

A. Feedback controller design

Feedback controllers are widely used in mechanical and electrical systems, and they have also gained widespread use in computer systems [30]. A controller implements a feedback loop which monitors the system to be controlled, and implements automatic changes and then manipulates the input as needed to drive the system’s variable toward the desired setpoint.

Figure 7 shows the design of our proposed solution. The controller continuously monitors the measured output (called Process Variable, PV, in control theory terminology) of the kubefed-controller-manager to detect indications that the CHCT value is either too high or too low. It then produces a signal called control output (CO) that reduces the error (\(e\)) that indicates the deviation of the measured output from the reference value. Finally, the controller decides whether CHCT must be adjusted, and the actuator implements the change.

a) Choosing the Process Variable: A naive approach would consist of periodically evaluating the KubeFed stability metric, and of incrementing the CHCT value whenever the measured stability differs from the desired setpoint of 100% (\(\epsilon > 0\)). However, this would mean that the system must enter a period of instability before the CHCT value is updated. This reaction would be too late for our purpose which is precisely to prevent instabilities from occurring.

It is, therefore, necessary to base the controller reactions on other metrics which show early indications that instability is about to occur. For this we use the timeout errors written by the kubefed-controller-manager in its logfile whenever it fails to reach one of its member clusters. KubeFed starts deleting containers in the “failed” cluster and restarting them in other clusters when these timeout errors accumulate, thereby potentially triggering instability. We can thus use the occurrence of the first timeout errors in the log file as early indications that instability may soon take place.

Another motivation for selecting the number of timeout errors as the process variable PV is because this metric is readily available in the host cluster where we deploy our controller, unlike stability which needs to be computed after collecting metrics from each individual member cluster. Frequently collecting metrics from the member clusters may be very difficult, especially in periods of bad network performance when the CHCT value needs to be quickly adjusted.

b) Controlling the CHCT value: If no timeout errors are reported, then we know that the system should achieve 100% stability. We, therefore, define the setpoint SP to 0 timeout error. As a result, the controller increments the CHCT parameter value until the number of timeout errors found in the log file during the control interval reaches zero.

However, setting SP to zero creates a new problem. In the standard feedback control theory, one should allow both positive and negative errors so that the controller can automatically increase or decrease CO proportional to the error. In our case, since we define SP as zero, it is impossible to observe a number of timeout errors lower than the setpoint, and the controller cannot decrease the CHCT value as a result of such
Algorithm 1: Feedback controller algorithm.

Data: Positive Gain $K_p$, Negative Gain $K_n$.

Result: CO

initialization;

SP := 0;

decrement_period := 3;

t := decrement_period;

while true do

PV = number of timeout errors;
e = -(SP - PV);

CHCT = current value of the CHCT parameter;

if $e > 0$ then

CO = CHCT + $K_p$ * e;
t = decrement_period;

else

Do nothing;
t = t - 1;

if $t == 0$ then

CO = $(1 - K_n)$ * CHCT;
t = decrement_period;

end

end

end

negative errors. As a result, even though we can increase CO proportionally to the error, for the decreasing part we need to deviate from the standard approach of feedback control design and come up with a different approach. For simplicity, we decided to decrease CHCT periodically if no timeout error has been identified, independently from any indication that the CHCT value may be too high. The controller ensures the trade-off between improving stability and fast detection of failures by preventing CHCT from reaching very large values that could lead to increase in the failure detection delay.

We choose a sampling interval of 1 minute for practical reasons. To change the CHCT value of a running KubeFed, it is necessary to stop and restart the containers which execute the `kubefed-controller-manager`. This operation takes a few dozen seconds. A sampling interval of 1 minute, therefore, gives enough time for the system to change the CHCT value before starting the next iteration of the control algorithm.

c) Control algorithm: Our control algorithm is presented in Algorithm 1. The controller periodically measures the number of timeout errors which occurred in the previous period, and compares it to the setpoint $SP = 0$. If timeout errors have occurred, then the controller increments CHCT by a value proportional to the number of timeout errors and to the positive gain $K_p$, which is in line with the standard design of a proportional feedback controller. On the other hand, if no timeout errors have been found during three consecutive periods, the controller decreases CHCT proportionally to the negative gain $K_n$.

The two gain parameters $K_p$ and $K_n$ respectively define how aggressive the controller should be in increasing and in decreasing the CHCT value.

B. Tuning the controller parameters

Defining the controller parameters requires one to find a trade-off between a system which would react too slow to environment changes to provide an appropriate reaction and one which may potentially over-react to any such changes.

To get an initial estimate for $K_p$ we use the Ziegler-Nichols rules, which are a set of simple heuristics that perform well a wide variety of situations [31]. The Ziegler-Nichols tuning method does not require detailed knowledge of the controlled system, and the rules can be expressed entirely in terms of the system’s step-input response (i.e., the system’s reaction characteristics upon a change of its parameter value) [30].

Figure 8 shows the step response of the system as the CHCT parameter is suddenly increased from the default value of $CHCT = 3$ sec to $CHCT = 13$ sec at time $t = 10$ min. From the system’s step response, we estimate three parameters that are used in the Ziegler Nichols tuning rules:

- The process gain $K$ is the ratio of the change in process output $\Delta PV$ that results from a change of input $\Delta CO$:
  \[
  K = \frac{\Delta PV}{\Delta CO}
  \]

- The time constant $T$ is the time it takes for the process to settle to a new steady-state after experiencing a sudden change in input, i.e., the time it takes the process to reach about two-thirds of its final value.

- The dead time $\tau$ is the delay until an input change begins to affect the output.

From Figure 8, we find $K = 1.5$, $T = 60$ s, and $\tau = 60$ s. Based on these step-input response values we can define $K_p$:

\[
K_p = \alpha \times \frac{T}{K \times \tau}
\]

where $\alpha$ is a coefficient which typically falls in the range $[0.3, 1.2]$ [30]. We can therefore estimate that $K_p$ should fall in the range $[0.2, 0.8]$. Based on these estimations, in the next section we experiment the controller with $K_p$ values of 0.1, 0.5, and 1. Similarly, we use $K_n$ values of 0.1, 0.25, and 0.5.

VI. EVALUATION

A. Experimental Setup

We evaluate our controller using the same experimental setup as described in Section IV-A with the same application and workload for the three scenarios. However, now we also deploy our controller on the master node of the host cluster. We run a total of nine experiments per scenario, one for each combination of the $K_p$ and $K_n$ parameters of the controller.
Each experiment is run for two hours. We repeat each of the 27 experiments three times and report the mean value.

B. Experimental Results

We present the results of the stationary, network variability and cluster failure scenarios in Figures 9, 10 and 11 respectively. In all scenarios, the controller adjusts CHCT according to the conditions, and significantly improves the federation stability compared to the no-controller scenario from Figure 6.

In the Network variability scenario, CHCT increases from $t = 30\, \text{min}$ as a reaction to the degraded networking performance, and decreases back at $t = 90\, \text{min}$ when network performance returns to normal. Similarly, in the Cluster failure scenario, CHCT increases after $t = 30\, \text{min}$ as a reaction to the detected cluster failure, and decreases from $t = 90\, \text{min}$ after cluster recovery. The stability drop at $t = 90\, \text{min}$ is a direct consequence of cluster recovery, as several pods get stopped in other clusters and restarted in the recovered one.

In all scenarios, we see a faster increase of the CHCT parameter as $K_p$ increases, and faster decrease as $K_n$ increases. In some cases, the larger values of $K_n$ lead to a brief instability as the CHCT parameter is aggressively decreased to very low values, leading to timeouts.

To determine values of $K_p$ and $K_n$ which work in all three scenarios, we compare all 27 cases for accuracy. Table III shows the accuracy of the controller in decreasing the number of timeout errors $N$, and in improving the stability $v$. For each scenario, we show the best values for $N$ and $v$ in bold and the worst values in italic. We see that the controller with $K_n$ value of 1 and $K_n$ value of 0.1 has the best values for $N$ in all three scenarios, and the best value of $v$ in two out of three scenarios. Thus, we conclude that the controller works best in all three scenarios for this combination of values of the parameters $K_p$ and $K_n$. This configuration improves stability from 83–92% with no controller (see Table II) in stationary situations to 99.5–100% using the controller, even in challenging scenarios with network variability or cluster failures.
Fig. 11. Cluster failure scenario: one cluster fails at $t = 30$ min and recovers at $t = 90$ min.

VII. CONCLUSION

Geo-distributed systems such as fog computing platforms need to operate in difficult and uncertain networking conditions. In particular, it is notoriously difficult to distinguish actual node failures from delays caused by the networking or local node condition. We demonstrated that these effects can create significant instability in Kubernetes Federations. We identified the main configuration parameter which influences this behavior, and proposed a feedback controller which dynamically adapts its value to the operational conditions, and improves the system stability from 83–92% with no controller to 99.5–100% using the controller.

REFERENCES


Security-Performance Trade-offs of Kubernetes Container Runtimes

William Viktorsson+, Cristian Klein++, Johan Tordsson++
+Department of Computing Science, Umeå University, Umeå, Sweden; *Elastisys AB, Umeå, Sweden
{williamv,cklein,tordsson}@cs.umu.se

Abstract—
The extreme adoption rate of container technologies along with raised security concerns have resulted in the development of multiple alternative container runtimes targeting security through additional layers of indirection. In an apples-to-apples comparison, we deploy three runtimes in the same Kubernetes cluster, the security focused Kata and gVisor, as well as the default Kubernetes runtime runC. Our evaluation based on three real applications demonstrate that runC outperforms the more secure alternatives up to 5x, that gVisor deploys containers up to 2x faster than Kata, but that Kata executes container up to 1.6x faster than gVisor. Our work illustrates that alternative, more secure, runtimes can be used in a plug-and-play manner in Kubernetes, but at a significant performance penalty. Our study is useful both to practitioners – to understand the current state of the technology in order to make the right decision in the selection, operation and/or design of platforms – and to scholars to illustrate how these technologies evolved over time.

I. INTRODUCTION

Application containers are stand-alone software packages that consist of one or more applications and the libraries, programming runtimes, and other dependencies required to operate the applications. Containers provide application portability across platforms and are light-weight in both resource usage and required storage. These features simplify development and deployment of applications. The two key sandboxing mechanisms commonly used in containers are namespaces for isolation (of processes, networks, filesystems, etc.) and resource management (of CPU, memory, network, etc.), often implemented using cgroups. A potential shortcoming with container sandboxing is that the host operating system is shared among multiple containers on the same host. This contrasts the Virtual Machine (VM) approach where multiple applications share a host but each have their own host operating system.

Container have gained widespread adoption across a broad spectrum of industries, in particular following the releases of first Docker1 and later Kubernetes2. A wave of Linux kernel vulnerabilities, such as Dirty COW3, which can be used to escape containers, increased fears of containers providing insufficient isolation. This gave rise to multiple alternative runtimes that aim to increase security, e.g., by adding additional layer(s) of indirection and/or reducing the attack vector.

We analyze three container runtimes with respect to security and performance: Kata [1], gVisor [2], and runc (the default Kubernetes runtime). These were selected as they integrate seamlessly with Kubernetes, which is required for a runtime to gain broad adoption. For this reason, we excluded several interesting runtimes identified in the cloud native landscape4, including the Firecracker microVM5, and Nabla6 that reduce the attack surface by limiting the set of accessible system calls.

While performance evaluation of various containerization platforms is not new ([2]–[7]), our work distinguishes itself by evaluating container runtimes in a typical Kubernetes cluster using benchmark applications that are representative of the workloads that run in such clusters (Section II-B).

Our contributions include an analysis of the security mechanisms of the three studied runtimes (Section II), where we discuss to what extent they fulfill a set of requirements with respect to isolation and resource management. We also empirically evaluate the performance of the three runtimes (Section III) using a microservice benchmark (TeaStore), an in-memory data-store (Redis), and a big data processing framework (Spark). In our experiments, runc performs up to 5x better than Kata and gVisor in both container deployment time and application execution. Notably, gVisor deploys containers faster than Kata (up to 2x) but executes applications slower.

II. BACKGROUND

In this section, we present the relevant background of our work. We first summarize the relevant architectural choices of the selected container runtimes and quantify how well each container runtime fulfills the security requirements. Finally, we summarize earlier benchmarking efforts.

A. Security-Relevant Architecture

Container runtimes need to ensure process isolation, filesystem isolation, device isolation, Inter-Process Communication (IPC) isolation, network isolation and resource management. Fig. 1 compares the architecture of runC (the default container runtime), gVisor and Kata.

**gVisor** introduces a novel approach of provisioning a virtualized environment by moving system interfaces normally implemented by the host kernel into a user space kernel.

---

This work was partially supported by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

---

1https://www.docker.com
2https://www.kubernetes.io
3https://dirtycow.ninja/
4https://landscape.cncf.io/category=container-runtime
5https://github.com/firecracker-microvm
6https://nabla-containers.github.io/
System calls are intercepted and performed on behalf of the container, without the overhead imposed by a hypervisor [2].

gVisor consists two main components. Sentry is a user-space kernel that intercepts system calls made by the application. The platform responsible for intercepting syscalls is ptrace by default, but gVisor can be configured to use the Kernel-based Virtual Machine (KVM) [8] instead. The sentry provides a full network stack (called netstack) that isolates all networking from the host network stack. Gofer is a separate process running in each gVisor instance that provides file system access to the container using the 9P protocol [9].

Kata containers provide lightweight VMs with the goal of achieving performance equal to containers, but with a second layer of isolation using hardware isolation. It supports multiple hypervisors including QEMU, Firecracker and Cloud Hypervisor [1]. A VM is booted using a compact guest OS image, optimized for low kernel boot time and minimal memory footprint. Libcontainer is used to create the execution environment inside the VM, and run containers within that environment [10]. This implies that the containers run on the guest with cgroups and namespaces applied in the same manner a runC container uses these mechanisms on a host. To enable communication from the host into the guest, a Kata-agent is created, a process running inside the VM as a supervisor for containers running on the guest. Commands to the Kata runtime are forwarded to the Kata-agent over gRPC [11] that in turn makes use of Libcontainer to manage the lifecycle of containers. Network and storage are accessed via a set of paravirtualized devices through the virtio virtualization framework [12].

Table I presents the number of layers of indirection for each security aspect. More layers of indirection entail more security, as an attacker who gained control of the application would need to find more vulnerabilities to compromise security of the container runtime.

As expected, runC, being optimized for performance, only provides a single layer of indirection for each security requirement. Next comes gVisor with two layers of indirection for almost all requirements except resource management. In our security analysis, Kata comes out as the winner, having two layers of indirection for all security aspects.

### B. Earlier Benchmarking Results

Earlier work, including projects behind the secure runtimes, compare container runtimes using micro-benchmarks [13]. Various stressors are for CPU, memory or I/O. In some cases, syscalls are called repeatedly to highlight the careful design that went into optimizing a specific use-case or code path. Most of these micro-benchmarks are executed on bare-metal without needing to deal with nested virtualization [4]. The latter is more likely to be encountered typical Kubernetes clusters. Other run micro-benchmarks in nested virtualization environments [14], even on top of Kubernetes [15].

Our benchmarks complemented the above works as follows:

- We use full applications deployed on top of Kubernetes as benchmarks, as opposed to micro-benchmarks.
- We evaluate the container runtimes on Kubernetes clusters in a nested virtualization environments, as is common with public cloud providers.

### III. PERFORMANCE EVALUATION

We evaluate the performance of each container runtime, focusing on deployment and execution time for a set of typical microservice applications running in Kubernetes.

### A. Experiment Setup

Table II: Components used in experiments.

<table>
<thead>
<tr>
<th>Component</th>
<th>Version / Configuration</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kubernetes</td>
<td>1.17.0 (overlays)</td>
</tr>
<tr>
<td>Containerd</td>
<td>1.3.0</td>
</tr>
<tr>
<td>runC</td>
<td>1.0.1-dev</td>
</tr>
<tr>
<td>gVisor</td>
<td>2019-11-14 (ptrace)</td>
</tr>
<tr>
<td>Kata</td>
<td>1.9.0 (QEMU+KVM)</td>
</tr>
</tbody>
</table>
All experiments were conducted by deploying and running applications on a two node Kubernetes cluster deployed on Google Compute Engine VM instances. The master ran on a n1-standard-2 (2 vCPUs, 7.5 GB memory) machine and the worker ran on a n1-standard-8 (8 vCPUs, 30 GB memory) machine. These machines ran Ubuntu 16.04 with nested virtualization enabled and were configured to use the Intel Haswell CPU platform. All Pods ran only one container and each container was limited to 1 vCPU and 2GiB of working memory.

All container runtimes were installed and exposed for usage within the same Kubernetes cluster through the RuntimeClass Kubernetes feature gate [16]. This ensured that all experiments were conducted in the same hardware and software setting without the need of tearing down and reconfiguring the Kubernetes cluster for each runtime. This setup also serves as a proof of concept for multi-runtime Kubernetes clusters. All benchmarks were repeated 10 times. Performance-relevant configuration details are presented in Table II.

Applications:

**TeaStore** [17] is a microservice benchmark. It emulates a web store that features browsing, selecting and making orders of tea and tea supplies. It features five services each with different responsibilities. In the experiments, each service had only one replica. The benchmark score for TeaStore is defined as the average requests per second throughput for eight available operations in the TeaStore API such as adding and removing items or listing and browsing items. The operations are performed sequentially in a round robin fashion by two benchmark clients running simultaneously. Each client performed 200 of each request. Time to deploy was measured as the time from requesting all Kubernetes objects required to run TeaStore until each service reported itself as successfully running on the TeaStore status page. TeaStore deployment includes time consuming training of a ML model for a recommendation system in the web store.

**Redis** [18] is an in-memory data-store that features data structures such as hashes, lists, sets and more. It has built-in replication in a primary-replica manner. It can also provide high availability through a Redis Sentinel that monitors Redis instances and takes action to promote replicas to primary when a primary seems to be failing. For the benchmark, one primary, two replicas and two sentinels were deployed. The benchmark score for Redis is defined as request per second throughput for the O(1) GET operation. The Redis built-in benchmark tool redis-benchmark was used to perform and measure the throughput of the operations. The tool was configured to spawn 50 clients that each ran 16 operations in parallel until each operation was executed one million times. Time to deploy was measured as the time from requesting all Kubernetes objects required to run Redis until the Redis master was available for external requests.

**Spark** [19] is a distributed general purpose computing framework for big data processing. A Spark cluster was deployed with one master and 3 workers. The benchmark score for Spark is defined as the average amount of primes found per second when finding all prime numbers in the first million numbers. Apache Zeppelin [20] was used to submit workloads using PySpark, the Spark Python API. Time to deploy was measured as the time from requesting all Kubernetes objects required to run Spark until all workers had joined the master.

B. Experimental Results

Figs. 2 and 3 presents the results for deployment time and application performance, respectively. The x-axis represents
the three benchmark applications, while the y-axis presents the metric. All experiments are repeated 10 times, each box-plot represents the quartiles.

**Deployment Time:** When comparing the deployment of TeaStore to runC with the other runtimes, gVisor take about three times as long to deploy and Kata almost four times as long. The same relationship in deployment times are true for deploying Spark. The deployment time for Redis shows a slightly different ratio between the runtimes, with Kata and gVisor both deploying Redis close to twice the time of runC.

Given their architecture, these results are expected and represent the cost of launching two extra processes (Sentry and Gofer) for gVisor, as well as a VM and hypervisor for Kata.

**Application Performance:** When comparing application performance, gVisor and Kata achieved around 40% and 60%, respectively, of the runC score for TeaStore. A similar relationship exist for Spark, with Kata outperforming gVisor performance-wise. Regarding Redis performance, both gVisor and Kata scored a request throughput of around 20% of that of runC. This is due to the fact that Redis performance is measured using a simple GET requesting in-memory data. The low CPU and memory requirements of the operation itself implies that the application performance is mostly determined by networking performance.

**C. Overhead**

Besides impacting deployment and application performance, container runtimes also impose an overhead in terms of CPU and memory usage. We measured such overhead by running an idle container for five minutes and scraping resource usage metrics for the entire Pod using cAdvisor [21]. The results (not shown for briefness) highlight that the CPU overhead is negligible for all runtimes, whereas for memory this is not the case: gVisor imposes a 14 MB memory overhead and Kata 143 MB.

### IV. OUTLOOK

In this paper, we evaluate the security capabilities and performance of two alternative Kubernetes container runtimes: Kata and gVisor. Overall, our results show that higher security comes at a very high cost, both in terms of deployment time and application performance loss. When focusing on the two security-hardened runtimes, Kata – which provides the most security layers – also features the highest deployment time, but surprisingly – better application performance than gVisor, both for a compute-bound and network-bound applications.

Increasing the security of container runtimes by adding layers of indirection is still a fast-evolving landscape. Scholars need to regularly evaluate said landscape to better understand trends, design trade-offs and fundamental limits. For our study, Kata and gVisor were selected due to their plug-and-play compatibility with Kubernetes, but other container runtime – in particular Firecracker and Nabra – are soon expected to challenge their favored position. In the future, although each container runtime is expected to conceptually stay the same, the underlying technology that they use may change, significantly altering their performance. For example, gVisor is switching from ptrace-based syscall interception to KVM.

Although the performance of security-hardened container runtimes will increase as the technology matures, it is unlikely that there will be a one-size-fits-all container runtime. Therefore, besides comparing the container runtimes in a competition, future work should also focus on how container runtimes can complement each other, perhaps providing the user with a wider range of choices than only between strong security and high performance.

### REFERENCES

Infrastructure-Aware TensorFlow for Heterogeneous Datacenters

Moiz Arif†, M. Mustafa Rafique†, Seung-Hwan Lim‡, Zaki Malik§
†Rochester Institute of Technology, ‡Oak Ridge National Laboratory, §Texas A & M University – Commerce
†{ma3890, mrafique}@cs.rit.edu, ‡lims1@ornl.gov, §Zaki.Malik@tamuc.edu

Abstract—Heterogeneous datacenters, with a variety of compute, memory, and network resources, are becoming increasingly popular to address the resource requirements of time-sensitive applications. One such application framework is the TensorFlow platform, which has become a platform of choice for running machine learning workloads. The state-of-the-art TensorFlow platform is oblivious to the availability and performance profiles of the underlying datacenter resources and does not incorporate resource requirements of the given workloads for distributed training. This leads to executing the training tasks on busy and resource-constrained worker nodes, which results in a significant increase in the overall training time. In this paper, we address this challenge and propose architectural improvements and new software modules in the default TensorFlow platform to make it aware of the availability and capabilities of the underlying datacenter resources. The proposed Infrastructure-Aware TensorFlow efficiently schedules the training tasks on the best possible resources for execution and reduces the overall training time. Our evaluation using the worker nodes with varying availability and performance profiles shows that the proposed enhancements yield up to 54% reduced training time as compared to the default TensorFlow platform.

Index Terms—Distributed TensorFlow; heterogeneous datacenters; datacenter resource management; datacenter utilization

I. INTRODUCTION

Large scale datacenters typically comprise tens and thousands of servers, where servers are generally added in stages over the years to meet user requirements. This leads to performance and architectural heterogeneity [1], which must be addressed in managing datacenter resources. Most software platforms are designed for homogeneous datacenters and lead to sub-optimal performance when executing in heterogeneous environments [2]–[4]. Datacenter resources are typically shared between software platforms that run diverse workloads with unique execution profiles and resource requirements [5]. However, most software platforms do not incorporate resource heterogeneity and application performance requirements while scheduling workloads on the available resources.

TensorFlow [6] is an open-source platform that is widely used for executing machine learning (ML) and deep learning (DL) applications. It maps worker nodes to a dataflow graph across datacenter that may contain multi-core central processing units (CPUs), programmable graphics processing units (GPUs), and custom-designed application-specific integrated circuit (ASIC) [7] known as Tensor Processing Unit (TPU) [8]. TensorFlow partitions the computational graph of the ML model into subgraphs, which are then scheduled for execution on the computing devices. The master node is responsible for optimizing the graph and coordinating the execution of subgraphs across different tasks. The application developer defines computation as a dataflow graph and initiates its execution on the given worker nodes.

The placement algorithm of TensorFlow follows a cost model that calculates input and output tensor sizes along with the estimated computation time for each subgraph of the computational graph. The cost model utilizes statistical estimation based on heuristics associated with different operation types and the available historical data. TensorFlow simulates scheduling the computational graph to obtain a device placement map. The placement strategy uses a greedy heuristic to estimate the execution time of all available devices on the worker nodes, and the actual execution time is compared with the estimated time. Finally, the device with the lowest execution time is selected for execution, and this process continues for all subgraphs. When a TensorFlow job is submitted, the master node divides it into smaller tasks and schedules them on the available servers, and aggregates the results after the job is completed. However, it does not incorporate heterogeneous network, CPU, GPU, and available memory at each node, which leads to non-optimal scheduling of ML tasks on datacenter resources.

In a distributed environment, TensorFlow works similarly to single machine multi-device environments. However, in a distributed setting, inter-worker communication is periodically required for updating the model parameters. If a sub-graph is assigned on a busy or straggler node for execution, then it will increase the overall execution time of the training job [9]. The overloaded workers lack sufficient resources to execute the task effectively, and the slowest worker node determines the total execution time for the given ML job. Application developers cannot control the execution of sub-graphs or define resource requirements, e.g., number of GPUs, amount of system and GPU memory, and network bandwidth, for a training job.

In this paper, we extend the TensorFlow platform and incorporate the information about the performance profiles and the availability of the underlying heterogeneous datacenter resources without requiring resource orchestration from the application developer. Our optimizations make the TensorFlow platform aware of the available infrastructure resources and enable it to effectively manage training task distribution and assignment to the ideal worker nodes. We rank each worker...
according to the available resources, i.e., CPUs, GPUs, memory, and network, and the master node uses this ranking for selecting the appropriate worker nodes to execute the given ML workload. Furthermore, the master node keeps track of the past ML workload execution times and leverages this information to identify the straggler nodes in the datacenter. This enables the TensorFlow platform to avoid using the straggler nodes and reduce the overall execution time of the given ML workload.

Specifically, we make the following contributions.

- We propose a design for Infrastructure-Aware TensorFlow platform that extends off-the-shelf TensorFlow platform by adding four key components, namely, Workload Specification Module, Resources Module, Monitoring Module, and Infrastructure Module. Our proposed design alleviates application developers from managing the underlying datacenter resources and enables full utilization of the available resources.
- We provide a prototype implementation of the proposed platform in a datacenter environment and successfully integrate it with the distributed TensorFlow platform to demonstrate its effectiveness.
- We conduct a performance evaluation of the proposed platform and compare it with an ideal and a constrained execution environments using the default TensorFlow. Our evaluation shows that the proposed platform achieves up to 54% performance improvements over the constrained TensorFlow environment.

II. MOTIVATION

A. Heterogeneity in Datacenters

Large-scale datacenters are typically equipped with a variety of compute, software, storage, and networking resources with different capabilities and performance profiles. Heterogeneity in datacenters is inevitable because of the continuous technological advancements and hardware component upgrades [1] leading to infrastructure-level performance variations. This impacts the performance [10] of complex workflows and large-scale distributed applications, which can be avoided by making them aware of the performance variations of the underlying datacenter resources. If an application is aware of the underlying performance heterogeneity, then it may adapt and schedule its tasks based on the performance profiles of the hardware resources and the resource requirements of various application tasks.

B. Limitations of TensorFlow for Managing Resource Heterogeneity and Availability

The TensorFlow platform supports conventional multi-core processors and computational accelerators. However, it does not incorporate underlying infrastructure type and resource capabilities while executing the training process. If the training jobs are scheduled on the worker nodes that are already running other tasks, then the overall training time would increase for the scheduled jobs. Similarly, training jobs scheduled on the straggler worker nodes will significantly increase the training time [11], [12] as all parameters of the given ML model must be gathered and updated from each node after every iteration. This may lead to missed deadlines for a time-sensitive training process [13], [14]. Application developers can hard code worker nodes and devices for training the given model using the default TensorFlow, which is not a scalable approach in a large-scale datacenter that is shared by many users. Moreover, this leads to poor utilization of datacenter resources as it is practically impossible for developers to accurately execute the training workload on idle resources for distributed training.

We conducted experiments to study the impact of existing system load on the execution time of ML jobs using different batch sizes. Figure 1 shows the result of this experiment using ResNet32 [15] model with the CIFAR10 [16] dataset. We run jobs that utilize 10% CPU, 20% GPU, 4 GB of main memory, and 2.6 GB of GPU device memory to introduce background load on the worker nodes. We run multiple instances of these jobs to achieve different background load levels and report the average load as a combination of CPU, GPU, memory, and network resource utilization. The training time is increased by 37%, 43%, 45%, and 41%, as the background load increases from 0% to 70% for the batch size of 64, 128, 256, and 512, respectively. We observe similar performance trends with other ML models during our evaluation. This shows that it is critical to execute the training jobs on resources with minimum interference from other tasks.

III. INFRASTRUCTURE-AWARE TENSORFLOW DESIGN

A. Design Objectives

The main goal of the Infrastructure-Aware TensorFlow platform is to improve the performance of TensorFlow in a distributed setting and enable it to handle the underlying performance heterogeneity. The key objectives of the proposed platform are as follows:

1) Make TensorFlow aware of the underlying infrastructure in heterogeneous datacenters. This will enable TensorFlow to make efficient resource management decisions, and will result in reduced training time.

2) Isolate and limit the impact of busy and straggler worker nodes in large datacenters on the performance of distributed training. This will significantly improve the performance of the training process.
3) Optimize TensorFlow scheduling by incorporating information about the underlying resources and worker nodes. This will improve datacenter utilization by scheduling jobs on idle resources.

TensorFlow provides several strategies for training a given model in a distributed setting depending upon the ML model and the underlying physical resources. The mirrored strategy [17] supports synchronous distributed training on multiple GPUs on a single machine. Similarly, multi-worker mirrored strategy [18] implements synchronous distributed training across multiple workers, where multiple GPUs are potentially available at each worker. In this paper, we use the multi-worker mirrored strategy because of its popularity and wider adaptation in the GPU-enabled datacenters [19], [20].

B. Design Overview

In this paper, we extend the TensorFlow platform with new software components to collect and provide the latest information about datacenter resources. Our approach involves gathering CPU, GPU, memory, and network utilization metrics from the worker nodes, and enable TensorFlow to incorporate them in executing ML jobs. Figure 2 shows the proposed architecture of the Infrastructure-Aware TensorFlow platform. We develop a Workload Specification Module to enable the application developers to specify the resource requirements of the given TensorFlow workload. We use a Resources Module to collect information about the hardware resources, e.g., CPUs, number and type of installed GPUs, and the amount of installed memory, at each worker node. We introduce a Monitoring Module to gather the latest resource utilization information from the worker nodes. Finally, we develop an Infrastructure Module that leverages the information from the Workload Specification, Resources, and Monitoring modules to execute the given ML job on the distributed resources.

C. Infrastructure-Aware TensorFlow Architecture

Figure 3 shows the workflow of the proposed platform. At startup, all modules are initialized and synchronized with the database that is used to store the updated resource information. After the initialization phase, the Resources Module starts and captures the information of all worker nodes in the datacenter. Next, the Monitoring Module starts and captures the current utilization of the workers that have been identified by the Resources Module. When a new TensorFlow job is submitted by the user through the Workload Specification Module, the Infrastructure Module fetches the up-to-date information about the underlying infrastructure from the database and then selects appropriate workers to execute the given workload. This decision, along with the job execution information, is stored back in the database for future use.

In the following, we describe details of new software modules that we have added in the TensorFlow platform to make it aware of the underlying resource availability and efficiently selecting the worker nodes.

1) Workload Specification Module: The Workload Specification Module enables the user to specify the resource requirements of the given TensorFlow job. To enable users to provide the resource requirements of ML jobs, we extend the existing TensorFlow configuration specification and add a section, resources, to obtain resource requirements of the submitted jobs. Listing 1 provides an example of an ML job configuration. The user can set the minimum number of CPU and GPU along with the minimum amount of memory, in Gigabytes, required to run the given job. The user can also specify the minimum network bandwidth, in Gbps, that should be available at the worker nodes to run the given ML job. These resource requirements are read by the Workload Specification Module and are used by the Infrastructure Module for selecting the suitable worker nodes to run the given ML job.

2) Resources Module: The Resources Module captures the information about the compute, accelerator, memory, and network utilization metrics from the worker nodes. This information is stored in the database. The Monitoring Module periodically updates the database with the latest utilization information of the worker nodes.

Listing 1: A sample configuration file to specify the resource requirements of a TensorFlow job.
network resources that are installed at each worker node and stores them in the database. It maintains a table of resources associated with each server. The Resources Module is responsible for the following:

- **Maintain Worker Inventory:** The Resources Module builds a list of all worker nodes that are available to the distributed TensorFlow environment. Each entry of the list contains hostname, IP address, and an optional port number that may be required for monitoring the resource usage of a worker node.

- **Capture Resource Information:** The Resources Module uses the information in the worker inventory and captures information about the installed hardware resources, i.e., CPU and GPU count along with their capabilities, system memory, and network link speed, at each worker node. If a worker node is updated with new hardware, then the Resources Module automatically detects the changes in the installed hardware and updates its record accordingly. This is achieved by using the information collected by the Monitoring Module, which continuously monitors the underlying resources at each worker node.

3) **Monitoring Module:** The Monitoring Module runs at each worker node for monitoring key resources. We have developed a lightweight python utility that periodically collects the CPU, GPU, memory, and network utilization metrics, and stores them in the configured database. We set the initial monitoring frequency to 10 seconds to collect these metrics. However, to reduce the monitoring overhead, monitoring frequency is automatically adapted based on the local resource utilization. Specifically, if the Monitoring Module detects a decrease in CPU utilization of the master node, it increases the monitoring frequency to collect more resource utilization data. Note that our Monitoring Module can be replaced with other popular monitoring tools, e.g., Ganglia [21] and Nagios [22], which provide application programming interfaces (APIs) for integration with other software.

4) **Infrastructure Module:** The Infrastructure Module is the core component that uses the information provided by the other modules to schedule the given ML job. As the default TensorFlow does not leverage the latest resource utilization metrics of the worker nodes, it may schedule ML jobs on worker nodes with busy CPU and GPU resources, limited available memory, and network bottlenecks. The Infrastructure Module enables TensorFlow to incorporate the latest resource utilization metrics to make an informed decision while selecting a suitable worker to schedule the given ML job.

   a) **Identifying Busy or Straggler Nodes:** The Monitoring Module maintains historical monitoring data for the workers, which is used by the Infrastructure Module to identify workers that repeatedly exceed resource utilization thresholds. This information is correlated with the current resource utilization metrics reported by the Monitoring Module and the job completion statistics stored in the database. The historical monitoring data is used to effectively determine if a worker can experience high resource utilization during the execution of an ML job. The Infrastructure Module categorizes a worker as a straggler if its historic and current resource utilization exceeds the predefined threshold. The information about the stragglers are stored back in the database and is utilized during the worker selection process.

   b) **Worker Nodes Selection for Executing TensorFlow Jobs:** Algorithm 1 shows the approach used by the Infrastructure Module for selecting the required worker nodes for executing the given ML job. The Infrastructure Module fetches the list of worker nodes from the database, the information about the installed resources at each worker, and the up-to-date list of the straggler nodes. The filtering phase excludes straggler nodes from the list of potential workers to execute the given ML job. Next, the workers that do not have enough resources installed to meet the specified resource requirements are excluded from executing the given job. Next, the current resource utilization values are obtained for each resource from the worker nodes. Those workers which have higher resource utilization than the adaptive threshold are excluded from executing the given job. The threshold values are based on the historical job profiling data available to the Infrastructure Module through the Monitoring Module. Finally, all remaining worker nodes undergo a weighing and scoring process and are assigned numeric scores based on the current resource utilization metrics. We use predefined numeric multipliers as weights to normalize the impact of each resource utilization.

Algorithm 1: Workers Selection in the Infrastructure-Aware TensorFlow.

**Input:** worker nodes \((N)\), job resource requirements \((R)\)

**Output:** List of ideal worker nodes.

begin
1. Get list of straggler nodes \((\text{stragglers})\)
2. Get \(\text{cpu}_{\text{req}}, \text{mem}_{\text{req}}, \text{net}_{\text{bw}}, \text{gpu}_{\text{req}}\)
3. Get \(\text{cpu}_{\text{tot}}, \text{mem}_{\text{tot}}, \text{net}_{\text{bw}}, \text{gpu}_{\text{tot}}\)
4. Get \(\text{cpu}_{\text{th}}, \text{mem}_{\text{th}}, \text{net}_{\text{th}}, \text{gpu}_{\text{th}}\)
5. for all workers \(n_j \in N\) do
6. if \(n_j \in \text{stragglers}\) then
7. Skip this node and continue to the next node
8. else
9. Retain worker node
10. end
11. Get \(\text{cpu}_{\text{tot}}, \text{mem}_{\text{tot}}, \text{net}_{\text{bw}}, \text{gpu}_{\text{tot}}, \text{gpu}_{\text{mem}_{\text{tot}}}\)
12. if \(\text{cpu}_{\text{tot}} \geq \text{cpu}_{\text{req}}\) AND \(\text{mem}_{\text{tot}} \geq \text{mem}_{\text{req}}\) AND \(\text{net}_{\text{bw}} \geq \text{net}_{\text{req}}\) AND \(\text{gpu}_{\text{tot}} \geq \text{gpu}_{\text{req}}\) then
13. Retain worker node
14. else
15. Skip this node and continue to the next node
16. end
17. Get \(\text{cpu}_{\text{util}}, \text{mem}_{\text{util}}, \text{net}_{\text{util}}, \text{gpu}_{\text{util}}, \text{gpu}_{\text{mem}_{\text{util}}}\)
18. if \(\text{cpu}_{\text{util}} \geq \text{cpu}_{\text{th}}, \text{OR} \text{mem}_{\text{util}} \geq \text{mem}_{\text{th}}, \text{OR} \text{net}_{\text{util}} \geq \text{net}_{\text{th}}, \text{OR} \text{gpu}_{\text{util}} \geq \text{gpu}_{\text{th}}, \text{OR} \text{gpu}_{\text{mem}_{\text{util}}} \geq \text{gpu}_{\text{mem}_{\text{th}}}\) then
19. Skip this node and continue to the next node
20. else
21. Retain worker node
22. end
23. \(\text{score}_{n_j} = \text{cpu}_{\text{util}} \times \text{cpu}_{\text{tot}} + \text{mem}_{\text{util}} \times \text{mem}_{\text{tot}} + \text{net}_{\text{util}} \times \text{net}_{\text{bw}} + \text{gpu}_{\text{util}} \times \text{gpu}_{\text{tot}}\)
24. end
25. Sort score for all workers in descending order
26. Select and return top \(w\) workers from sorted list, where \(w\) represents the number of workers specified in \(R\)
The normalized value is used as a score for each resource, such that a higher score refers to lower resource utilization. The final score for a node is calculated by aggregating the scores of its individual resources. The worker with the highest aggregated score has the most available resources and is the ideal node to run the given ML job to ensure an even resource utilization across all worker nodes. In the end, the final list of candidate worker nodes is sorted based on their aggregated scores, and the required number of nodes are selected based on the given resource specification of the given ML job.

IV. PERFORMANCE EVALUATION

In this section, we present the evaluation of the proposed Infrastructure-Aware TensorFlow platform. We explain our experimental setup, the performance metrics, and alternative environments that we use to evaluate the effectiveness of the proposed platform.

A. Evaluation Setup

Our evaluation setup consists of eight Dell PowerEdge R730 servers having two 3.20 GHz Intel Xeon E5-2670 v3 processors, 128 GB main memory, and two NVIDIA P100 GPUs. Our setup uses a 10G network interconnect between the servers. We run a distributed TensorFlow environment on these servers using Ubuntu 18.04 LTS server operating system. We use multiple TensorFlow jobs to create background noise on the TensorFlow worker nodes. We use MNIST [23], ImageNet [24] and CIFAR10/100 [16] datasets and Keras [25], ResNet32/56 [15], Inception-V1 [26] and MobileNet [27] models to evaluate the performance of distributed training using different execution environments.

For evaluating our proposed Infrastructure-Aware TensorFlow platform on varying load scenarios, we overload the servers by increasing the CPU, GPU, memory, and network utilization. To this end, we use netem [28] along with ethtool [29] to simulate different network constraints on the links between the master and worker nodes. Moreover, we use stress [30], which is a Linux utility to overload CPU and memory on the worker nodes to mimic the behavior of stragglers and busy nodes in the datacenter.

In our evaluation, we consider the total execution time, i.e., the time required to train an ML model, as the performance metric to compare the performance of the studied approaches.

B. TensorFlow Execution Environments

We use three flavors of the TensorFlow environment in our evaluation. These flavors showcase various capabilities of the TensorFlow platform to address changing resource availability in datacenter settings. These environments are:

1) Unconstrained TensorFlow Environment: This environment does not have any resource constraints and is based on the ideal scenario that there is no resource sharing between applications and that all of the resources are dedicated to run a single workload. We use the default TensorFlow platform to run the given ML job in this environment.

2) Constrained TensorFlow Environment: This environment is subjected to resource constraints, such as increased CPU, GPU, memory, and network bandwidth utilization of the heterogeneous datacenter resources. We use the default TensorFlow platform to run the given ML job in this environment. We run background jobs that utilize 10% CPU, 20% GPU, 4 GB of main memory, and 2.6 GB of GPU memory to introduce constraints on the worker nodes. We run multiple instances of these background jobs to mimic different load scenarios.

3) Infrastructure-Aware TensorFlow Environment: This is the same as the constrained TensorFlow environment, however, we use a hand-tuned implementation of the proposed Infrastructure-Aware TensorFlow platform to run the given ML job.

C. Performance Results

In this section, we execute the studied benchmarks on the considered execution environments and compare their performance. We repeat each experiment five times and report the average execution time for running the given ML job.

1) Training Makespan using Multiple Nodes: We ran several experiments using a combination of ML models and datasets using a varying number of iterations, batch sizes, and epochs and recorded the total execution time for the studied environments. Figure 4 shows the performance comparison of the unconstrained, constrained, and Infrastructure-Aware TensorFlow environment using a batch size of 64 to train the studied models. For all ML models, we observe that the constrained environment takes the longest time to complete the training process due to the resource constraints applicable to the worker nodes and the inability of the default TensorFlow platform to execute the given job. The proposed Infrastructure-Aware TensorFlow is cognizant of the availability limitations of underlying resources and avoids scheduling the jobs on the affected worker nodes. We observe similar performance trends when using the training batch size of 128 and 256, as shown in Figure 5 and Figure 6, for the models used in our evaluation. Figure 7 shows the performance comparison using the batch size of 512. Here, we observe that the Inception-V1 model fails to execute using the constrained TensorFlow environment because of the resource constraints. On average, the execution time of the Infrastructure-Aware TensorFlow for the studied models is 37% more than the unconstrained (dedicated) environment. However, on average, the execution time for the Infrastructure-Aware TensorFlow is 24% less than the constrained TensorFlow environments for the studied models. This happens because the straggler node delays the execution of the assigned training task and results in increased overall training time in the constrained TensorFlow environment. Other worker nodes that depend upon the straggler node for aggregating the model parameters incur execution stalls, thus leading to the reduced overall performance of the ML jobs.

2) Impact of Available Network Bandwidth: In this experiment, we study the impact of available network bandwidth on
the performance of the Infrastructure-Aware TensorFlow and compare it with the default TensorFlow platform. Figure 8 shows the result of this experiment using ResNet32 with the CIFAR10 dataset. We observe similar performance trends with other models and datasets. Here, we limit the available bandwidth on the network link from 1 Gbps to 100 Mbps of a worker node and observe that the overall training time significantly increases with the decrease in the available network bandwidth as expected. The default TensorFlow platform does not incorporate busy network links in scheduling the training tasks to the worker nodes. This exponentially increases the overall model training time as the network link gets fully congested. However, the Infrastructure Module of the proposed platform addresses this constraint by incorporating the available bandwidth metric reported by the Monitoring Module and assigns appropriate weights during the scoring phase. This enables our TensorFlow platform to exclude a possibly powerful worker from executing the ML job as it will increase the overall training time due to network congestion between the worker and the master nodes. On average, the proposed TensorFlow platform result in a 42% reduction in the overall training time when the available network bandwidth is between 100 Mbps and 700 Mbps.

3) Impact of Increased Network Latency: We study the impact of increased network latency on the performance of the studied ML models using unconstrained, constrained, and Infrastructure-Aware TensorFlow platforms. Workers that are far away from the master node add additional latency, which leads to a significant increase in the overall execution time. Figure 9 shows the result of this experiment where additional network latency is added to one of the worker nodes using the netem tool to mimic the behavior of a straggler node. We report the training performance of the ResNet32 model using the CIFAR10 dataset, however, we observe similar performance trends with other models and datasets that we use in this evaluation. The data point where the induced network latency is 0 millisecond using the constrained TensorFlow captures the environment of the unconstrained TensorFlow platform. We observe that the training time increases significantly with the increase in the network latency between the worker and master nodes. This shows that the distributed training process is highly sensitive to variations in the network latency between the worker and the master nodes. Infrastructure-Aware TensorFlow can handle heterogeneous network latencies between the worker nodes and avoids scheduling training jobs on workers with increased network latencies. The Infrastructure Module takes into account the latency information between the nodes and uses this information to exclude the straggler nodes from the list of available worker nodes. We observe that increased latency causes additional time to fetch the updated model parameters from the worker nodes. However, in the unconstrained TensorFlow environment, the worker nodes are selected regardless of their current network latencies, which leads to increased overall model training time. Overall, the proposed Infrastructure-Aware TensorFlow performs 52%, 54%, 41%, and 36% better than the constrained TensorFlow environment for model training when using the batch size of 64, 128, 256, and 512, respectively, with increased network latency between the worker and the master node.
In this experiment, we study the impact of available GPU resources on the overall training time. We use resource-intensive TensorFlow jobs to overload the GPU resources at the worker nodes before starting the actual training jobs. We use `nvidia-smi` [31] to monitor GPU utilization, and run background TensorFlow jobs to saturate its utilization to specific levels, and analyze the behavior of the TensorFlow platform when GPUs are overloaded with other tasks. Figure 10 shows the result of this experiment where we compare the performance of constrained and Infrastructure-Aware TensorFlow platforms. Here, we report the performance of the ResNet32 model with the CIFAR10 dataset. However, we observe similar performance trends with other ML models and datasets used in this evaluation. The scenario where GPU utilization is 0% at the constrained TensorFlow represents the performance of the TensorFlow platform when GPUs are overloaded with other tasks. Although it incorporates various compute resources, i.e., CPU and GPU, it does not incorporate other performance metrics, e.g., GPU capabilities, memory, and network latency, while making the placement decisions.

Previous efforts have explored different aspects to improve the performance of the TensorFlow platform. However, they do not holistically address the challenges of resource and performance heterogeneity. In this paper, we address this by proposing enhancements to the TensorFlow platform to capture the heterogeneity and availability of the underlying resources.
VI. CONCLUSION

In this paper, we present an Infrastructure-Aware TensorFlow platform that extends the popular TensorFlow platform by adding new software modules to incorporate the heterogeneity and availability of the underlying datacenter resources for executing machine learning (ML) jobs on distributed worker nodes. The newly added modules enable application developers to specify the resource requirements of the given ML job, capture the installed heterogeneous resources on the worker nodes, consider the latest monitoring information from the worker nodes, and schedule the given ML jobs on heterogeneous datacenter resources by incorporating their resource requirements and the availability of the resources at the worker nodes. Our evaluation shows that the proposed Infrastructure-Aware TensorFlow reduces the overall execution time by up to 54% as compared to the default TensorFlow platform for scenarios when a limited amount of compute, graphics processing unit (GPU), memory, and network resources are available at the worker nodes. In our future work, we will extend the proposed platform to handle the presence of a large number of straggler nodes in scheduling and executing the given ML jobs. Moreover, we will explore techniques for adapting the proposed platform to other distributed training approaches, such as parameter-server based distributed training.

ACKNOWLEDGMENT

Results presented in this paper are obtained using the Chameleon and CloudLab testbeds supported by the National Science Foundation. This manuscript has been authored by UT-Battelle, LLC under Contract No. DE-AC05-00OR22725 with the U.S. Department of Energy (DOE). The U.S. Government retains and the publisher, by accepting the article for publication, acknowledges that the U.S. Government retains a non-exclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or to allow others to do so, for U.S. Government purposes. The DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (https://www.energy.gov/downloads/doe-public-access-plan).

REFERENCES


Model-Aided Learning for URLLC Transmission in Unlicensed Spectrum

Ayat Zaki-Hindi$^{1,2}$, Salah-Eddine Elayoubi$^3$, Tijani Chahed$^2$

$^1$ Orange Labs, 44 Avenue de la Republique, 92320 Chatillon, France
$^2$ Institut Polytechnique de Paris; Telecom SudParis, 19 Place Marguerite Perey, 91120 Palaiseau, France
$^3$ Université Paris Saclay, CentraleSupelec, CNRS, 3 Rue Joliot Curie, 91190 Gif-sur-Yvette, France

ayat.zakhindi@orange.com, salaheddine.elayoubi@centralesupelec.fr, tijani.chahed@telecom-sudparis.eu

Abstract—We focus in this paper on the transport of critical services in unlicensed spectrum, where stringent constraints on latency and reliability are to be met, in the context of Ultra-Reliable Low Latency Communication (URLLC). Since contention-based medium access performs poorly in the case of high traffic load, we propose a new transmission scheme where the transmitter can increase its transmission power when the delay of the packet approaches the delay constraint, increasing by that its chance of being decoded even in case of collision with other lower-power packets. We are however interested in minimizing the usage of high power transmissions, mainly to conserve energy for battery-powered devices and to limit the range of interference. Therefore, we define a transmission policy that makes use of a delay threshold after which the high-power transmission starts, and propose a new online-learning approach based on Multi-Armed Bandit (MAB) in order to identify the policy which achieves minimum energy consumption while guaranteeing reliability. However, we observe that the MAB converges slowly to the optimal policy because the loss event is rare in the load regime of interest. We then propose a model-aided learning approach where a simple analytical model helps estimating the long-term reliability resulting from an action and thus its reward. Our results show a significant enhancement of the convergence towards the optimal policy.

I. INTRODUCTION

Ultra-Reliable Low-Latency Communication (URLLC) is a promising class of services proposed in 5G with very stringent reliability and latency requirements, in order to enable industrial applications such as factory automation or vehicular communications. URLLC deployment is considered very challenging since the requirements are on the order of 99.999% and 1ms for reliability and latency, respectively, increasing the cost of deployment of the network, especially when coexisting with other 5G services like enhanced Mobile Broadband (eMBB). To reduce this cost, unlicensed spectrum is advocated in some industrial environments rendering the challenge even more difficult. The interest in unlicensed spectrum has begun since LTE Licensed-Assisted-Access (LTE-LAA) and 5G has shown its flexibility by regulating several unlicensed bands in the 5G New Radio (NR), notably the 5GHz bands currently used by some IEEE 802.11 technologies.

Most works consider URLLC performance in licensed spectrum, where transmissions occur in a frame-based system, like LTE, within transmission time intervals (TTI). This procedure is however not suitable for URLLC as the rhythm of (re-)transmissions and feedback depends on the size of the TTI, which can easily violate the reliability and latency constraints. For this, optimal power control for uplink grant-free URLLC transmission in licensed spectrum was studied in [1].

Regarding unlicensed spectrum, [2] reviews and evaluates different PHY and MAC layer mechanisms to demonstrate their potential benefits to URLLC. Uplink URLLC transmission in industrial scenarios was discussed in [3] and [4] proposing licensed-aided transmission where supplementary licensed bands are used in order to increase the system reliability when unlicensed transmission fails.

We consider in this paper the transport of URLLC on unlicensed spectrum only and propose a new transmission scheme where the transmitter can increase its transmission power as the delay of the packet approaches the imposed constraint. This behaviour is to remain as a last resort only, and this for two reasons: first, to minimize the energy consumption of the transmitters as the devices in industrial environments are mostly battery-powered, and second, in order to limit the impact on other services/users in terms of interference. Hence, our goal is to find a dynamic transmission policy which minimizes the utilization of high power transmissions while guaranteeing the stringent performance metrics.

To achieve this aim, in general, a central entity has to collect information about the performance of all stations and solve high order and complex optimization problems, in order to determine the optimal transmission policy for each one of them. This becomes prohibitive in the presence of a large number of connected objects. Therefore, we advocate in this paper the use of online learning to solve such problem efficiently, where each transmitter adapts its policy individually, in a distributed manner, without the help of a central unit.

In online learning, the learner attempts to identify the action which maximizes a certain expected reward over a time horizon depending on the feedback from the environment to that action. When the action causes the
environment to change (move to another state), then the learning process can be solved using reinforcement learning (RL) for instance, which involves mainly solving a Markov Decision Process (MDP) [5]. However, solving complex MDPs is unnecessary in many cases (similar to ours) where the feedback of the environment is not very dependent on the learner’s actions, due to the large number of devices. In this case, Multi-Armed Bandit (MAB) becomes the best candidate to solve such problems. Stochastic MAB is a special case of RL (it is equivalent to a one-state MDP), where each action is denoted by an arm and is associated to a reward distribution that is unknown to the learner. In our case, the action is represented by the transmission policy (switching moment between low and high power regimes). In every round, i.e. at each packet generation, the learner picks an arm and receives an observation from the reward distribution of that arm; it refines its knowledge about the arm by selecting it for a sufficient number of times. At the same time, the learner desires to identify the best arm as soon as possible in order to maximize its cumulative reward over a horizon of rounds. MAB illustrates the famous learning trade-off exploration versus exploitation.

For applications where some measures are linked to rare events, as does reliability in our case (it has to remain below $10^{-5}$), it is hard to estimate efficiently the reward, which renders the learning process very slow, i.e., the exploration phase requires a very large number of rounds to collect accurate estimates of the reward distribution for all arms. This drawback originates from the fact that MAB (and learning in general) deals with the environment as a black box, and the observations in one round are exclusive to the picked arm. One possibility for accelerating the learning process, that we adopt in this paper, is to exploit our knowledge of the environment and the system to model the metrics that are hard to assess, and incorporate them in the learning algorithm.

Accelerating the learning speed using prior knowledge of the environment model is studied in [6] for the case of RL which accelerates the solution of the MDP whilst increasing the computation complexity as well. In our case, we do not have an MDP. Other works [7] dealing with rare events in the context of RL refer to reaching some rare states, not events, where importance sampling is used to capture the rare events in a system simulator.

To the best of our knowledge, the current work is the first to employ the knowledge of the system model to accelerate the learning process in MAB in the presence of "rare" events. We specifically apply this approach to learn the best transmission policy of different power levels to our scenario of URLLC coexistence with eMBB in unlicensed spectrum. We model the evolution of the system under the LBT mechanism using renewal theory, use the model for computing the long-term loss rate for the observed environment parameters, and fit the estimated loss rate into the reward of the selected action.

The rest of the paper is organized as follows. Section II describes the system model of LBT and the transmission policy in our bi-level power transmission case. Section III formulates the problem using multi-armed bandit and describes the considered MAB algorithms. Section IV derives a model for the loss rate under delay constraint and proposes the new model-aided MAB algorithms based on estimating some environment parameters. In section V, we compare numerically the performance of classical versus model-aided MAB algorithms and demonstrate the learning speed enhancement achieved by the latter. We eventually conclude the paper in section VI.

II. System model

Our scenario includes a set of $N_u$ automated machines in an industrial area, communicating URLLC packets in the uplink to a base station (BS). Time is slotted and we express all temporal quantities in time slot unit. Packets are generated sporadically with probability $1 - p_0$ per time slot per station. These small-sized packets need $x_u$ time slots to be transmitted to the BS, and are to be transmitted within a delay budget denoted by $T$; or else, the packet is considered lost. The reliability constraint denoted by $1 - \Lambda$, translates to a limited loss rate of $\Lambda$. On the other hand, other types of traffic might be present on the same frequency resources, originating from the machines or from other users in the area. We consider $N_e$ saturated eMBB stations generating constantly bulky packets which need $x_e$ time slots to be transmitted: $x_e \geq x_u$.

A. Listen-Before-Talk

When competing to access the channel, all users have to follow Listen-Before-Talk (LBT), unless they occupy the channel for a very small percentage of time (1% in most of the unlicensed spectrum bands, below which Aloha could be used). LBT guarantees fairness among transmitters by the random choice of the contention window (CW) before every transmission. In every time slot, the station senses the medium and decrements its CW for idle slots only until it hits zero, where it transmits the packet without sensing and waits for the ACK/NACK from the BS (the absence of response after a given time is considered as NACK). In case of collision, the station repeats the previous procedure without exceeding a maximum number of trials, called stages. In our case, we assume that the number of stages is bounded by the time budget $T$, i.e., the station can attempt (re-)transmission as long as the packet delay does not exceed $T$. Note that the duration of a given stage is a random variable (rv) denoted by $\theta$ and depends on CW and the probability of sensing the medium idle or busy, as illustrated in the derivation of its distribution proposed in [8].

The priority of a packet transmitted using LBT can be determined by CW, where a smaller range of choices prioritizes the packet since it increases its chance of being transmitted faster, but at the same time it increases the...
probability of collision with other stations, if all stations adopt smaller ranges. We consider LBT cat3 in our scenario, which means that the CW range is constant for the packet for all its attempts. We denote the maximum CW size for URLLC and eMBB packets by \( W_u \) and \( W_e \), respectively.

### B. Transmission policy

As indicated above, we consider two possible transmission power levels: \( P_1 \) and \( P_2 \); \( P_1 < P_2 \). We denote by \( q_1 \) and \( q_2 \) the corresponding collision probabilities \( (q_1 \geq q_2) \) and by \( C_1 \) and \( C_2 \) the corresponding costs \( (C_1 < C_2) \). More power levels can be considered, but in practice, received power levels should be distinguishable enough in order to decode the packet with the highest one. For simplicity, we ignore the difference between received and transmitted power levels which can be affected by the channel conditions. We consider that eMBB transmission power is always set to \( P_1 \), and \( P_2 \) is exclusively used by URLLC stations to increase the chance of their packets being decoded in case of collision with packets using \( P_1 \) transmission power level. Collisions between two or more packets using \( P_2 \) transmission power level induce the loss of all the collided packets. eMBB performance is not affected by the different power levels.

From the standpoint of a URLLC station competing to access the medium, the end of every stage can be seen as a future opportunity for transmitting the packet. Each time an opportunity arises, the packet is transmitted and, if the transmission does not succeed, it has to wait for a future opportunity.

We define the policy \( \tau \) by the pattern \( P_\tau(t) \) which determines the transmitting power level if a transmission opportunity arises at time \( t \) after the generation of the packet. \( \tau \) represents a switching time before which the station transmits at power \( P_1 \) and after which at power \( P_2 \). \( P_\tau(t) = P_1 1_{t \leq \tau} + P_2 1_{t > \tau} ; \tau \in \{ 0, 1, ..., T \} \). Note that \( \tau = 0 \) corresponds to the case where all transmission is done using \( P_2 \) and \( \tau = T \) is when all transmission is done using \( P_1 \).

We denote the number of opportunities in \([0, \tau]\) and \([\tau, T]\) by \( n_1(\tau) \) and \( n_2(\tau) \) respectively. \( n_1(\tau) \) and \( n_2(\tau) \) are two independent random variables (rvs) which depend directly on the stage duration denoted by \( \theta \). In the sequel, we refer to \( \theta \) as the inter-transmission time.

Figure 1 illustrates the described transmission policy.

As indicated previously, the energy consumption is to be minimized, and hence we have to find the optimal policy which consumes the minimum energy needed to achieve the latency and reliability constraints. While the existence of a central entity which controls the policies of all stations is hard to deploy in our scenario, we prefer using online learning, and more specifically Multi-Armed Bandit (MAB), to allow transmitters to adapt their optimal policies in a distributed manner.

### III. Multi-Armed Bandit formulation

In MAB context, policies are called arms. We denote the set of all possible arms by \( A = \{ 0, 1, ..., T \} \), where arm \( a \) is associated to policy \( \tau \). Each arm has a reward distribution that is unknown to the learner whose aim is to estimate its expected value over a horizon of rounds. Every packet represents a round, and at the beginning of each round, the station chooses an arm and receives one observation of its reward distribution.

#### A. MAB algorithms

The choice of arms is divided into two phases, exploration and exploitation. In the exploration phase, arms are chosen in order to collect more information about their reward distributions, while in the exploitation one, the station chooses the empirically best arm explored so far in order to maximize its cumulative reward.

These two phases can be totally separated like in the *explore-first* algorithm which explores all arms uniformly (in a round robin fashion) for a given number of rounds, denoted by \( N_{\text{explore}} \), then chooses the empirically best arm for the rest of the rounds, or mixed as for the \( \epsilon \)-greedy algorithm which, in every round, explores the arms randomly with probability \( \epsilon \) and exploits the best identified arm so far with probability \( 1 - \epsilon \). The \( \epsilon \)-greedy algorithm can be preceded by a uniform exploration phase for a number of rounds denoted by \( N_{\text{greedy}} \).

The aforementioned algorithms represent the basic forms of MAB, where the exploration is considered deterministic. More advanced algorithms use adaptive exploration methods based on the uncertainty of estimation (the number of times each arm has been picked), as in so-called *Upper Confidence Bound* (UCB) and its variants [5].

As explained previously, the stringent constraint on reliability, on the order of \( 10^{-5} \), which must be included in the reward calculation, indicates that a packet loss event is to occur once every \( 10^5 \) rounds on average, and as MAB updates the estimates for one arm per round only, then the learning of all reward distributions requires a huge number of rounds, rendering the learning process pointless. For this reason, we choose to use the basic MAB algorithm as a benchmark, since UCB algorithm does not have any added value in this context (the uncertainty cannot be defined properly as long as the rare event does not happen). We
show in Section IV our proposal to remedy to this slow learning issue.

B. MAB performance

The most widely used measure to evaluate the performance of a MAB algorithm is the regret, defined as the difference between the effective cumulative reward obtained from the real realization of the algorithm and the cumulative reward obtained from constantly choosing the best arm, over a given horizon of rounds.

The regret of different algorithms is very dependent on the application and context. For the explore-first algorithm, it is hard to know when to stop the exploring phase, because committing to a non-optimal arm due to lack of exploration increases the regret. Likewise when exploring because committing to a non-optimal arm due to lack of algorithm, it is hard to know when to stop the exploring phase, where the widely used value is $\epsilon = 0.2$.

For this reason, we choose to measure the performance by the convergence time, which we define as the number of rounds needed for the algorithm to converge and stabilize to the optimal arm. For explore-first, this corresponds to the number of rounds needed for the exploration phase to identify the optimal arm.

C. Reward and penalty

The reward expresses the positive part of the feedback we receive after taking an action, and the penalty expresses the negative one. The highest rewarding action is at the same time the least penalizing one, thus, searching for the action which maximizes the reward is equivalent to searching for the one which minimizes the penalty. In our context, it is more relevant to express the feedback in terms of penalty, since measures like energy consumption and context, it is more relevant to express the feedback in terms of penalty. In our case multiple arms with identical penalties are identified, we choose the arm with highest policy to avoid unnecessary energy consumption.

The loss rate of arm $a$ over a horizon of $n$ rounds is given by:

$$\bar{L}_n(a) = \frac{\sum_{i=1}^{n} L_i(a) 1_{a_i = a}}{\sum_{i=1}^{n} 1_{a_i = a}}$$

where $L_i(a) = 1$ if the packet is lost in round $i$ using arm $a$ and is equal to 0 otherwise.

We define the expected penalty of arm $a$ over a horizon of $n$ rounds as:

$$\Gamma_n(a) = \bar{C}_n(a) \times |\bar{L}_n(a) - \Lambda|$$

noting that $\Gamma_n(a) \in [0,1]$.

The description of the problem suggests that $\bar{C}_n(a)$ is a decreasing functions of $a$ while $\bar{L}_n(a)$ is an increasing one. If $\Lambda$ is included in the range of $\bar{C}_n(a)$ for $a \in A$, then the difference $|\bar{L}_n(a) - \Lambda|$ is strictly convex (has at least one minimum); if not, then it is a monotonically increasing or decreasing function depending on the range of values of $\bar{L}_n(a)$. We are interested in the case of $\Lambda \in [\min(\bar{L}_n(a)), \max(\bar{L}_n(a))]$. We denote the optimal arm which has the minimum penalty with $a^*$.

D. Implementation

Using the derived penalty $\Gamma_n(a)$ in any MAB algorithm (Explore-First, $\epsilon$-greedy, UCB, etc) leads to the best arm identification after a finite number of rounds. The following algorithms detail the steps of explore-first and $\epsilon$-greedy. In the case multiple arms with identical penalties are identified, we choose the arm with highest policy $\tau$ so as to avoid unnecessary energy consumption.

Algorithm 1: Explore-First

Result: $a^*$

for $i = 1, ..., N_{\text{explore}}$ do

| Choose $a_i \in A$ following Round Robin; |
| Update $\Gamma_i(a_i)$; |
end

$a^* \leftarrow \arg \max_{a_i} \left[ \Gamma_{N_{\text{explore}}}(a_i) \right]$;

for $i = N_{\text{explore}} + 1, ..., n$ do

| $a_i \leftarrow a^*$; |
end

IV. MODEL- AIDED MULTI-ARMED BANDIT

The slow behaviour of MAB algorithms is due to the limited interpretation of the feedback, i.e., the feedback in every round is exclusive to the selected arm. This is sometimes preferable because of the low complexity, but in certain cases where the algorithm requires millions and millions of rounds to give an accurate solution, then an algorithm with higher complexity and faster convergence to the solution becomes more desirable.

We propose a model for the measure which is causing slow learning convergence in our case which is the loss probability. This model is based on observations of the
environment which can be assessed by the learner in every round, independently from the chosen arm. We then incorporate the derived model into the MAB algorithms.

A. Loss rate model

Referring to Figure 1, we can compute the loss probability of a packet within a delay budget $T$, given the number of opportunities $n_1(\tau)$ in $[0, \tau]$ and $n_2(\tau)$ in $[\tau, T]$, by the following formula:

$$P_L = q_1^{n_1(\tau)} \times q_2^{n_2(\tau)}$$

where $q_1$ and $q_2$ are the probabilities of collision using $P_1$ and $P_2$ power transmission levels, respectively.

Since $n_1(\tau)$ and $n_2(\tau)$ are independent rv's, the loss rate of arm $a$ under policy $T$ $P^\text{model}_a$ corresponds to the expected value of $P_L$ over the distributions of $n_1(\tau)$ and $n_2(\tau)$.

$$P^\text{model}_a = E_{\{n_1(\tau), n_2(\tau)\}}[q_1^{n_1(\tau)}q_2^{n_2(\tau)}]$$

leading to the PDF:

$$f_{n_1(\tau)}(m) = \mathbb{P}(n_1(\tau) = m) = F_m(\tau) - F_{m+1}(\tau)$$

Similarly, the PDF of $n_2(\tau)$ is given by:

$$f_{n_2(\tau)}(m) = \mathbb{P}(n_2(\tau) = m) = F_m(T - \tau) - F_{m+1}(T - \tau)$$

The loss rate $P^\text{model}_a$ of arm $a$ under policy $\tau$ can be calculated using equation (1), as follows:

$$P^\text{model}_a = \sum_{i=1}^{\infty} f_{n_1(\tau)}(i) q_1^i \times \sum_{j=1}^{\infty} f_{n_2(\tau)}(j) q_2^j$$

B. Environment parameters

In practice, the environment parameters ($N_u$, $N_e$, $N_u^H$, $W_e$, $x_a$ and $x_e$) cannot be estimated, but their impact on the channel load can be observed through $q_1$, $q_2$ and $\theta$.

The probability of transmitting a URLLC packet per time slot per station is $p_u \approx 1 - p_0$ because $p_0$ is typically large (e.g., 0.999) and the LBT procedure has a limited effect on $p_u$. The proportion of URLLC stations transmitting with high power at a given time is denoted by $N_u^H$. The transmission probability of eMBB packet (per time slot per station) is given by $p_e = \frac{2}{W_e + 1}$ [10] (formula for LBT cat3 and saturated sources derived in [10]). From the viewpoint of a URLLC station, we have:

$$q_1 = 1 - (1 - p_u)^{N_u-1}(1 - p_e)^{N_e}$$
\[ q_2 = 1 - (1 - p_u)^{N_u^H} \]

We observe that \( q_2 \) depends on the transmission power of other URLLC stations while \( q_1 \) depends on the network setting. Note that \( q_2 \leq q_1 \), equality holds when \( N_e = 0 \) and all URLLC stations transmit using \( P_2 \) all the time. \( q_1 \) and \( q_2 \) are considered constant and independent of time.

The learner can acquire estimations of the parameters from the reaction of the environment to its transmissions. For instance, we can assess the following estimates:

\[
\hat{q}_1 = \frac{N_{\text{collision}}}{N_{\text{transmit}}}; \quad \hat{q}_2 = \frac{N^H_{\text{collision}}}{N^H_{\text{transmit}}} \;
\]

\[
f_\theta(l) = P(\theta = l) = \frac{N(\theta = l)}{\sum_l N(\theta = l)}
\]

where \( \hat{q}_1, \hat{q}_2 \) and \( f_\theta \) denote the estimates of \( q_1, q_2 \) and \( f_\theta \), respectively, \( N_{\text{collision}} \) and \( N^H_{\text{collision}} \) denote the number of collisions when transmitting with power levels \( P_1 \) and \( P_2 \), respectively, \( N_{\text{transmit}} \) and \( N^H_{\text{transmit}} \) denote the number of transmissions with power levels \( P_1 \) and \( P_2 \), respectively, and \( N(\theta = l) \) denotes the number of sensed inter-transmission times \( \theta \) equal to \( l \), where \( \theta \) is calculated from the packet generation (or collision) until the reception of ACK/NACK. \( f_\theta \) expresses also the histogram of the inter-transmission delays.

C. Notes on implementation

Our aim is to use the above derived model in the MAB algorithms shown in subsection III-D. Assessing the environment parameters \( (q_1, q_2 \) and \( f_\theta \) does not introduce complexity to the algorithms, as they only require small space in memory and simple addition and division operations. However, the added complexity comes mainly from the calculation of the convolution in the model in equation (2), when deriving the PDFs of \( n_1(\tau) \) and \( n_2(\tau) \).

For the explore-first algorithm, the calculation of the model has to be done only once, when the exploration phase terminates. Therefore, using our model into Algorithm 1 adds tolerable complexity and enhances the speed of learning significantly, as will be shown next in the numerical section. In this case, \( N_{\text{explore}} \) is adapted to the needs of estimating the system parameters.

In the \( \epsilon \)-greedy algorithm, there are several ways to use our model in the algorithm. The method we suggest is that in every round, the loss rate model is updated for the selected arm, and hence only one convolution calculation is needed per round. This introduces more complexity compared to the one added to the explore-first algorithm, but it is inevitable in \( \epsilon \)-greedy as the algorithm has to identify the best arm and exploit it during the run.

Another source of complexity of the algorithms is the size of the space of arms \( A \). In our case, we notice that adjacent arms (with close values of \( \tau \)) lead to similar performances because they have similar distributions of \( n_1(\tau) \) and \( n_2(\tau) \). On account of this, we consider the arms with policies spaced by \( x_u + 1 \) time slots (\( x_u + 1 \) is the minimum value of \( \theta \)). This also accelerates finding \( \alpha^* \).

V. Numerical application

We consider parameter values similar to the ones defined in latest IEEE 802.11 standards, notably a time slot duration of 9\( \mu \)s and a bit rate of 100 Mbps. Large eMBB packets are chosen as the maximum length of a Wi-Fi packet: 2312\( \text{Bytes} \), while it is sufficient for URLLC to consider packets of 32\( \text{Bytes} \) length. The durations \( x_u \) and \( x_e \) are calculated considering all headers, ACK/NACK reception and guard intervals. Table I shows the numerical values used in the numerical applications. Recall that temporal quantities are expressed in time slot unit.

<table>
<thead>
<tr>
<th>( N_u )</th>
<th>( x_u )</th>
<th>( x_e )</th>
<th>( W_e )</th>
<th>( C_1 )</th>
<th>( \epsilon_0 )</th>
<th>( \alpha_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>6</td>
<td>8</td>
<td>C_1</td>
<td>0.01</td>
<td>2</td>
<td>0.05</td>
</tr>
<tr>
<td>2</td>
<td>13</td>
<td>32</td>
<td>( \epsilon_0 )</td>
<td>( 1 - 10^{-\alpha_0} )</td>
<td>( \alpha_0 )</td>
<td>0.999</td>
</tr>
</tbody>
</table>

Note that in a given unlicensed band, we have several channels which can be chosen using Dynamic Frequency Selection (DFS) or combined using carrier aggregation.

We limit our evaluation to one channel, while the real capacity of the network can be evaluated by multiplying the capacity of one channel by the number of available channels.

The simulation is performed from the viewpoint of one URLLC station which transmits packets following LBT. Every packet is tagged with CW, stage count and timer. The probability of sensing the medium busy in a given time slot when using power level \( P_1 \) has two folds: 1) active URLLC stations which are represented by the outcome of a random Poisson-distributed vector of length \( N_u - 1 \) and parameter \( 1 - p_0 \); 2) the saturated eMBB stations which are tagged with CW and transmit following LBT when their CW hits zero. When transmitting with power level \( P_2 \), we consider \( N_u^H = \lfloor (N_u - 1) / 3 \rfloor \), and the probability of having another transmission with \( P_2 \) is represented with the outcome of a random Poisson-distributed vector of length \( N_u^H - 1 \) and parameter \( 1 - p_0 \).

We simulate the four studied algorithms (classical explore-first and \( \epsilon \)-greedy and our model-aided versions) using the numerical values in Table I. We evaluate the algorithms every 100 rounds and record the optimal arm for the algorithm at that round. For \( \epsilon \)-greedy algorithms, we consider \( \epsilon = 0.2 \), and \( N_{\text{greedy}} \) equal to three times the number of arms. Recall that we pick the arms spaced by \( x_u + 1 \) time slots, hence the number of arms is reduced from: \( T + 1 = 121 \), to: \( T + 1)/(x_u + 1) = 18 \) arms.

A. Convergence time

We first compare the convergence time to the optimal arm defined as the number of rounds needed for the algorithm to identify the best arm. We show in Figures
We observe from Figure 2 that classical explore-first algorithm does not converge to an optimal solution for a horizon of rounds of $18 \times 10^5$, which shows its incapability to solve our problem efficiently. Classical $\epsilon$-greedy algorithm converges to an optimal arm after almost 60000 rounds, which is significantly better than the previous one.

In Figure 3, we compare the model-aided versions of the previous algorithms. Both model-aided algorithms converge after less than a thousand rounds, and this is explained by the time needed to estimate $q_1$, $q_2$ and $f_\theta$ correctly. The model may oscillate around the optimal arm for a period of time as shown in the model-aided explore-first curve, due to variations in the simulation, where collisions may occur more or less often depending on the rvs used to generate the collisions in the simulation.

Regarding the difference of the best identified arm between classical $\epsilon$-greedy and the model-aided algorithms, where the first converges to $\tau^* = 105$ and the latter to $\tau^* = 98$, this is due to lack of exploration of $\epsilon$-greedy, where for this particular realization of the algorithm, the learner found the best arm and started exploiting it which reduces the exploration of the other arms for the rest of the horizon. We note that the two found optimal arms are very close.

B. Expected energy consumption

We illustrate in Figure 4 the expected energy consumption of each of the algorithms. Figure 4 confirms the intuition that the expected energy consumption is a decreasing function of $\tau$. It is also shown that the difference of energy consumption between arms decreases with $\tau$, because arms with larger $\tau$ do not get to transmit using $P_2$ in the simulation.

C. Expected loss rate

We compare in Figure 5 the estimated expected loss rate by the algorithms. Classical algorithms acquire their information from the number of lost packets in the given horizon, which is not enough to get accurate estimates for all arms. The $\epsilon$-greedy algorithm focuses on one arm leading to poor estimation for other arms, and the explore-first one obtains similar values for multiple arms since it explores them uniformly and it is possible that the loss event occurs only once per arm leading to poor estimation as well. Our model-aided algorithms estimate the loss rate accurately for all the arms since they rely on the model and the estimates of the environment parameters.

D. Expected penalty

We finally evaluate in Figure 6 the calculated expected penalty for the algorithms. As shown in the curves in Figure 6, the algorithms choose the best arm over the horizon of rounds. The
oscillation in explore-first algorithm shown in Figure 2 is followed by similar oscillations in the penalty estimates of the algorithm. Figure 6 also illustrates the convexity of the penalty due to the convexity of the difference of expected loss rate of the constraint: $|\bar{L}_n(a) - \Lambda|$, leading to a minimum value of the penalty function at the best arm.

VI. Conclusion

We studied in this paper the transport of URLLC using unlicensed spectrum, in the presence of eMBB traffic. To guarantee URLLC’s stringent requirements of latency and reliability, we proposed a new transmission scheme where URLLC stations transmitting in the uplink can increase their transmission power when the delay of their packets approaches the delay constraint, increasing by that its chance of being decoded even in the case of collision with other lower-power packets.

In order to minimize the energy consumption, we proposed a transmission policy based on a delay threshold, after which the high-power transmission starts. We made use of on-line learning Multi-Armed Bandit in order to find the optimal transmission policy which achieves minimum energy consumption while guaranteeing reliability as close as possible to the imposed constraint.

We addressed the issue of slow learning originating from the stringent reliability constraint and proposed an analytical model which helps compute this reliability. We then incorporated this model into the original MAB algorithms and demonstrated the significant enhancement of the learning speed albeit some added complexity.

Acknowledgement This work was partially funded by ANR-18-CE25-0012 project Maestro5G.

References

Statistical Learning of Markov Chains of Programs

Emilio Incerto
IMT School for Advanced Studies
Piazza San Francesco 19
Lucca, Italy
emilio.incerto@imtlucca.it

Annalisa Napolitano
IMT School for Advanced Studies
Piazza San Francesco 19
Lucca, Italy
annalisa.napolitano@imtlucca.it

Mirco Tribastone
IMT School for Advanced Studies
Piazza San Francesco 19
Lucca, Italy
mirco.tribastone@imtlucca.it

Abstract—Markov chains are a useful model for the quantitative analysis of extra-functional properties of software systems such as performance, reliability, and energy consumption. However, building Markov models of software systems remains a difficult task. Here we present a statistical method that learns a Markov chain directly from a program, by means of execution runs with inputs sampled by given probability distributions. Our technique is based on learning algorithms for so-called variable length Markov chains, which allow us to capture data dependency throughout execution paths by encoding part of the program history into each state of the chain. Our domain-specific adaptation exploits structural information about the program through its control-flow graph. Using a prototype implementation, we show that this approach represents a significant improvement over state-of-the-art general-purpose learning algorithms, providing accurate models in a number of benchmark programs.

Index Terms—statistical learning, quantitative models, extra-functional properties, Markov chains

I. INTRODUCTION

Markov chains (MCs) are a fundamental model to describe the dynamics of computer and communication systems as stochastic processes [5], [45], to enable reasoning about various extra-functional, quantitative properties such as reliability [28], [41], performance [4], and energy consumption [46].

Quantitative properties could also be analyzed through profilers; however, profiling lacks generalizing and predictive power (see also [51]), thus hindering program understanding. On the other hand, Markov modeling hinges on considerable craftsmanship in both the problem domain and in the required analytical techniques, hindering their adoption in practice [50].

A possible solution to mitigate the difficulties in obtaining an accurate MCs from a software system might be to derive it automatically. There has been much research into extending higher-level descriptions such as UML diagrams with quantitative annotations (using, for example, appropriate profiles such as MARTE [39]) from which both software artifacts and associated stochastic models are generated [2], [25]. However, this introduces a hard problem of synchronization between the model and the software, since automatically generated code stubs are typically subjected to further manual intervention by the developer [16]. Thus, such model-driven approaches can be particularly difficult to use in general, especially in the context of fast-paced software processes with continuous integration.

Here we aim to bridge this gap by learning a Markov model from a program. The starting point is to interpret a program probabilistically by following a well-known argument (see, e.g., [11], [44]) about resolving the nondeterminism of the input by assigning a representative probability law, or modeling intrinsic probabilistic behavior due to the use of primitives that sample values drawn at random from given distributions. However, although such a probabilistic semantics of programs is not new [17], [30], it has not yet been used to synthesize probabilistic models for a program as a whole, i.e., without focusing on the quantification of the probability to satisfy specific path formulas [11], [14].

In this paper, we start from the idea that the stochastic behavior of a program can be operationally given as an MC where each state tracks the program location as well as the current values of the program variables (e.g., [44]). Unfortunately, this detailed state representation may make probabilistic analysis infeasible, since the state space grows fast with the size of the domain of the variables.

To cope with this problem, we propose to build a Markov model which tracks program locations while abstracting from variable values. As an example, let us consider the toy snippet on the left, where the branches of the conditional statements are omitted for simplicity since we assume that they do not alter the input variable $x$; $l_0, \ldots, l_6$ label the program locations (PLs); $x$ is assumed to be equal to 0 or 1 with equal probability 0.5. Intuitively, one could envisage an MC that only tracks program locations as shown in Fig. 1a, where each conditional statement is represented by a state (here, $l_0$ and $l_3$) with two outgoing transitions labelled with the probability of the boolean expression. We refer to it as the first-order (i.e., memoryless) MC, since the future behavior only depends on the current state/PL.

The int $x U\{0,1\}$ $l_0$ if ($x == 0$) $l_1$ // if stmt $l_2$ else // else stmt $l_3$ end $l_4$ if ($x == 0$) $l_5$ // if stmt $l_6$ else // else stmt $l_6$ end

It can be easily seen that easily foregoing the information about variable values in the state representation may require knowledge about the history of visited program locations to obtain a precise model. Indeed, the MC in Fig. 1a allows the path $l_0 \rightarrow l_1 \rightarrow l_3 \rightarrow l_5 \rightarrow l_6$ with probability 0.25, but this cannot correspond to any feasible path in the program because the value of variable $x$ tested in line $l_3$ is not re-sampled (i.e., the truth value of the two branches cannot be discordant).

Higher-order MCs can account for data dependency by next-state probabilities that depend not only on the current PL...
In particular, we propose the use of variable length Markov chains (VLMCs) [8], with a tunable learning algorithm to attain a user-desired trade-off between precision and the amount of history to incorporate in the retained contexts.

Our solution, i.e., ProgramToVLMC, is a domain-specific adaptation of a popular algorithm for the statistical learning of VLMCs [31], [33], based on training traces that represent execution paths, and informed by the control-flow graph (CFG) of a program. On a number of benchmarks using a prototype implementation, we show that ProgramToVLMC outperforms general-purpose state-of-the-art implementations that would otherwise suffer severe memory or time cost, thus offering a viable solution to derive probabilistic models of programs.

Fig. 1: (a) First-order (i.e., memoryless) and (b) higher-order Markov chain for the example in Section 1.

Paper organization: The remainder of this paper is organized as follows. Section II presents related work; Section III provides the foundational concepts about VLMC; Section IV describes the internals of the learning algorithm; in Section V, the numerical evaluation is presented; Section VI discusses relevant open issues and the possible lines of future research.

II. RELATED WORK

a) Code-driven generation of quantitative models: In software performance engineering, the use of MCs is widely accepted (e.g., [13]). However, most of the literature has focused on model-driven approaches [2], while code-driven generation of models has been less explored. Tarvo and Reiss develop a technique for the extraction of queuing network models from a class of multi-threaded programs [47]. They focus on aspects about concurrency and synchronization, but not on detailed models of single tasks, which is instead the objective of the present paper. In [40], an approach for mining the software resource-aware behavior from logs is presented. Although such models could be used to improve the developers’ understanding of the system they do not produce an exact analytical representation of the software dynamics. In BEAR [19], MC that model users behaviours are constructed from logs. They do not consider memory effects and they are not concerned with performance. Other works use n-Grams Markov models to improve code autocompletion capabilities or bug finding, such as [22], [48], but they do not focus on performance. Brüning et al. [6] presents an approach that mines a performance model from running systems automatically. However, unlike our approach, it creates performance models by focusing on hot functions (i.e., the most frequently executed ones) only.

b) Code-driven generation of performance profiles: Larus proposes an effective profiling approach for quantifying the impact of the hot portions of program by combining the execution frequencies and measured path cost [27]. Starting from the relative frequency of each path, a probabilistic interpretation of the model could be provided. However, this would be of maximal order since all the paths of the program are explicitly considered. On the contrary, our method is able to produce models that present a high level of accuracy while incorporating only the statistically relevant information.

PerfPlotter analyzes code for performance evaluation [11]. Similarly to our method, this is done by resolving the nondeterminism of a program input probabilistically by assuming a given usage profile through a probability distribution on the input variables. PerfPlotter is based on probabilistic symbolic execution [17], and extensions thereof for the reliability analysis of software [14]. It introduces suitable heuristics for the exploration of a program’s paths in order to discover exactly the best- and worst-case execution times, together with an approximated version of the whole distribution.

Despite the similar goal, there are significant differences with PerfPlotter. First, PerfPlotter does not produce a model, but an empirical distribution of a performance metric. From
this it is not possible to directly associate performance information to program components. Second, our method explores paths using random sampling, so it may miss best- and worst-case scenarios. However, since the analysis of a program is not tailored to finding these extreme behaviors, the whole distribution can be more accurately captured by our approach in general.

c) Code-driven generation of VLMCs: Approaches that extract VLMCs from code are [35], [36]. There, the focus is on intrusion detection, with a granularity of the model chosen to keep track of sequences of system calls. These VLMCs are built using the general-purpose algorithm in [42], against which we compare in Section V.

d) Probabilistic grammars inference: In [9] the ALEGIA algorithm has been proposed for the identification of probabilistic finite automaton (PFA) from sample execution traces. In [42], an equivalence relation and a transformation algorithm between PFA and VLMC have been formally provided showing that a PFA can be seen as a Markovianization of a generating VLMC (i.e., see Section III). However, in [33], it is shown that a PFA could be, in the worst case, quadratically bigger of the respective VLMC, making the usage of VLMC more convenient in practice.

III. PRELIMINARIES

In this section we briefly recall the basic facts about VLMCs [8] instrumental for the development of the paper.

A process shows a variable length memory when the memory length depends on the specific sequence of states that has been observed. The information about which part of the process history is relevant to determine the conditional probability distribution for the next state is encoded in the context function, hereby denoted by \( C \). For a sequence of states \( x_n x_{n-1} \cdots x_0 \), the context \( C(x_n x_{n-1} \cdots x_0) \) is the longest prefix \( x_n \cdots x_{n-k+1} \) such that \( \Pr(x_i \mid x_n x_{n-1} \cdots x_0) = \Pr(x_i \mid x_n x_{n-k+1}) \) for all states \( x_i \).

To better illustrate this, let us consider a stochastic process with state space \( L = \{ l_0, l_1 \} \), and assume that the conditional probability distribution for the next state depends only on the current state \( l_j \) if \( j = 0 \), but on the previous history if \( j = 1 \), according to the following context function:

\[
C(x_n \cdots x_0) =
\begin{cases}
  l_0 & \text{if } x_n = l_0, \\
  l_1 l_1 & \text{if } x_n = l_1 \land x_{n-1} = l_1, \\
  l_1 l_0 l_0 & \text{if } x_n = l_1 \land x_{n-1} = l_0 \land x_{n-2} = l_0, \\
  l_1 l_0 l_1 & \text{if } x_n = l_1 \land x_{n-1} = l_0 \land x_{n-2} = l_1,
\end{cases}
\]

The set of all contexts can be conveniently represented as a tree, the context tree, where each context is encoded as a path ordered from the most to the least recently observed state when traversed from the root to the leaves [42].

Figure 2 shows the context tree of the simple example just described. In this representation, first-level nodes represent 1-length knowledge; thus, a context tree with first-level nodes only is a first order MC. On the contrary, a complete \( k \)-height context tree implies a \( k \)-order full MC.

A VLMC is obtained by turning the context tree into a probabilistic suffix tree (PST) [42], which associates each node with a probability distribution: this represents the next symbol distribution, that is, the conditional distribution for the next state given its context. Suppose, for instance, that we wish to find \( \Pr(l_i \mid l_1 l_0) \) in the tree of Figure 2: this is encoded in the probability distribution of the second-level node \( l_0 \) that is a direct child of the first-level node \( l_1 \).

IV. LEARNING ALGORITHM

In this section we show how to learn a VMLC from samples of execution traces of a program by using ProgramToVLMC, which is our customized version of the learning algorithm originally proposed in [31], [33].

A. Input Programs

As input we consider any bounded loop-free program [21] written in an imperative programming language. That is, we assume that the program has undergone standard transformation techniques such as loop unwinding and function inlining. Specifically, we consider the program’s full control-flow graph (CFG) \( G = (L, E) \), where \( L \) is the set of PLs and \( E \) is the set of transitions between PLs. Listing 3 reports a simple program that will be used as an illustrative example throughout this section, with its CFG.

Capturing the behavior of this program with a stochastic model is not straightforward because of two main difficulties: (i) the behavior is not memoryless, e.g., \( b[2] \) depends on the evaluation of \( b[0] \), which is statements behind the current

\begin{verbatim}
public void syntheticP(){
  int[] b=new int[3];
  Random r=new Random();
  l0 Random r=new Random();
  l1 int[] b=new int[3];
  l2 if(r.nextInt(100)<20)
    l3 b[0]=1;
    else
      l4 b[0]=0;
  l5 if(r.nextInt(100)<80)
    l6 b[1]=1;
    else
      l7 b[1]=0;
  l8 if((b[0]==0 && r.nextInt(100)<40)
      ||(b[0]==1 && r.nextInt(100)<90))
    l9 b[2]=1;
    else
      l10 b[2]=0;
}
\end{verbatim}

Fig. 2: Context tree enriched with examples of next-symbol probability distributions

Fig. 3: Simple synthetic program with its control flow
Algorithm 1: Fit a VLMC for program P

ProgramToVLMC (Traces, G, n_{min}, \alpha)

inputs: The set of traces; the control-flow of P; the minimal occurrences of a context; the pruning factor
output: A VLMC for P

Global sa \leftarrow buildSuffixArray(Traces)
PST \leftarrow \emptyset

foreach PL \ s_i \in G \ do
    LocationPST \leftarrow growPST(l_i)
    prune(LocationPST)
    PST.addChild(LocationPST)
end

return PST

Algorithm 2: Recursively create the location PST of PL \ s_c \ by backwardly enlarging the considered context

growPST (s_c s_{c-1} \ldots s_0)

inputs: The input context \ s_c s_{c-1} \ldots s_0
output: The candidate location PST of the PL \ s_c

pstNode \leftarrow \text{new Node} (s_0)
pstNode \leftarrow Pr(s_1 \mid s_c s_{c-1} \ldots s_0), \ \forall s_i \mid s_c \rightarrow s_i \in E

foreach PL \ s_j \mid s_j \rightarrow s_0 \in E \ do
    if (isObserved(s_c s_{c-1} \ldots s_0 \cup s_j, \ n_{min})) then
        pstNode.addChild(growPST(s_c s_{c-1} \ldots s_0 \cup s_j))
end
end

return pstNode

one; (ii) the memory has variable length, i.e., b[0] and b[1] are memoryless conditions while b[2] is not.

B. VLMC of a Program

Consider a program with n PLs. The output of the learning algorithm is a VLMC defined through the PST given by the tuple \ (C, P) where \ C : \bigcup_{i=1}^{n} L^i \rightarrow \bigcup_{i=1}^{n} L^i \ is the context function and \ P : Im(C) \rightarrow \mathbb{R}^L \ is the next-symbol probability distribution, which associates each context with a conditional probability distribution over the PLs.

C. Learning by ProgramToVLMC

a) Overview: Algorithm 1 describes the construction of a VLMC for a program with CFG \ G, exercised with a training set of traces. Each trace is a sequence of nodes of \ G, recorded by executing the program under statistically independent runs. The algorithm is configured by two hyper-parameters that determine the order and accuracy of the learned model: the minimal number of occurrences \ n_{min} \ that each context must be found in the traces in order to be used for the generation of the PST, and a pruning factor \ \alpha. \ These be detailed later.

First, ProgramToVLMC builds the suffix array (cf. [29]) from the traces (line 2); this allows us to efficiently count the number of occurrences of any context, and will be used in the subsequent stages to efficiently compute all the next-symbol probability distributions. For each location \ l_i \ in the CFG, the algorithm builds a candidate sub-PST rooted at \ l_i \ which, from now on, we call the location PST (line 5). This contains all the prefixes of the CFG paths ending in \ l_i, \ since they represent all the possible memory dependencies for that PL. In addition, the function growPST computes all the next-symbol probability distributions for the location PST of \ l_i. Finally, the pruning phase will retain only statistically significant nodes for each location PST (line 6), then add it to the final PST (line 7).

At this stage, building the VLMC by exploiting the structural information of \ G \ has one practical advantage: it allows pruning each location PST earlier than general-purpose algorithms, thus reducing the peak memory requirement. See Section V for a numerical evaluation of this aspect.

b) Growing the PST: In the following we will denote a generic context with \ s_c s_{c-1} \ldots s_0 \ where \ s_c \ and \ s_0 \ are its most and least recent PLs respectively. Algorithm 2 reports the function growPST. Its goal is twofold: (i) to create the candidate location PST for \ s_c, \ by recursively visiting all of its ancestors, i.e., the PLs from which there exists a path to \ s_c; \ and (ii) to compute the corresponding next-symbol probability distributions. To better illustrate this, we detail the base case and the recursive step separately. As an example, we apply Algorithm 2 for generating the location PST of \ l_5 \ in Figure 3.

Base step. This represents the first call to the function and is performed by Algorithm 1 at line 5; in our example hence we have \ s_c s_{c-1} \ldots s_0 \equiv l_5 \ as input context. At line 2, the node corresponding to \ s_0 \ is created (i.e., since \ s_0 \equiv l_5 \ this represents the root of the location PST of \ l_5). Next, in line 3, the next-symbol distribution is calculated considering only \ l_5 \ as context, that is, we compute the probabilities \ Pr(s_1 \mid l_5) \ for all locations. Once calculated, this is stored in the newly created node of the location PST.

Recursive step. In line 4, the recursion starts. The growPST function is invoked on a context enlarged with the PL label of the ancestors; in our example, these are \ l_3 \ and \ l_4. Moreover, at line 5 we limit the exploration to those ancestors whose context has occurred at least \ n_{min} \ times in the traces. Once all these recursions are closed, the location PST related to \ l_5 \ is returned (line 9) to Algorithm 1 which adds it as a child of the root of the final PST.

c) Learning the next-symbol distribution: The next-symbol probability distribution (see line 3 of Algorithm 2) is computed using the classical frequentist approach: for each context \ s_c s_{c-1} \ldots s_0, \ the probability of visiting the location \ s_i \ with that context, \ Pr(s_i \mid s_c s_{c-1} \ldots s_0) \ is given by the number of sequences containing \ s_i s_c s_{c-1} \ldots s_0 \ divided by the total number of sequences containing \ s_c s_{c-1} \ldots s_0. \ Even this step can benefit from the structural information about the program by limiting the computation of the next-symbol distribution only on the outgoing labels from \ s_c \ (of size at most 2), instead of the entire state space \ L \ as in the general-purpose learning method of [33].
Fig. 4: Learned PST for the Program 3. The red crossed nodes identify pruned nodes in the location PST of $l_2$.

d) Pruning the tree: The goal of this phase is to eliminate contexts that do not contribute statistically significant memory [8]. For this, the next-symbol probability distribution of each leaf node of the PST is compared with that of its parent and pruned when their Kullback-Leibler (KL) divergence [26] is less than a user-specified threshold given by

$$d = \text{cutoff} \cdot \alpha,$$

where $\alpha$ is the pruning parameter of Algorithm 1. Small $\alpha$ values (i.e., tending to zero) involve smaller PSTs, less memory occupation, and coarser approximations. Instead, high values of $\alpha$ lead to a more faithful reproduction of the program behavior, but also to a greater memory occupation for the PST.

The pruning process is recursively iterated until there are no unexplored leaves (notice that an internal node could become a leaf by possibly pruning all its children). We do not report the pseudo-code for this phase since it has no substantial modification with respect to the state-of-the-art [33].

e) Example: Figure 4 depicts the PST learned for the Program 3 with CFG of Figure 3. Below each node is reported the next-symbol probability distribution of its corresponding context. The values are truncated to the first decimal digit, for readability. Starting from the leaf $l_0$ the next-symbol probability distribution of the context $l_2 l_1 l_0$ is compared with that of its father context $l_2 l_1$. Since they are equal, the corresponding Kullback-Leibler divergence is zero and $l_0$ is removed from the tree. The same process is recursively iterated by comparing the probability distribution of the context $l_2 l_1 l_0$ with that of the context $l_2 l_1 l_0$, which similarly causes the removal of $l_1$ from this location PST. The red-crossed nodes in Figure 4 identify pruned nodes in the location PST of $l_2$.

f) Computational Complexity: ProgramToVLMC takes $O(n^2(\log(n) + n))$ time, where $r$ and $n$ denote the number of sequences and maximal length of a path of the CGF, respectively (i.e., we have at most $n$ PLs). Although we do not improve the theoretical complexity of the learning process, which is between the state-of-the-art approaches in [8] and [42], thanks to the heuristics defined in this section we are able to extract VLMCs generated by benchmark programs that would not be analyzable through the currently available implementations of the state-of-the-art.

V. Numerical Evaluation

Here we evaluate the accuracy of ProgramToVLMC on benchmarks, we conduct a sensitivity analysis with respect to its hyper-parameters, and compare it against the general-purpose VLMC learning techniques from the literature.

a) Set-up: All experiments were performed on a Linux machine with 48 cores and 60 GB of memory through a prototype Java implementation. The benchmarks are:

- **Double Loop [11]**: It is composed of two loops executed in sequence. For each loop, the number of iterations is random between 0 and 30. We use this as a base case because the program can be formally proven as memoryless. Hence we can evaluate the ability of our approach to produce a VLMC of proper order.
- **Nested Loop [11]**: Two nested loops, each iterating a random number of times between 0 and 20.
- **Bubble Sort, Insertion Sort and Quick Sort [12]**: The input is an array of size 9 with random 32-bit integers.
- **Binary Search [12]**: It implements the binary search that looks for a random generated 32-bit integer $y$ in an array of size 100. The value of $y$ is chosen such that the probability of finding it in the array is 20%.
- **Square root Newton [43]**: It computes the square root of a random double-precision floating point variable between 0 and 100000 using Newton’s method.
- **Euclid Algorithm [43]**: It computes the greatest common divisor of two uniformly distributed integer variables between 0 and 200 using Euclid’s algorithm.
- **Miller-Rabin [37]**: It implements the Miller-Rabin primality test. Given a uniformly distributed integer variable $x$ between 0 and 1000 it checks if $x$ is prime or not. Differently from the other, it is a randomized algorithm.

For every program, we generated learning traces with uniformly distributed input; however, our approach can be used with any other probability distribution. As quantitative measure of interest, for each benchmark we computed the probability distribution of the number of steps, i.e., the number of locations visited until reaching the final state of the program. To obtain statistically robust estimates, we executed independent samples from each program obtaining independent batches, and stopped when the KL divergence of two successive distributions was less than a threshold (i.e., $10^{-5}$). The KL test is meaningful since it is commonly employed to quantify the information loss when one distribution is used in place of another [3]; here, a divergence tending to zero indicates high fidelity of the approximating distribution. We then took the resulting empirical distribution as the “ground-truth” one.

b) Accuracy evaluation: To study the accuracy of Algorithm 1, we ran it with $\alpha = 1$ and $n_{\text{min}} = 1$ and compared the ground-truth distribution of the number of steps, $E$, against the corresponding distribution obtained by simulating the learned VLMC, $V$, with the same number of runs with which $E$ was generated. We denote the computed KL divergence $D_{KL}(E \parallel V)$ by $D_{KL}$.
ground-truth models (see Figure 5) and that the KL divergence is always below the measured ones. The precision is also confirmed by the fact that the learned model is memoryless by construction. This behavior is supposed to be memoryless by construction. This behavior of order 31 for the Double Loop program, which was instead also possible to observe that the memoriless property is not impaired by the pruning factor $\alpha = 1$, which implies that pruning is only allowed when the compared distributions are equal. This is a too stringent requirement, since $\alpha = 1$ makes the comparison highly sensitive to the noise generated by empirically estimating such distributions by repeating independent runs. Indeed, the confirmation that this choice of $\alpha$ led to an over-sized model is provided by the fact that, by using an infinitesimally less stringent value of $\alpha$ (i.e., $\alpha = 0.999999$), we obtained a memoryless VLMC with $D_{KL} = 7.72 \times 10^{-5}$.

c) Sensitivity evaluation: In this section we study the sensitivity of the proposed technique with respect to its hyperparameters $\alpha$ and $n_{\text{min}}$.\footnote{For space reasons, here we report only the most representative benchmarks and choices of hyper-parameters.} Differently from the previous section, to quantify the accuracy of the learned model here we rely on the notion of log-likelihood function $\mathcal{L}$ of a VLMC, as defined in [32]. We focused on this metric since it is commonly employed for measuring the goodness of fit of a set of candidate statistical models to a sample of data [38]. Lower values of $\mathcal{L}$ lead to high accuracy of the analyzed model, which is therefore preferable among the candidate ones. Moreover, this metric allows the comparison between distributions that do not have the same support, which is instead required for KL divergence. This was needed because the choice of hyper-parameters can have a significant impact on the shape and the support of the learned model.

Table I reports the sensitivity analysis with respect to the pruning factor $\alpha$. For each benchmark we used the same runs of the corresponding ground-truth distribution and the same number of runs for the benchmark programs.

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>Double Loop</th>
<th>Nested Loop</th>
<th>Bubble Sort</th>
<th>Insertion Sort</th>
<th>Quick Sort</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>31 77.37 1427</td>
<td>22 407.61 9200</td>
<td>84 717.21 1823996</td>
<td>48 221.91 1116394</td>
<td>81 476.63 1459552</td>
</tr>
<tr>
<td>0.9</td>
<td>1 77.37 62 22 408.55 8481 16 726.48 4736 1 221.91 53 40 492.75 20877</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.8</td>
<td>1 77.37 62 22 408.75 8456 16 726.53 4722 1 221.91 53 40 493.26 20515</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0.7</td>
<td>1 77.37 62 22 408.88 8438 16 726.62 4704 1 221.91 53 40 493.71 20130</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 5 shows that, given a sufficient number of learning samples, $\text{ProgramToVLMC}$ yields highly accurate models that produce distributions of the number of steps overlapping with the measured ones. The precision is also confirmed by the fact that the KL divergence is always below $10^{-4}$. However, it is also possible to observe that $\text{ProgramToVLMC}$ learns a VLMC of order 31 for the Double Loop program, which was instead supposed to be memoryless by construction. This behavior is supposed to be memoryless by construction. This behavior of order 31 for the Double Loop program, which was instead also possible to observe that the memoriless property is not impaired by the pruning factor $\alpha = 1$, which implies that pruning is only allowed when the compared distributions are equal. This is a too stringent requirement, since $\alpha = 1$ makes the comparison highly sensitive to the noise generated by empirically estimating such distributions by repeating independent runs. Indeed, the confirmation that this choice of $\alpha$ led to an over-sized model is provided by the fact that, by using an infinitesimally less stringent value of $\alpha$ (i.e., $\alpha = 0.999999$), we obtained a memoryless VLMC with $D_{KL} = 7.72 \times 10^{-5}$.

Table I reports the sensitivity analysis with respect to the pruning factor $\alpha$, when each model was learned using the same number of runs of the corresponding ground-truth distribution (see Figure 5). Columns $\text{Order}$ and $\#\text{Nodes}$ give the learned VLMC order and the total number of nodes respectively.

As expected, decreasing values of $\alpha$ tend to considerably smaller size of the learned models, but to larger prediction errors, i.e., higher log-likelihoods. There are, however, cases where smaller $\alpha$ does not impair statistical significance, while causing significant compression of the learned model, as in the case of Double Loop and Insertion Sort. For Double Loop, the results confirm that values of $\alpha$ less than 1.0 yield a memoryless model. Furthermore, under uniformly distributed inputs, Insertion Sort is also memoryless. The other examples
show variable memory length. As with all statistical methods, ProgramToVLMC exhibits a behavior that may depend on the choice of the hyper-parameters. We propose possible strategies for addressing this issue in Section VI.

Comparison against state-of-the-art: Here we evaluate the scalability of our method against general-purpose VLMC learning algorithms originally introduced in [8] and [42], by means of the corresponding implementations available in [32] and [15], respectively.

For this analysis we restricted to two benchmarks, Double Loop and Nested Loop, due to technical limitations of the available implementations. These concern the huge memory consumption; for instance, using [32] requires more than 60 GB of RAM for learning the VLMC of Quick Sort on an array of size 3. Double Loop and Nested Loop were the only two benchmarks usable for a meaningful assessment of scalability with respect to increasing program sizes. Nevertheless, these are representative programs because they have starkly different order of the learned VLMCs. The learning was conducted using the same value for the two hyper-parameters (i.e., $\alpha = 1.00, n_{\text{min}} = 1$) and with the same number of statistically independent runs (see Figure 5). By manual inspection we verified that all algorithms provided equivalent VLMCs up to small numerical differences due to the use of different statistical tests in the pruning phase.

Table II reports the numerical results where we compare execution time and peak memory occupation of the algorithms when employed for learning VLMCs from the selected benchmarks of increasing complexity, represented by the column #I, which gives the number of iterations with which each loop has been executed. Our approach consistently outperforms the general-purpose approaches for both metrics. We attribute this to the fact that [32] and [15] do not exploit structural information about the program. Indeed, when compared to [32], ProgramToVLMC shows considerably less memory consumption. This is because [32] does not limit the maximal order of the learned VLMC; for each context in the collected traces, all the preceding symbols are analyzed in order to discover statistical dependencies. By doing so, a potentially huge number of spurious paths may be explored, such as the ones across two consecutive statistically independent runs. In addition, the pruning phase in [32] is invoked only after the complete creation of the PST. Instead, as discussed in Algorithm 1, ProgramToVLMC prunes each location PST, leading to earlier memory release.

We observe that [15] consumes less memory than [32]. This is due to the possibility of specifying a maximal order of the learned VLMC. Still, when compared to our approach, [15] shows considerably longer execution times, although the worst-case computational complexities are similar, as discussed in the previous section. This can be explained by the fact that [15] does not exploit the sparse nature of the VLMC, where each state has at most two transitions, but assumes that the state space is fully connected. As final remark, we stress that albeit [32] works on a fully connected model too, its computation time does not explode with the dimension of the input because of its lower computational complexity.

VI. Conclusion

We have developed a statistical algorithm to derive the VLMC from sample traces of program executions. The model can be used for several types of quantitative analysis, such as estimations of execution times (by computing the probability distribution of the number of steps until termination), hot spots (by finding the program locations with the largest probabilities of being visited), as well as formal queries based, for instance, on probabilistic logics by model checking.

Technically, our main contribution is an adaptation of general-purpose learning algorithms for VLMCs to programs. Despite the promising results, it is possible to identify some limitations which we discuss below.

The required number of samples. We showed that statistical convergence requires a large number of program runs. As a mitigation, it would be possible to parallelize the analysis. Further improvements, which we leave for future work, is to integrate symbolic execution (cf. [24]) with rare event sampling [7], or to use use more sophisticated techniques such as important sampling [20], [23] or concolic testing [34], [49].

Scalability of the learning algorithm. We showed how our method can improve the performance of existing approaches even if it does not improve the theoretical complexity. Although the performance gain is significant both in terms of memory consumption and computing times, extracting VLMCs from complex programs could still be prohibitive since the context are exponential with the number of branches. For taming such an issue, we plan to further improve the computational complexity of the algorithm by following the approach presented in [1] and developing a linear time and space algorithm with respect to the collected traces.

Hyper-parameters tuning. The accuracy of learned VLMC is sensitive to the value of the hyperparameters (i.e., $\alpha$ and $n_{\text{min}}$). Although this issue has been widely discussed in [8], [42], an established methodology has not been proposed. One
solution might be to integrate the creation of the VLMC with static program analysis techniques such as model counting [10] or probabilistic symbolic execution [17]. This would allow the precise calculation of path probabilities to guarantee VLMCs of minimal order. We postpone the combination of our approach with the symbolic ones to future work.

REFERENCES

Age of Information in an Overtake-Free Network of Quasi-Reversible Queues

Ioannis Koukoutsidis*

ORCID: 0000-0002-6761-6163

Abstract—We show how to calculate the Age of Information in an overtake-free network of quasi-reversible queues, with exponential exogenous interarrivals of multiple classes of update packets and exponential service times at all nodes. Results are provided for any number of M/M/1 First-Come-First-Served (FCFS) queues in tandem, and for a network with two classes of update packets, entering through different queues in the network and exiting through the same queue. The main takeaway is that in a network with different classes of update packets, individual classes roughly preserve the ages they would achieve if they were alone in the network, except when shared queues become saturated, in which case the ages increase considerably. The results are extensible for other quasi-reversible queues for which sojourn time distributions are known, such as M/M/c FCFS queues and processor-sharing queues.

Index Terms—Age of Information, queueing network, quasi-reversibility, overtake-free path

I. INTRODUCTION

The Age of Information (AoI, or simply “age”) equals the time from the generation of a piece of information, until that piece is received by its intended destination. In packet networks, we consider that information updates are received in packets, and at each time instant $t$, the destination observes an age $H(t) = t - u(t)$, where $u(t)$ is the generation time of the last packet received. If the process $H(t)$ is ergodic, then its expectation $E[H(t)]$ converges to the average $\overline{H}$ of the ages of all packets, as seen by the destination when the packets are received. The latter metric will be meant when referring to the AoI metric in this paper. Packets may also be of different types or classes, in which case we are interested in the AoI $\overline{H}_c$ separately for each class $c$.

The AoI can serve as a metric in numerous applications, where we are interested in the freshness of the received information. For example, in the Internet of Things, where sensor devices can transmit updates of environmental parameters, or the values of technical parameters such as location and velocity in autonomous vehicles; in the storage of data in computer systems, where we are interested in the freshness of information in the cache memory, or in robotics and control systems, where the fast feedback of information plays a prominent role.

These applications justify the necessity for the AoI as a metric, but perhaps its importance stems more from the fact that it motivates a redesign of computer and networking systems, to also take it into account, as the traditional metrics of throughput and delay were shown from the very start to poorly reflect freshness [1]. Moreover, information freshness trade-offs with energy efficiency, this also motivating the scheduling of transmissions of energy-constrained devices to take AoI into account [2]. An extensive survey of conducted research and the numerous applications of the AoI metric was recently published in [3].

Mathematically, the main tool for theoretically calculating the AoI is queueing theory. Update packets are seen as arriving to a queue, where they wait for processing by a monitor, which reads the related information. So far, results have mainly been derived for single-server queues, with the exception of [4], where the AoI was calculated for M/M/1 queues in tandem under Last-Come-First-Served (LCFS) service discipline with service preemption, and the more general method in [5] (see Sect. II for a description of related work). The method in [5] can theoretically be applied for any network in which the movements of updates are described by a finite-state continuous-time Markov chain, but it is a matrix-based method which can lead to combinatorial explosion for systems with a large number of states, such as networks with infinite queues.

In this paper, we derive results for the AoI in networks, starting from the initial analysis in [1], but presenting it in a new, more simplified manner, and then relying on classical queueing theory results from [6] and [7], which were derived more than three decades ago. The model (in Sect. III) and analysis (in Sect. IV) allow to calculate the AoI for different types or classes of update packets, both at the output of each node, where the packets pass from, and at the exit of the network.

The concept that allows us to extend the calculation of the AoI to a queueing network is that of a quasi-reversible queue. A Markovian queue is called quasi-reversible if the state of the process at time $t$ is independent of the arrival process after time $t$ and is also independent of the departure process prior to time $t$ [8]. The important property of a quasi-reversible queue, which will be of great use here, is that Poisson streams passing through the queue are statistically unchanged, hence producing Poisson output streams with the same rate. This is called the Poisson-in-Poisson-out property.

When quasi-reversible queues are connected into a network,
the network is called a quasi-reversible network. The network itself possesses the Poisson-in-Poisson-out property; however, whether or not internal traffic is Poisson depends on the topology of the network [7]. More specifically, internal traffic is Poisson in overtake-free, or order-preserving networks. These are networks where the order of transmission is preserved and jobs sent later in time cannot arrive at the destination earlier than previously sent jobs. Note that if there is a possibility for update packets to arrive through multiple routes to a destination, then a packet that was sent earlier may arrive later than a subsequent packet, which carries fresher information. The destination could simply discard the packet that arrived later, but this significantly complicates the analysis. Thus, we confine ourselves to overtake-free paths in this paper.

We extend the AoI results in the literature by calculating the AoI for any number of M/M/1 queues in tandem, as well as a single network with two classes of update packets, where the different class update packets do not directly share an output queue, but pass from other nodes first (see Sect. V). The use of FCFS M/M/1 queues might appear simplistic, but apart from its theoretical value, it also has practical significance in networks of queues with multiple classes of update packets, where different class packets should share common resources, without one suffocating the other. For example, it was shown in [9] that for two classes of packets directly sharing an output M/M/1 queue, the FCFS discipline can minimize the sum AoI of the two classes for a large range of load values, outperforming LCFS disciplines both with and without preemption in service. Nevertheless, as discussed in Sect. VI, the work in this paper can also be extended to other service disciplines, such as processor-sharing queues.

II. RELATED WORK

The basic method for calculating the AoI in a single server queue with First-Come-First-Served (FCFS) service discipline was shown in [1], where the authors derived results for M/M/1, M/D/1 and D/M/1 queues. This paper builds upon this basic method for deriving results for networks of queues. Another fundamental paper was [12], where the authors derived results for M/M/1/1 (at most one packet being served, no packets waiting) and M/M/1/2 (at most one packet being served and one packet waiting) queues with FCFS service discipline. This paper also included the definition of the peak AoI, or peak age metric, which reflects the average maximum age seen by the destination, prior to receiving updates.

The intuition that waiting may not be efficient when we are interested in receiving update packets as soon as possible, led to the exploration of LCFS service disciplines. Results for a single server M/M/1 queue were derived in [13], where the authors showed that LCFS, with or without service preemption, always outperforms FCFS for a single class of update packets. Enforcing packet deadlines, or timeouts, after which an update packet waiting in the queue is discarded, was studied in [14] for an M/M/1/2 queue, showing that using a deadline can outperform both the M/M/1/1 and M/M/1/2 without deadline.

Results for multiple sources sharing a single-server M/M/1 queue with FCFS service discipline were first derived in [10]. This also produced insights into how a source can choose its update rate in the presence of interfering traffic, basically showing that the minimum AoI is achieved at a smaller update rate, compared to when only one source sends packets to the queue. For the peak AoI, results for more general multi-class M/G/1 and M/G/1/1 queues were presented in [15].

A breakthrough in the method of calculating the AoI was achieved in [4], where the author used a Stochastic Hybrid System (SHS) to model the system and derived AoI results for M/M/1 queues in tandem, under LCFS service discipline with service preemption. The use of SHS leads to a general method for calculating the AoI for a variety of queues with different service disciplines. In [9], the authors showed the application of the method for calculating the AoI for two-sources sharing an M/M/1 queue under FCFS and LCFS service disciplines. The technique was presented more generally in [5] for any network described by a finite-state continuous-time Markov chain, and led to a system of ordinary linear differential equations that describe the temporal evolution of the moments and moment-generating functions of the age process. For a line network of preemptive memoryless servers and Poisson input, [5] showed that the age at a node is identical in distribution to the sum of the interarrival interval and the service times at the preceding nodes (all exponentially distributed).

III. THE MODEL

We consider a quasi-reversible network consisting of a set of nodes $M = \{1, \ldots, m\}$ and a fictitious node 0 that represents both the source and the sink of the network. The network admits different types, or classes of update packets; these will be the jobs circulating in the network, and may correspond to different applications that share the network for transmission. External update packets of type $c$ arrive at the network from the source node 0 as a Poisson process with rate $\lambda_0^c$. Each arrival joins node $i$ with probability $r_{0,i}^c$. On service completion at node $i$, it is routed to node $j$ with probability $r_{i,j}^c$, or leaves the network with probability $r_{i,0}^c = 1 - \sum_{j \neq 0} r_{i,j}^c$. Generally, an update packet of type $c$ follows a path or itinerary $r^{c} := (r^c_1, \ldots, r^c_n)$, defined by the sequence of nodes $r^{c}_1, \ldots, r^{c}_n$, visited by the packet, from its entry in the network through node $r^c_1$, until its exit from the network via node $r^c_n$. The number of nodes $n$ is called the length of the path. When the path and the packet type are obvious, we will simply denote a path by its start and end nodes, as $r_1 \rightarrow r_0$.

In the queueing network, the total equilibrium rate of type-$c$ update packets through node $i$ (including both external arrivals and internal transitions) can be found by solving the system of traffic equations:

$$\lambda_i^c = \lambda_0^c r_{0,i}^c + \sum_{j \neq i} \lambda_j^c r_{j,i}^c, \quad i, j = 1, \ldots, m.$$  \hspace{1cm} (1)
The rate at which update packets of type \( c \) exit the network from node \( i \) \( (i = 1, \ldots, m) \) is \( \sum_{j} \lambda_{i,j}^{c} \). Additionally, the total rate of update packets (of all types) in node \( i \) is \( \lambda_{i} = \sum_{j} \lambda_{i,j} \).

We assume that all nodes have a FCFS queueing discipline and do not admit batch processing, neither go on vacations. The service times of all packets at each node \( i \) are exponentially distributed with parameter \( \mu_{i} \). Additionally, the network is assumed to contain only overtake-free paths. An overtake-free path is defined as a path:

- where every node is overtake-free (this is already assumed by the FCFS service discipline),
- which is cycle-free, i.e. every node in the path is distinct, and
- which does not contain any forward short-circuits, i.e. packets sharing parts of an itinerary in the forward direction cannot overtake each other by taking a shorter route (a route which intersects two non-contiguous nodes of another packet’s route).

For a more rigorous definition of overtake-free paths, see \([6]\).

It is straightforward to understand the need for overtake-free paths for simplifying the calculation of the AoI: as long as the path is overtake-free, the age of an update packet, as calculated in some point in the forward path, does not depend on update packets behind it. Otherwise, other packets that overtake it will affect its age. Moreover, similarly to what was noted in \([6]\) for the calculation of sojourn times, backward short circuits are immaterial for the calculation of the AoI, since packets along such paths do not change the order in the forward direction.

IV. AOI IN AN OVERTAKE-FREE PATH

Without loss of generality, we will calculate the AoI for a type of update packets at the output of an overtake-free path, by looking at the path as a black box and only being interested in the input and output processes. Consider the path of length \( n \) shown in Fig.1, where update packets of type \( c \) arrive at the first node in the path as a Poisson process at time instants \( \{d_{1}^{c}, d_{2}^{c}, \ldots, d_{n}^{c}\} \), and exit the path at time instants \( \{d_{1}^{c+1}, d_{2}^{c+1}, \ldots\} \). There may also be traffic of other types sharing the path, either wholly or partially, depicted by the diagonal arrows in the figure. The total traffic at each node \( j \) \( (j = 1, \ldots, n) \) is \( \lambda_{j} \), and the service rate is \( \mu_{j} \). Given that \( \lambda_{j} < \mu_{j} \) (so that all queues are stable), we will calculate the AoI at the output of the path. We will further assume that all update packets arrive fresh at the entry of the path, i.e. their initial ages are zero.

![Fig. 1: A view of a path as a “black-box”](image)

For an overtake-free path, it was shown in \([6]\) that the end-to-end sojourn time of a customer is distributed in the limit as a sum of independent exponential sojourns at each node \( j \), \( (j = 1, \ldots, n) \), with respective parameters \( \mu_{j} - \lambda_{j} \). That is, the sojourn time of a job in the network depends on its type only through the itinerary followed, and sojourn time of different type jobs in the same node have the same distribution.

To illustrate the method of calculation, consider an example run of the age process at the output of the path, as shown in Fig. 2. At time \( d_{i}^{c} \), the age of the type-\( c \) update packet received at the output of the path is equal to its end-to-end sojourn time \( S_{i}^{c,1\rightarrow n} \) through the whole path.

The AoI at the receiver is equal to the time average

\[
\hat{H}^{c} = \lim_{t \to \infty} \frac{1}{t} \left( \sum_{i=1}^{N(t)-1} (S_{i}^{c,1\rightarrow n} + (d_{i+1}^{c} - d_{i}^{c})/2)(d_{i+1}^{c} - d_{i}^{c}) \right),
\]

where \( N(t) \) is the number of packet departures in time \( t \) (the receiver perceives a mean age \( S_{i}^{c,1\rightarrow n} + (d_{i+1}^{c} - d_{i}^{c})/2 \) in the interval \([d_{i}^{c}, d_{i+1}^{c}])\).

Similarly, we can define the left and right limits bounding the AoI:

\[
\hat{H}_{left}^{c} = \lim_{t \to \infty} \frac{1}{t} \left( \sum_{i=1}^{N(t)-1} S_{i}^{c,1\rightarrow n} (d_{i+1}^{c} - d_{i}^{c}) \right), \quad (3a)
\]

\[
\hat{H}_{right}^{c} = \lim_{t \to \infty} \frac{1}{t} \left( \sum_{i=1}^{N(t)-1} (S_{i}^{c,1\rightarrow n} + d_{i+1}^{c} - d_{i}^{c})(d_{i+1}^{c} - d_{i}^{c}) \right). \quad (3b)
\]

In Fig. 2 the age process is depicted as the sawtooth curve in bold, and is right-continuous with left limits. The right limits (solid black circles) average to the lower (left) bounds of the age process \((3a)\), while the left limits (solid white circles) average to the upper (right) bounds of the update process \((3b)\). Note that \( \hat{H}_{right}^{c} \) is also referred to as the peak age, which was first introduced in \([12]\).

It is straightforward to see that

\[
\hat{H}^{c} = \hat{H}_{left}^{c} + \lim_{t \to \infty} \frac{1}{t} \left( \sum_{i=1}^{N(t)-1} (d_{i+1}^{c} - d_{i}^{c})^{2}/2 \right), \quad (4a)
\]

\[
\hat{H}_{right}^{c} = \hat{H}_{left}^{c} + \lim_{t \to \infty} \frac{1}{t} \left( \sum_{i=1}^{N(t)-1} (d_{i+1}^{c} - d_{i}^{c})^{2} \right). \quad (4b)
\]
The $D_i^c := d_{i+1}^c - d_i^c$ correspond to the interdeparture times of the type-$c$ update packets exiting the path, and, for an overtime-free path of quasi-reversable queues, in the limit $t \to \infty$ successive $D_i^c$ ($i = 1, \ldots, N(t) - 1$) are independent and have the same exponential distribution. Therefore it holds with probability 1 that $\lim_{t \to \infty} N(t)/t = 1/E[D_i^c]$. Moreover, since the interdeparture process is ergodic,

$$\lim_{t \to \infty} \frac{1}{t} \left( \sum_{i=1}^{N(t)-1} (d_{i+1}^c - d_i^c)^2 \right) = \lim_{t \to \infty} \frac{N(t) - 1}{t} \left( \sum_{i=1}^{N(t)-1} (d_{i+1}^c - d_i^c)^2 \right) = E[(D_i^c)^2]/E[D_i^c].$$

Therefore, equations (4) become:

$$\tilde{H}^c = \tilde{H}_{left}^c + \frac{E[(D_i^c)^2]}{2E[D_i^c]}, \quad (5a)$$

$$\tilde{H}_{right}^c = \tilde{H}_{left}^c + \frac{E[(D_i^c)^2]}{E[D_i^c]} \quad (5b)$$

Similarly, from (3a):

$$\tilde{H}_{left}^c = E[S_i^{c,1\to n}D_i^c]/E[D_i^c] \quad (6)$$

In summary, the Aol of a type of update packets only depends on the departure rate of that type and the expected value of the product between the sojourn time of a packet and the interdeparture interval between the departure of that type of packet and the one of the same type following it in the path.

We calculate $E[S_i^{c,1\to n}D_i^c]$ by noting that, since the network (consisting of all queues in the path) is quasi-reversable, the distribution of interarrivals is the same as the distribution of interdepartures. The system is also time-reversible, since the interval between the departure of a packet and the one that followed it in the forward-time system corresponds to the interval between the arrival of a packet and the one that preceded it in the reverse-time system. Denoting the interarrival interval $(a_i^c - a_{i-1}^c)$ by $A_i^c$, it therefore holds that $D_i^c \sim A_i^c$, and $S_i^{c,1\to n}D_i^c \sim S_i^{c,1\to n}A_i^c$.

When the parameter of the exponential interarrival distribution for type-$c$ update packets is $\lambda_c$, we have $E[(D_i^c)^2] = 2/\lambda_c^2$ and (5a), (5b), (6) become:

$$\tilde{H}^c = \tilde{H}_{left}^c + \frac{1}{\lambda_c}, \quad (7a)$$

$$\tilde{H}_{right}^c = \tilde{H}_{left}^c + \frac{2}{\lambda_c} \quad (7b)$$

$$\tilde{H}_{left}^c = \lambda_c E[S_i^{c,1\to n}D_i^c]. \quad (7c)$$

Denoting by $S_{i,j}^c$ the sojourn time of type-$c$ update packet $i$ at node $j$, we have $S_i^{c,1\to n} = \sum_{j=1}^{n} S_{i,j}^c$. We also decompose $S_{i,j}^c$ to $W_{i,j}^c$ and $X_{i,j}^c$, the waiting time and service time of this packet at node $j$. When dealing with distributions or expectations of $S_{i,j}^c$, $W_{i,j}^c$ and $X_{i,j}^c$, we can also drop the index $c$, because these distributions are independent of packet type. We have:

$$E[S_i^{c,1\to n}D_i^c] = E[S_i^{c,1\to n}A_i^c] = \sum_{j=1}^{n} E[S_i^jA_i^c]$$

$$= \sum_{j=1}^{n} E[(W_{i,j}^c + X_{i,j}^c)A_i^c]$$

$$= \sum_{j=1}^{n} (E[W_{i,j}^cA_i^c] + E[X_{i,j}^cE[A_i^c]]) \quad (8)$$

(The last step occurs because of the independence of service times and interarrival times.)

From the law of total expectation, we have for $W_{i,j}^cA_i^c$ ($j = 1 \ldots n$):

$$E[W_{i,j}^cA_i^c] = E[E[W_{i,j}^cA_i^c|x]]$$

$$= \int_{0}^{\infty} \int_{0}^{\infty} x(t-x)f_{S_{i,j}^c}(t)f_{A_i^c}(x)dt dx \quad (9)$$

(Note again that, although a packet of different type may be ahead of the type-$c$ packet in the queue, they all have the same sojourn time distribution.)

For a path with input (and output) rate of type-$c$ update packets $\lambda_c$, total arrival rate (of all type update packets) $\lambda_j$ and service rate $\mu_j$ at each queue $j$ (irrespective of the type of update packets), we have the probability density functions $f_{S_{i,j}^c} = (\mu_j - \lambda_j)e^{-(\mu_j - \lambda_j)t}$, and $f_{A_i^c}(x) = \lambda_c e^{-\lambda_c x} \forall i$. Hence

$$E[W_{i,j}^cA_i^c] = \int_{0}^{\infty} \lambda_c e^{-\lambda_c x} \int_{0}^{\infty} x(t-x)(\mu_j - \lambda_j)e^{-(\mu_j - \lambda_j)t}dt dx \quad (10)$$

Combining (8) and (10), we finally get

$$E[S_i^{c,1\to n}D_i^c] = \sum_{j=1}^{n} \left( \frac{\lambda_c}{(\mu_j - \lambda_j)(\lambda_c + \mu_j - \lambda_j)^2} + \frac{1}{\mu_j \lambda_c} \right) \quad (11)$$

It is also noted that since each of the successive sojourn, interarrival and service times are i.i.d., the above expectations are the same for all successive packets in the limit, and the subscripts $i$ can be dropped for notational simplicity.

Finally, from (7c) and (7a), we get the Aol for type-$c$ update packets at the output of the path:

$$\tilde{H}^c = \sum_{j=1}^{n} \left( \frac{\lambda_j^2}{(\mu_j - \lambda_j)(\lambda_c + \mu_j - \lambda_j)^2} + \frac{1}{\mu_j \lambda_c} \right) \quad (12)$$

Summarizing, this section has showed that the Aol for a type of update packets at the output of a quasi-reversable, overtake-free path depends only on the distribution of sojourn times at
each node in the path, and the rate of incoming (and outgoing) packets of that type in and out of the path. In the next section we provide some examples to show how the method is applied.

V. EXAMPLES

A. Queues in tandem

Consider the simple network shown in Fig. 1, where \( n \) queues are connected in tandem and there is only one type of update packets, which arrive at the network with rate \( \lambda \). For simplicity, it is assumed that the queues have the same service rate \( \mu \). In order for the queues to be stable, we must have \( \rho = \lambda/\mu < 1 \). It is noted that so far results have only been taken for a network of M/M/1/1 preemptive LCFS queues (\( \rho \geq 1 \))

Taking \( \lambda_j = \lambda_x = \lambda \) and \( \mu_j = \mu \forall j \), (12) becomes:

\[
\bar{H} = \frac{n\rho^2}{\mu - \lambda} + \frac{n}{\mu} + 1. \tag{13}
\]

Fig. 3 shows results for the mean AoI for \( \mu = 1 \) and varying values of \( \lambda, n \). The same pattern appears, as in the case of a single queue: the minimum is achieved for some intermediate load value. This value also decreases with the number of nodes; for \( n = 1 \), the minimum is achieved approximately at \( \rho = 0.53 \), for \( n = 2 \) at \( \rho = 0.46 \), for \( n = 5 \) at \( \rho = 0.37 \) and for \( n = 10 \) at \( \rho = 0.31 \). We also remark that the minimum AoI does not increase significantly with the number of nodes. The effect of the added queues is more apparent as the load increases, and the age can increase tremendously for large values. In the figure, the AoI increase for \( \rho = 0.99 \) is equal to 891 time units between \( n = 1 \) and \( n = 10 \).

Fig. 3: AoI for \( n \) identical M/M/1 queues in tandem, for \( \mu = 1 \) and varying values of the arrival rate \( \lambda \).

B. A simple network with two classes

Consider the network with two classes shown in Fig. 4. There are two types of update packets \( \alpha \) and \( \beta \), entering the network as Poisson processes with rates \( \lambda_\alpha \) and \( \lambda_\beta \), through nodes 1 and 2 respectively. Both types of packets exit the network through node 3. The service times at each node \( i \) are exponential with mean \( \mu_i^{-1} \). We will calculate the AoI for each class of update packets at the exit of the network. It is noted that the case where two classes of traffic directly share an output queue (without passing through a network) was studied in [10].

We assume that the stability conditions \( \lambda_\alpha < \mu_1, \lambda_\beta < \mu_2 \) and \( \lambda_\alpha, \lambda_\beta < \mu_3 \) hold. Since all queues are quasi-reversible and all paths are overtake-free, internal traffic is Poisson and the rates of class-\( \alpha \) and class-\( \beta \) update packets at the output of nodes 1 and 2 are also \( \lambda_\alpha \) and \( \lambda_\beta \), respectively, and the output rate at node 3 is \( \lambda_\alpha + \lambda_\beta \). The mean sojourn time at nodes 1 and 2 are \((\mu_1 - \lambda_\alpha)^{-1}\) and \((\mu_2 - \lambda_\beta)^{-1}\), respectively, while the mean sojourn time of any type update packet at node 3 is \((\mu_3 - (\lambda_\alpha + \lambda_\beta))^{-1}\).

From (12), substituting \( \mu_1 = \lambda_\alpha, \mu_2 = \lambda_\beta, \mu_3 = \lambda_\alpha + \lambda_\beta, \) and taking \( \lambda_\epsilon = \lambda_\alpha \), we can calculate the AoI of class-\( \alpha \) update packets at the output of node 3:

\[
\bar{H}_\alpha = \frac{\rho_1^2}{\mu_1 - \lambda_\alpha} + \frac{\lambda_\alpha^2}{(\mu_3 - (\lambda_\alpha + \lambda_\beta))(\mu_3 - \lambda_\beta)^2} + \frac{1}{\mu_1} + \frac{1}{\mu_3} + \frac{\lambda_\alpha}{\lambda_\beta},
\]

where \( \rho_1 = \lambda_1/\mu_1 = \lambda_\alpha/\mu_1 \) is the load at queue 1. By symmetry, the AoI of class-\( \beta \) update packets in the path \( 2 \rightarrow 3 \), \( \bar{H}_\beta \), can then be found by interchanging \( \lambda_\alpha \) and \( \lambda_\beta \), and \( \mu_1 \) and \( \mu_2 \) in the expression for \( \bar{H}_\alpha \), and also defining \( \rho_2 = \lambda_2/\mu_2 = \lambda_\beta/\mu_2 \):

\[
\bar{H}_\beta = \frac{\rho_2^2}{\mu_2 - \lambda_\beta} + \frac{\lambda_\beta^2}{(\mu_3 - (\lambda_\alpha + \lambda_\beta))(\mu_3 - \lambda_\alpha)^2} + \frac{1}{\mu_2} + \frac{1}{\mu_3} + \frac{\lambda_\beta}{\lambda_\alpha}.
\]

Fig. 5a shows how the AoI of update packets in the output of the path \( 1 \rightarrow 3 \) varies in the region of \( \lambda_\alpha, \lambda_\beta \) values (the service rates \( \mu_i \) are fixed at 1 for all queues). The AoI is represented by color intensities in the different parts of the feasible region. It can be seen that \( \bar{H}_\alpha \) primarily depends on the value of \( \lambda_\alpha \); the arrival of class-\( \beta \) packets effects by increasing \( \bar{H}_\alpha \), but only when \( \lambda_\beta \) is high enough, so that the queue is close to saturation. Fig. 5b shows a scatter plot of \( \bar{H}_\alpha, \bar{H}_\beta \) values in the region defined by \( \lambda_\alpha, \lambda_\beta \). We notice that there are subregions where the AoI of one class of update packets stays relatively stable while the other’s increases, and subregions where they are inversely related. This is in agreement with Fig. 5a, where increasing the arrival rate of one class always effects on the AoI of that class, but the other class is affected only when the output queue is close to saturation.
The minimum of $\bar{H}_\alpha$ equals 4.97 and is achieved when $\lambda_1 = 0.46$ and $\lambda_2 = 0$ (and vice-versa for $\bar{H}_\beta$). This is anticipated, as it coincides with the minimum $\rho$ value for $n = 2$ in the case of queues in tandem. The minimum value for $\bar{H}_\alpha + \bar{H}_\beta$ equals 11.78 and is achieved for $\lambda_1 = 0.32$, corresponding to $\bar{H}_\alpha = \bar{H}_\beta = 5.89$. This is slightly higher than the case where classes $\alpha$ and $\beta$ directly share the output queue, which was studied in [10], [11], and where the sum was minimized at $\lambda_1 = \lambda_2 \approx 0.3$, yielding $\bar{H}_\alpha = \bar{H}_\beta \approx 5.34$.

VI. CONCLUSIONS AND EXTENSIONS

This paper presented a method for calculating the AoI in quasi-reversible, overtake-free networks of queues. The presented analysis simplified and extended the original analysis in [1] for FCFS single-server queues, and revealed the relationship between the AoI and its upper and lower bound (the average maximum age seen be the destination before reception of an update, known as the peak age, and the average minimum age right after reception of an update).

Results for $n$ identical M/M/1 queues in tandem showed that, although the minimum AoI value (achieved for some intermediate load value, which decreases with $n$) does not increase significantly with the number of queues, the effect of the added queues becomes apparent for high load values, and the age can then increase tremendously. Results for a simple network with two classes of update packets, entering through different queues in the network but sharing the output queue, revealed that changing the arrival rate of one class always effects on the AoI of that class, but the other class is significantly affected only when the output queue is close to saturation.

The use of the method for calculating the AoI requires that queues in the network path are Markovian and quasi-reversible, the path is overtake-free, and that the sojourn and interarrival time distributions at each queue are known. Quasi-reversibility for Markovian queues is equivalent to the Poisson-in-Poisson-out property and holds more generally for M/M/c queues with exponential constant-rate arrivals, as well as for processor-sharing queues in BCMP networks. Sojourn time distributions are well-known for the first (see e.g. [16]), and also sometimes exist for the latter [17].

REFERENCES

Energy Packet Networks with general service time distribution

Youssef Ait El Mahjoub
DAVID-UVSQ & SAMOVAR-TSP
Université Paris-Saclay & Télécom SudParis
Versailles, France
youssef.ait-el-mahjoub@uvsq.fr

Jean-Michel Fourneau
DAVID-UVSQ
Université Paris-Saclay
Versailles, France
Jean-Michel.Fourneau@uvsq.fr

Hind Castel-Taleb
SAMOVAR, Télécom SudParis
Institut Polytechnique de Paris,
Evy, France
hind.castel@telecom-sudparis.eu

Abstract—We consider an Energy Packet Network (EPN) model where the service times of the packets follow a Coxian distribution. EPNs are a recent type of models proposed by Gelenbe and his colleagues [1]–[4]. They are used to study interactions between energy consumption and the processing or the transmission of data. We prove that such a network has a product form solution for its steady-state distribution. We also state some sufficient conditions for the existence of the flow equations. Finally we study the performances and the energy consumption and we show how to optimize the assignment of Δ solar panels over a sensor network.

Index Terms—Energy Packet Network, Product Form equilibrium distribution, Cox process

I. INTRODUCTION

Energy Packet Networks (EPNs) were recently introduced by Gelenbe and his colleagues [1]–[4]. They are used to study interactions between IT devices consuming energy like sensors, cpu, storage systems and networking elements and the flow of intermittent sources of energy like batteries or solar or wind based generators. The most important model is based on the discretization of energy using the so-called Energy Packets (EPs). Each EP represents a certain number of Joules. As we model production by intermittent sources of energy (typically wind or solar, less frequently tides), the flow of EPs is associated with some random processes, EPs are consumed by some devices after some random duration to perform requested works or can also be stored in a battery from which they can also leak after a random delay. Their absence creates delay (if energy is missing to process a data packet) or a loss (if energy is missing to receive a radio packet). Both aspects can be represented in this type of model. Some EP Networks (EPN in the following) are linked to the theory of G-networks of queues and signals, which was introduced by the seminal papers by Gelenbe on networks of queues with positive and negative customers [5]. Since then, many networks of queues with various signals (Triggers for one the first papers [6] and Adders for a recent one [7]) where proved to have a product form for the steady-state distribution of positive customers in the queues under some classical assumptions. In the literature, such results were obtained under classical assumptions: Poisson arrivals of packets (both EP and data), exponential duration for service and leakage, independence, Markovian routing. Other stochastic processes, like Interrupted Poisson Process [8], have been considered to model energy packet generation variation during a day. Here, we extend the results to general service time distribution. We assume that the service times of the Energy Packets which trigger the data processing follow a Coxian distribution. Since all non-negative distributions can be arbitrarily closely approximated by Coxian distributions [9], we can use them to model EP networks with general service times for both the energy consumption of energy packets and data packets processing.

The technical part of the paper is as follows. In section II, we describe the EPN model. We give the proof of the product form steady-state distribution. We also study the existence of a solution to the flow equations. In Section III, we focus on performance evaluation and energy losses rate. Finally, in section IV, we show how to optimize a sensor network with a solar panels harvesting capacity and we present two methods to distribute the panels among the stations to optimize the average delay. This illustrates one of the main advantages of EP networks models. Based on its closed form solution, it is possible to conduct an optimisation of the systems based on some utility function like mean response time or loss rates [4], [10]–[12].

II. MODEL DESCRIPTION AND MARKOV CHAIN ANALYSIS

We consider a distributed system (as a fog [13] or a sensor system) consuming energy provided by intermittent sources. It is represented by an EPN (see Fig. 1) which is an open network with N cells, where each of the cells is represented by one queue that stores DPs (Data Packets) and one battery that stores EPs (Energy Packets). Both EP and DP queues have an infinite capacity. DPs arrive to each cell i following a Poisson distribution with parameter \( \lambda_i \), and EPs arrive to cell i following also a Poisson distribution with rate \( \alpha_i > 0 \). We suppose that queuing discipline is Processor Sharing, and the service is applied on EP queues and follows the Cox probability distribution law with K phases. Each phase \( n \in \{1, \ldots, K\} \) of cell \( i \in \{1, \ldots, N\} \) generates a transition with an exponential distribution with parameter \( \mu_{n,i} > 0 \). At each phase \( n \) there is a probability \( p_{i,n} \) that the corresponding exponential transition occurred, and a probability \( 1 - p_{i,n} \) that the process ends. Also note that \( p_{i,K} = 0 \) and we suppose that \( 0 \leq p_{i,n} \leq 1 \) for all \( n \in \{1, \ldots, K - 1\} \). The process could...
reach the last phase $K$ that generates the last transition $\mu_{i,K}$. We consider that there are energy leakages with exponential times and only occur in the first phase of the Cox process. Let $\gamma_i > 0$ be the leakage rate of one EP in the battery of cell $i$. The service takes place in EP queues, and is described as follows for the cell $i$ and at each phase $n$ ($0 \leq n < K$):

- With rate $\mu_{i,n}(1 - p_{i,n})$ an EP packet is consumed and a data packet is sent;
- with rate $\mu_{i,n}p_{i,n}$ the EP service continues, to go to the next phase.

Note that the triggered data packet either goes outside (with probability $d_i$) or is routed to another DP queue of cell $j$ ($1 \leq j \leq N$) with probability $P(i,j)$. Hence, for all $i$ ($1 \leq i \leq N$):

$$d_i + \sum_{j=1}^{N} P(i,j) = 1. \quad (1)$$

If the DP queue is empty, the energy packet is lost. If no EPs are present in the battery of a cell then DPs are blocked in the queue until the next external arrival of an EP. This model represents two important features: first without energy, the data packets are not served, and second even if there is no data packets in the server, there is still some energy consumption.

A. Markov chain analysis

In this section we propose to analyse the system with a Markov chain model. For each cell $i$ we have the following components:

- $X_i$ is the number of DPs at the DP queue of cell $i$;
- $Y_{i,n}$ is the number of energy packets in phase $n$ at the battery of cell $i$.

Let us note by $X = (X_1, \ldots, X_N)$ and $Y_n = (Y_{1,n}, \ldots, Y_{N,n})$. Under the assumptions about the arrivals and the services, $\{(X, Y_1, \ldots, Y_K), t \geq 0\}$ is a continuous-time Markov chain where:

- on $X$ component, transitions are due to (i) external arrivals of DPs, (ii) service departure of a DP in phase $n$ ($n \in \{1, \ldots, K\}$), and (iii) transit of DPs after routing between the cells;
- on $Y_1$ component, transitions are caused by (i) external arrivals of EPs, (ii) Leakage of an EP, (iii) transformation to phase 2 of an EP after a service, and (iii) the triggering (departure/routing) of a DP after a service;
- on $Y_K$ component, transitions are (i) arrivals of EPs by transformation from phase $K - 1$, and (ii) the triggering of a DP (departure/routing) after a service.

In the next theorem, we prove that the Markov chain $\{(X, Y_1, \ldots, Y_K), t \geq 0\}$ has a product form steady-state distribution if the flow equations have a solution which satisfies the stability constraints. Let us first introduce the flow equations, for all $i \in \{1, \ldots, N\}$:

$$\beta_{i,1} = \frac{\alpha_i}{\gamma_i + \mu_{i,1}}, \ \beta_{i,n} = \left(\frac{p_{i,n-1}\mu_{i,n-1}}{\mu_{i,n}}\right)\beta_{i,n-1} \quad (2)$$

which is equivalent to:

$$\beta_{i,1} = \frac{\alpha_i}{\gamma_i + \mu_{i,1}}, \ \forall n \in \{2, \ldots, K\} \ \beta_{i,n} = \frac{\mu_{i,1}\beta_{i,1}}{\mu_{i,n}} \prod_{r=1}^{n-1} p_{i,r}. \quad (3)$$

Also we have:

$$\rho_i = \frac{\lambda_i + \sum_{n=1}^{K} \sum_{j=1}^{N} (1 - p_{j,n})\mu_{j,n}p_{j,2}\mu_{j,2}P(j,i)}{\sum_{n=1}^{K} (1 - p_{i,n})\mu_{i,n}p_{i,2}\mu_{i,2}}. \quad (3)$$

Lemma 2.1: Flow equations of EPs in each cell $i$ are:

$$\gamma_i\beta_{i,1} + \sum_{n=1}^{K} (1 - p_{i,n})\mu_{i,n}\beta_{i,n} = \alpha_i. \quad (4)$$

Where the l.h.s of the equation represents the total rate for the departure of an EP from a cell $i$ and the r.h.s stands for the arrivals rate of an EP to cell $i$.

Proof: By developing the second term of the l.h.s of (4), we have:

$$\gamma_i\beta_{i,1} + \sum_{n=1}^{K} (1 - p_{i,n})\mu_{i,n}\beta_{i,n} = \gamma_i\beta_{i,1} + \mu_{i,1}\beta_{i,1} - p_{i,1}\mu_{i,1}\beta_{i,1} + \mu_{i,2}\beta_{i,2} - p_{i,2}\mu_{i,2}\beta_{i,2} + \cdots + \mu_{i,K-1}\beta_{i,K-1} - p_{i,K-1}\mu_{i,K-1}\beta_{i,K-1} - p_{i,K}\mu_{i,K}\beta_{i,K}. $$

By substituting using (2) and using that $p_{i,K} = 0$:

$$\gamma_i\beta_{i,1} + \sum_{n=1}^{K} (1 - p_{i,n})\mu_{i,n}\beta_{i,n} = \beta_{i,1}(\gamma_i + \mu_{i,1}) - p_{i,1}\mu_{i,1}\beta_{i,1} + \mu_{i,2}\beta_{i,2} - p_{i,2}\mu_{i,2}\beta_{i,2} + \cdots + \mu_{i,K-1}\beta_{i,K-1} - p_{i,K-1}\mu_{i,K-1}\beta_{i,K-1} + p_{i,K}\mu_{i,K}\beta_{i,K}. $$

All r.h.s terms of the last equation are eliminated except the first one, so:

$$\gamma_i\beta_{i,1} + \sum_{n=1}^{K} (1 - p_{i,n})\mu_{i,n}\beta_{i,n} = \beta_{i,1}(\gamma_i + \mu_{i,1}) = \alpha_i.$$

Fig. 1. An EPN network with three cells.
The proof is complete.

**B. Product form of the EPN network**

**Theorem 2.1:** Assume that Markov chain \((X, Y_1, \ldots, Y_K)_t\) is ergodic. If the flow equations (2) and (3) have a fixed point solution \((\rho_1, \beta_{1,1}, \ldots, \beta_{i,K})\) such that \(\rho_i < 1\) and \(\sum_{n=1}^{K} \beta_{i,n} < 1\), then the steady-state distribution is:

\[
\pi(X, Y_1, \ldots, Y_K) = G \prod_{i=1}^{N} (\rho_i)^{X_i} \prod_{n=1}^{K} \frac{(\beta_{i,n})^{Y_{i,n}}}{Y_{i,n}!}
\]

where \(||Y_i|| = \sum_{n=1}^{K} Y_{i,n}\). See Lemma 2.2 for the value of \(G\).

**Proof:** The proof of this theorem is based on the analysis of the global balance equation. First, let us introduce some notations. As usual, \(e_i\) will denote a vector whose components are all 0 except component with index \(i\) which is 1. Moreover, let \(1_e\) be the step function equal to 1 when condition \(C\) is true and 0 otherwise. Let us now write the Chapman Kolmogorov equation at steady-state:

\[
\pi(X, Y_1, \ldots, Y_K) \left[ \sum_{i=1}^{N} \left( \lambda_i + \alpha_i + \gamma_i Y_{i,1} \frac{1}{||Y_i||} \right) 1_{|Y_i|>0} + \sum_{n=1}^{K} \mu_i Y_{i,n} \frac{1}{||Y_i||} 1_{|Y_i|>0} \right ]
\]

\[
= \sum_{i=1}^{N} \lambda_i \pi(X - e_i, Y_1, \ldots, Y_K) 1_{X_i>0}
\]

\[
+ \sum_{i=1}^{N} \alpha_i \pi(X, Y_1 - e_i, Y_2, \ldots, Y_K) 1_{Y_{1,i}>0}
\]

\[
+ \sum_{i=1}^{N} \gamma_i \frac{Y_{i,1} + 1}{||Y_i|| + 1} \pi(X, Y_1, \ldots, Y_n + e_i, Y_{n+1} - e_i, \ldots, Y_K) 1_{Y_{n+1,1}>0}
\]

\[
+ \sum_{n=1}^{N} \sum_{i=1}^{N} (1 - p_{i,n}) d_i \mu_i Y_{i,n} + 1 \pi(X, Y_{i,1} + 1) 1_{X_i>0}
\]

\[
+ \sum_{n=1}^{N} \mu_i \frac{1}{||Y_i|| + 1} \pi(X + e_i, Y_1, \ldots, Y_n + e_i, Y_{n+1} - e_i, \ldots, Y_K) 1_{X_i>0}
\]

\[
+ \sum_{n=1}^{N} \sum_{i=1}^{N} (1 - p_{i,n}) \mu_i P(i,j) Y_{i,n} + 1 \pi(X, Y_{i,1} + 1) 1_{X_j>0}
\]

\[
+ \sum_{n=1}^{N} \mu_i \frac{1}{||Y_i|| + 1} \pi(X + e_i, Y_1, \ldots, Y_n + e_i, Y_{n+1} - e_i, \ldots, Y_K) 1_{X_i=0}
\]

Let’s describe the right-hand side of the balance equation (6). Term [1] (resp. Term [2]) represents an external arrival of a DP (resp. EP) in the DP (resp. EP) queue of cell \(i\). Term [3] represents the leakage of an EP from the battery of cell \(i\). Term [4] represents the transition of an EP from phase \(n\) to phase \(n+1\) after a service in cell \(i\). Term [5] (resp. Term [6]) is the departure (resp. routing transitions from a cell \(i\) to a cell \(j\)) of a DP after a service in each phase \(n\), the triggering EP also vanishes. The last term describes the consumption of an EP while the DP queue is empty.

We divide both sides of (6) by \(\pi(X, Y_1, \ldots, Y_K)\) and take into account the product form of the solution and the simplifications. We obtain:

\[
\sum_{i=1}^{N} \left( \frac{\lambda_i + \alpha_i + \gamma_i Y_{i,1}}{||Y_i||} 1_{|Y_i|>0} + \sum_{n=1}^{K} \frac{\mu_i Y_{i,n}}{||Y_i||} 1_{|Y_i|>0} \right )
\]

\[
= \sum_{i=1}^{N} \frac{\lambda_i}{\rho_i} 1_{X_i>0}
\]

\[
+ \sum_{i=1}^{N} \frac{\alpha_i Y_{i,1}}{\beta_{i,1} ||Y_i||} 1_{|Y_i|>0}
\]

\[
+ \sum_{i=1}^{N} \frac{\gamma_i \beta_{i,1}}{1 - p_{i,n} \rho_i \mu_i \beta_{i,n} Y_{i,n+1}} 1_{|Y_i|>0}
\]

\[
+ \sum_{n=1}^{K} \sum_{i=1}^{N} (1 - p_{i,n}) d_i \mu_i Y_{i,n} + 1 \frac{1}{||Y_i||} 1_{X_i>0}
\]

\[
+ \sum_{n=1}^{K} \sum_{i=1}^{N} (1 - p_{i,n}) \mu_i P(i,j) Y_{i,n} + 1 \frac{1}{||Y_i||} 1_{X_j>0}
\]

\[
+ \sum_{n=1}^{K} \sum_{i=1}^{N} (1 - p_{i,n}) \mu_i \frac{1}{||Y_i||} 1_{X_i=0}
\]

In Term [4] of the r.h.s. of (7), we apply an index change on \(n\). In Term [6], we substitute \(1_{X_i} = 0\) with \(1_{X_i>0}\) and we move the negative terms in the l.h.s. to get:

\[
\sum_{i=1}^{N} \left( \frac{\lambda_i + \alpha_i + \gamma_i Y_{i,1}}{||Y_i||} + \sum_{n=1}^{K} \frac{\mu_i Y_{i,n}}{||Y_i||} \right ) 1_{|Y_i|>0}
\]

\[
+ \sum_{n=1}^{K} \left( 1 - p_{i,n} \rho_i \mu_i \beta_{i,n} Y_{i,n+1} 1_{X_i>0} \right ) = \sum_{i=1}^{N} \frac{\lambda_i}{\rho_i} 1_{X_i>0}
\]

\[
+ \sum_{i=1}^{N} \frac{\alpha_i Y_{i,1}}{\beta_{i,1} ||Y_i||} 1_{|Y_i|>0}
\]

\[
+ \sum_{i=1}^{N} \frac{\gamma_i \beta_{i,1}}{1 - p_{i,n} \rho_i \mu_i \beta_{i,n} Y_{i,n+1}} 1_{|Y_i|>0}
\]

\[
+ \sum_{n=1}^{K} \sum_{i=1}^{N} (1 - p_{i,n}) d_i \mu_i Y_{i,n} + 1 \frac{1}{||Y_i||} 1_{X_i>0}
\]

\[
+ \sum_{n=1}^{K} \sum_{i=1}^{N} (1 - p_{i,n}) \mu_i P(i,j) Y_{i,n} + 1 \frac{1}{||Y_i||} 1_{X_j>0}
\]

\[
+ \sum_{n=1}^{K} \sum_{i=1}^{N} (1 - p_{i,n}) \mu_i \frac{1}{||Y_i||} 1_{X_i=0}
\]

We decompose (8) into three equations system based on their step functions:

\[
\sum_{i=1}^{N} \sum_{n=1}^{K} (1 - p_{i,n}) \beta_{i,n} Y_{i,n} 1_{X_i>0} = \sum_{i=1}^{N} \frac{\lambda_i}{\rho_i}
\]

\[
+ \sum_{n=1}^{K} \sum_{j=1}^{N} (1 - p_{i,n}) \mu_j P(j,i) \frac{\rho_i \beta_{j,n}}{\rho_j} 1_{X_j>0}
\]
and
\[
\left[ \sum_{n=1}^{N} \frac{1}{Y_{i,n}!} \left( Y_{i,n} \cdot \frac{1}{\pi Y_i} \right) + \sum_{i=1}^{N} \mu_{i,1} \left( Y_{i,n} \cdot \frac{1}{\pi Y_i} \right) \right] 1\|Y_i\| > 0 = \left[ \sum_{i=1}^{N} \frac{\alpha_i}{\gamma_i + \mu_{i,1}} \left( Y_{i,n} \cdot \frac{1}{\pi Y_i} \right) \right] 1\|Y_i\| > 0
\]
\[
+ \sum_{n=1}^{N} \frac{p_{i,n}}{\beta_i, n} \left( Y_{i,n} \cdot \frac{1}{\pi Y_i} \right) 1\|Y_i\| > 0
\]
and
\[
\sum_{n=1}^{N} (\lambda_i + \alpha_i) = \sum_{n=1}^{N} (1 - p_{i,n}) d_{i,n} \mu_{i,1} \beta_i, n + \sum_{i=1}^{N} \gamma_i \beta_i, 1.
\]
(10)
The values of \((\rho_i, \beta_i, 1, \ldots, \beta_i, K)\) we are looking for have to satisfy the three equations (9), (10) and (11).
First by the factorisation of \(\rho_i\) in (9) we get, for all \(i\):
\[
\rho_i = \lambda_i + \sum_{n=1}^{N} (1 - p_{i,n}) \mu_{i,n} \beta_i, n
\]
(12)
Next we decompose (10) into two equations. And we verify that two equations are satisfied.
\[
\sum_{n=1}^{N} (\gamma_i + \mu_{i,1}) \left( Y_{i,n} \cdot \frac{1}{\pi Y_i} \right) 1\|Y_i\| > 0
\]
\[
= \sum_{n=1}^{N} \frac{\alpha_i}{\gamma_i + \mu_{i,1}} \left( Y_{i,n} \cdot \frac{1}{\pi Y_i} \right) 1\|Y_i\| > 0
\]
and
\[
\sum_{n=1}^{N} (1 - p_{i,n}) \mu_{i,n} \beta_i, n + \sum_{i=1}^{N} \gamma_i \beta_i, 1.
\]
By the elimination of the multiplicative terms on both sides of the equations. We get in the first equation:
\[
\forall i \in \{1, \ldots, N\}, \quad \beta_i, 1 = \frac{\alpha_i}{\gamma_i + \mu_{i,1}},
\]
(13)
and in the second equation, \(\forall i \in \{1, \ldots, N\}\):
\[
\forall n \in \{2, \ldots, K\}, \quad \beta_i, n = \frac{p_{i,n-1} \mu_{i,n-1} \beta_i, n-1}{\mu_{i,n}}.
\]
(14)
Finally, we only have to check that the values of \((\rho_i, \beta_i, 1, \ldots, \beta_i, K)\) in (12), (13) and (14) also satisfy (11).
To verify (11), we use Lemma 2.1 and (1) (which states that \(\forall i, d_i = 1 - \sum_{j=1}^{N} P(i, j)\)). By moving negative terms to the l.h.s, (11) is reduced to 0 = 0. Therefore, all equations that constitute the global balance equation at steady-state are satisfied and the proof of the theorem is complete.

**Lemma 2.2:** The normalization constant is equal to:
\[
G = \prod_{i=1}^{N} G_i,
\]
(15)
where
\[
G_i = (1 - \rho_i)(1 - \sum_{n=1}^{K} \beta_i, n).
\]
(16)

**Proof:** We compute the value of \(G\) using that \(\forall X_i, Y_{i,1}, \ldots, Y_{i,K}\) \(\pi(X_i, 1, \ldots, Y_{i,K}) = 1\) and (5), then:
\[
\sum_{X_i, Y_{i,1}, \ldots, Y_{i,K}} G \prod_{i=1}^{N} (\rho_i)^{Y_{i,1}} \prod_{i=1}^{K} (\beta_i, n)^{Y_{i,n}} = 1.
\]
\[
\Rightarrow G \sum_{i=1}^{N} \sum_{Y_{i,1}, \ldots, Y_{i,K}} (\rho_i)^{Y_{i,1}} \prod_{i=1}^{K} (\beta_i, n)^{Y_{i,n}} = 1.
\]
From the assumption of the theorem \(\rho_i < 1\), then the sum on \(X_i\) converges:
\[
G \sum_{i=1}^{N} \frac{1}{1 - \rho_i} \sum_{Y_{i,1}, \ldots, Y_{i,K}} ||Y_i||! \prod_{i=1}^{K} (\beta_i, n)^{Y_{i,n}} = 1.
\]
We partition the summation on \(Y_{i,1}, \ldots, Y_{i,K}\) according to its norm \(||Y_i||\):
\[
G \sum_{i=1}^{N} \frac{1}{1 - \rho_i} \sum_{Y_{i,1}, \ldots, Y_{i,K}} \sum_{||Y_i||=m} ||Y_i||! \prod_{i=1}^{K} (\beta_i, n)^{Y_{i,n}} = 1.
\]
Substitute \(||Y_i||\) by \(m\) in the previous equation. Remember the definition of the multinomial theorem:
\[
\sum_{Y_{i,1}, \ldots, Y_{i,K}} ||Y_i||! \prod_{i=1}^{K} (\beta_i, n)^{Y_{i,n}} = \left( \sum_{n=1}^{K} \beta_i, n \right)^m.
\]
After substitution we obtain:
\[
G \prod_{i=1}^{N} \frac{1}{1 - \rho_i} \sum_{m=0}^{\infty} \left( \sum_{n=1}^{K} \beta_i, n \right)^m = 1.
\]
As by assumption \(\sum_{n=1}^{K} \beta_i, n < 1\), the sum converges, then:
\[
G \prod_{i=1}^{N} \frac{1}{1 - \rho_i}(1 - \sum_{i=1}^{K} \beta_i, n) = 1.
\]
The proof is complete.

**C. Existence of a fixed point solution**

The main result of our work states that a product form of the EPN exists if the solution of the fixed point problem defined in (2) and (3) exists. First of all we remind some properties related to the network.

**Assumption 2.1:** We consider an open and connected EPN.

**Lemma 2.3:** Let \(P\) be the routing matrix of a open connected EPN. Then \((I - P)^{-1}\) is non singular i.e. \((I-d-P)^{-1}\) exists.

The proof of this Lemma is in [14].

**Lemma 2.4:** For all \(i\) and \(j\) in \(\{1, \ldots, N\}\) and \(n \in \{1, \ldots, K\}\), we have \(\beta_{i,n}\) defined in (2) and \(\theta_j\) exists i.e. \(\beta_{i,n} > 0\) and \(\theta_j > 0\). Note that:
\[
\theta_j = \sum_{n=1}^{K} (1 - p_{j,n}) \mu_{j,n} \beta_{j,n}.
\]
(17)

**Proof:** By definition, we have for all \(i \in \{1, \ldots, N\}\) and \(n \in \{1, \ldots, K\}\), \(\mu_{i,n} > 0\), \(\alpha_i > 0\) and \(\gamma_i > 0\), then \(\beta_{i,n}\) expressed in (2) exists and \(\beta_{i,n} > 0\).
\[ \theta_j > 0 \text{ since (i) } \mu_{j,n} > 0, \text{ (ii) } \beta_{j,n} > 0 \text{ and (iii) we know that at least the last term } (1 - \rho_j \lambda_j) \mu_j k \beta_j k > 0 \text{ as } \rho_j k = 0. \]

**Lemma 2.5:** Equation (3) that represents flow equations of data packets in each cell has a fixed point \( \rho_i \).

**Proof:** By substitution using (17), (3) becomes \( \forall \ i \in \{1, \ldots, N\} \):

\[
\rho_i = \frac{\lambda_i + \sum_{j=1}^N \theta_j \rho_j P(j, i)}{\theta_i},
\]

\[
\Rightarrow \rho_i \theta_i - \sum_{j=1}^N \theta_j \rho_j P(j, i) = \lambda_i.
\]

Which represents the flow equations in Jackson networks [15].

Let \( \rho = (\rho_1, \ldots, \rho_N, \theta_N) \), \( X = (\lambda_1, \ldots, \lambda_N) \) and \( I \) the identity matrix, then from (19) we obtain for vectors:

\[
\rho \cdot (I - D) = X.
\]

Using Lemma 2.3, \( (I - D) \) admits an inverse:

\[
\Rightarrow \rho \theta = X(I - D)^{-1}.
\]

This proves the existence of the vector \( \rho \theta \), therefore \( \rho_i \) exists since \( \theta_i \) exists as stated in Lemma 2.4. ■

### III. Performance and Energy Evaluation

In this section, we compute both energy and performance measures. Since our system has a product form solution, the measures can be computed as expectations of rewards separately on each cell. Note that (5) can be expressed as:

\[
\pi(X, Y_1, \ldots, Y_K) = \prod_{i=1}^N \pi(X_i, Y_{i,1}, \ldots, Y_{i,K}),
\]

and \( \pi(X_i, Y_{i,1}, \ldots, Y_{i,K}) = \pi(X_i) \pi(Y_{i,1}, \ldots, Y_{i,K}) \)

where, by the substitution of \( G_i \), we have:

\[
\pi(X_i) = (1 - \rho_i)(\rho_i)^{X_i}, \quad \pi(0) = (1 - \rho_i) \quad \text{and} \quad \pi(Y_{i,1}, \ldots, Y_{i,K}) = (1 - \sum_{n=1}^K \beta_{i,n}) ||Y_i|| \prod_{n=1}^K \frac{Y_{i,n}}{Y_{i,n}!}.
\]

#### A. Loss rate of energy packets

For each cell in the EPN, energy packets are lost either when there is no data packets in the DP queue at the end of an EP service, or due to battery leakages which occur in phase 1 of the Cox process.

**Lemma 3.1:** Let \( \text{Loss}^T \) be the total loss rate of EPs in the network, then:

\[
\text{Loss}^T = \sum_{i=1}^N \sum_{|Y_i|>0} \pi(Y_{i,1}, \ldots, Y_{i,K}) \left[ \pi(0) r_1(Y_{i,1}, \ldots, Y_{i,K}) + \sum_{X_i} \pi(X_i) r_2(Y_{i,1}, \ldots, Y_{i,K}) \right].
\]

Where \( r_1(Y_{i,1}, \ldots, Y_{i,K}) \) (resp. \( r_2(Y_{i,1}, \ldots, Y_{i,K}) \)) is described in (28) (resp. (31)).

**Proof:** Let \( \text{Loss}^{(i)} \) be the loss rate of EPs in cell \( i \). As the EPN network has a product-form solution, then:

\[
\text{Loss}^T = \sum_{i=1}^N \text{Loss}^{(i)}.
\]

Let \( \text{Loss}_1^{(i)} \) (resp. \( \text{Loss}_2^{(i)} \)) be the loss rate of EPs due to emptiness of DP queue (resp. battery leakage) in cell \( i \), then:

\[
\text{Loss}^{(i)} = \text{Loss}_1^{(i)} + \text{Loss}_2^{(i)}.
\]

States that correspond to \( \text{Loss}_1^{(i)} \) in (27), are those where, \( X_i = 0 \) and at least one EP in the battery is ready to trigger a DP. This explains the sum on \(|Y_i|>0 \). Also, for each state, we multiply by the appropriate reward \( r_1(Y_{i,1}, \ldots, Y_{i,K}) \). The reward in this case, (28), represents the rate at which a DP is triggered in each phase of the Cox process. So:

\[
\text{Loss}_1^{(i)} = \sum_{Y_{i,1},\ldots,Y_{i,K}/|Y_i|>0} \pi(0, Y_{i,1}, \ldots, Y_{i,K}) r_1(Y_{i,1}, \ldots, Y_{i,K}),
\]

where \( r_1(Y_{i,1}, \ldots, Y_{i,K}) = \frac{\sum_{n=1}^K Y_{i,n}}{||Y_i||} (1 - p_{i,n}) \theta_{i,n}. \)

Using the separation simplifications in (21), (22) and (23), \( \text{Loss}_1^{(i)} \) can be expressed as:

\[
\text{Loss}_1^{(i)} = \pi(0) \sum_{|Y_i|>0} \pi(Y_{i,1}, \ldots, Y_{i,K}) r_1(Y_{i,1}, \ldots, Y_{i,K}).
\]

To formulate \( \text{Loss}_2^{(i)} \), we use the same approach as for \( \text{Loss}_1^{(i)} \). This means that we multiply the steady state probability of the appropriate states by the corresponding rewards \( r_2(Y_{i,1}, \ldots, Y_{i,K}) \). In this case, we are interested in the rate of EP loss due to battery leakage that occurs in the first phase of the Cox process with the rate \( \frac{Y_{i,1}}{||Y_i||} \).

\[
\text{Loss}_2^{(i)} = \sum_{X_i, Y_{i,1}, \ldots, Y_{i,K}/|Y_i|>0} \pi(X_i, Y_{i,1}, \ldots, Y_{i,K}) r_2(Y_{i,1}, \ldots, Y_{i,K}),
\]

where \( r_2(Y_{i,1}, \ldots, Y_{i,K}) = \frac{Y_{i,1}}{||Y_i||} \).

Also, using the separation simplifications in (21), (22) and (23) then:

\[
\text{Loss}_2^{(i)} = \sum_{X_i} \pi(X_i) \sum_{|Y_i|>0} \pi(Y_{i,1}, \ldots, Y_{i,K}) r_2(Y_{i,1}, \ldots, Y_{i,K}).
\]

Finally, by the substitution of \( \text{Loss}_1^{(i)} \) and \( \text{Loss}_2^{(i)} \) in (26), and using (25), we obtain (24) and the proof is complete. ■

#### B. Total number of data packets and sojourn time

**Lemma 3.2:** Let \( E[X] \) be the total number of data packets in the system, then:

\[
E[X] = \sum_{i=1}^N \left[ \frac{\lambda_i + \sum_{j=1}^N \theta_j \rho_j P(j, i)}{\theta_i - (\lambda_i + \sum_{j=1}^N \theta_j \rho_j P(j, i))} \right].
\]

See (17) for the value of \( \theta_i \).

**Proof:** We first calculate \( E[X_i] \) the mean number of data packets in cell \( i \). As the EPN network has a product-form solution, then:

\[
E[X] = \sum_{i=1}^N E[X_i].
\]
From (22) we can deduce that:

$$E[X_i] = \frac{\rho_i}{1-\rho_i}. \quad (35)$$

Then by substitution of $\rho_i$ (see (3)) and using (17) we get:

$$E[X_i] = \frac{\lambda_i + \sum_{j=1}^{N} \theta_j \rho_j P(j,i)}{\theta_i - (\lambda_i + \sum_{j=1}^{N} \theta_j \rho_j P(j,i))}. \quad (36)$$

Note that $E[X_i] > 0$ since from stability conditions (see Theorem 2.1), we have for all $i$, $\rho_i < 1$ then $\theta_i > 0$ and $\lambda_i + \sum_{j=1}^{N} \theta_j \rho_j P(j,i)$. Finally, using (34) we obtain (33) and the proof is complete.

**Lemma 3.3:** Let $E[T_i]$ be the average sojourn time of a data packet in cell $i$, then:

$$E[T_i] = \frac{1}{\theta_i - (\lambda_i + \sum_{j=1}^{N} \theta_j \rho_j P(j,i))}. \quad (37)$$

**Proof:** Let $\Lambda_i$ be the overall arrival rate to DP queue of cell $i$, including both external arrivals and internal transitions:

$$\Lambda_i = \lambda_i + \sum_{n=1}^{K} (1-p_{j,n})\mu_{j,n}\rho_j \beta_{j,n} P(j,i). \quad (38)$$

Then, from Little’s law for data packets and (35):

$$E[T_i] = \frac{E[X_i]}{\Lambda_i} \Rightarrow E[T_i] = \frac{\rho_i}{\Lambda_i(1-\rho_i)}. \quad (39)$$

Also using (3) to substitute $\rho_i$, (38) to substitute $\Lambda_i$ and (17), we obtain after simplifications:

$$E[T_i] = \frac{1}{\theta_i - (\lambda_i + \sum_{j=1}^{N} \theta_j \rho_j P(j,i))}. \quad (40)$$

As for $E[X_i]$, we have $E[T_i] > 0$ since the denominator is a strictly positive value, and the proof is complete.

**Lemma 3.4:** Let $E[T]$ be the average end to end delay of data packets in the EPN network, then:

$$E[T] = \frac{N}{\chi} \left[ \frac{E[X]}{\chi} \right]^{-1}. \quad (41)$$

**Proof:** Let $\chi = \sum_{i=1}^{N} \lambda_i$ be the total arrival rate of data packets incoming from outside the network. Then from Little’s law $E[T] = \frac{E[X]}{\chi}$. Therefore using Lemma 3.2, we obtain (40).

**IV. SOLAR PANEL ASSIGNMENT**

We are now studying an EPN sensor network that collects information in a backbone network. Each sensor gets energy packets from photo-voltaic solar panels, energy flow is stored in sensor’s battery. The aim is to obtain an optimal distribution of $\Delta$ solar panels over the $N$ sensors ($\Delta$ and $N$ being integers). More precisely, we are looking for an assignment of the solar panels that minimizes the average end to end delay of data packets in the network. Also we considered the following constraints: (a) A solar panel could be assigned to only one sensor, but a sensor receives energy from at least one panel. We consider identical panels (i.e all the panels generate the same rate of energy per time unit) (b) Stability constraints as described in Theorem 2.1.

**Lemma 4.1:** Let $\Phi_i$ be the number of solar panels affected to a cell $i$. Then, under EPNS stability conditions of the EPN, for all $i \in \{1,\ldots, N\}$:

$$\Phi_i < \frac{\gamma_i + \mu_{i,1}}{\alpha} \left(1 + \sum_{n=2}^{K} \mu_{i,n} \prod_{r=1}^{n-1} \rho_{i,r} \right)^{-1}. \quad (42)$$

Where $\alpha > 0$ is the rate of energy packets generated by a solar panel (which is supposed to be the same since the panels are identical).

**Proof:** In this lemma, we are interested in the stability condition of EPNs (i.e. for all $i \in \{1,\ldots, N\}$, $\sum_{n=1}^{K} \beta_{i,n} < 1$).

$$\sum_{n=1}^{K} \beta_{i,n} < 1 \Rightarrow \beta_{i,1} + \sum_{n=2}^{K} \beta_{i,n} < 1. \quad (43)$$

By substitution using (2):

$$\Rightarrow \frac{\alpha_i}{\gamma_i + \mu_{i,1}} \left(1 + \sum_{n=2}^{K} \mu_{i,n} \prod_{r=1}^{n-1} \rho_{i,r} \right) < 1. \quad (44)$$

The rate of energy packets coming into a cell’s battery $\alpha_i$ is a function of the number of solar panels assigned to the cell. Then $\alpha_i = \Phi_i \alpha_i$. By the substitution of $\alpha_i$ in (42), we obtain (41). The proof is complete.

Let us now simplify the flow equations due to the topology.

**Lemma 4.2:** Let $i$ be an arbitrary node of the tree, $A(i)$ will denote the set of nodes in the sub-tree rooted at $i$ (including $i$). Then, the flow equations of $DP$ queues (3) can be simplified as:

$$\rho_i = \frac{\sum_{j \in A(i)} \lambda_j}{\sum_{n=1}^{K} (1-p_{j,n})\mu_{i,n}\beta_{i,n}}. \quad (45)$$

**Proof:** By induction on the node number. We assume that the nodes are numbered according to a topological ordering of the tree.

- $i = 1$: node 1 is a leaf because it has no predecessor. Therefore $A(1) = \{1\}$. Thus:

$$\rho_1 = \frac{\lambda_1}{\sum_{n=1}^{K} (1-p_{1,n})\mu_{1,n}\beta_{1,n}},$$

and it is also equal to $\frac{\sum_{j \in A(1)} \lambda_j}{\sum_{n=1}^{K} (1-p_{1,n})\mu_{1,n}\beta_{1,n}}$. Thus the property is established for $i = 1$.

- arbitrary $i$. Let us now assume that the property is established for all $j < i$. Thus we have:

$$\rho_j \sum_{n=1}^{K} (1-p_{j,n})\mu_{j,n}\beta_{j,n} = \sum_{k \in A(j)} \lambda_k.$$

Let $\Gamma^{-}(i)$ be the set of predecessor of $i$. Remember that routing matrix $P$ encodes the tree topology. Therefore $P(j,i) = 1$ if $j$ is a predecessor of $i$ (i.e. in $\Gamma^{-}(i)$) and
0 otherwise. Consider the numerator of the flow equation for DP in station $i$:

$$\lambda_i + \sum_{j \in \Gamma^-(i)} \sum_{n=1}^{K} (1 - p_{j,n}) \mu_{j,n} \beta_{j,n}.$$

Applying the induction, the numerator becomes:

$$\lambda_i + \sum_{j \in \Gamma^-(i)} \sum_{k \in A(j)} \lambda_k.$$

Due to the recursive construction of any tree, we get:

$$\lambda_i + \sum_{j \in \Gamma^-(i)} \sum_{k \in A(j)} \lambda_k = \sum_{j \in A(i)} \lambda_j.$$

And the proof is complete.

\textbf{Lemma 4.3:} The DPs stability condition is $\rho_i < 1$ for all $i$. Then, in a tree EPN, we have for all $i \in \{1, \ldots, N\}$:

$$\Phi_i > \frac{(\gamma_i + \mu_i,1)}{\alpha \mu_{i,1}} \sum_{j \in A(i)} \lambda_j.$$  \hspace{1cm} (44)

\textbf{Proof:} From Lemma 2.1 we have for all $i \in \{1, \ldots, N\}$,

$$\sum_{n=1}^{K} (1 - p_{i,n}) \mu_{i,n} \beta_{i,n} = \alpha_i - \gamma_i \beta_{i,1}$$

and by the substitution of $\beta_{i,1}$ and $\alpha_i$ we obtain:

$$\sum_{n=1}^{K} (1 - p_{i,n}) \mu_{i,n} \beta_{i,n} = \frac{\alpha \Phi_{i,\mu_{i,1}}}{\gamma_i + \mu_{i,1}}.$$  \hspace{1cm} (45)

Using (45) in (43), we get:

$$\rho_i = \frac{(\gamma_i + \mu_i,1)}{\alpha \Phi_{i,\mu_{i,1}}} \sum_{j \in A(i)} \lambda_j.$$  \hspace{1cm} (46)

DPS stability conditions are $\forall i \in \{1, \ldots, N\}$, $\rho_i < 1$. Hence, by combining (46) and that $\rho_i < 1$ we obtain (44), and the proof is complete.

Note that these two constraints may lead to an empty set of solutions for some numerical values of the parameters. Thus the numerical techniques we propose also check if a solution exists.

\textbf{A. Optimization problem}

Let us now study the domain of definition. The set of possible values for $(\Phi_1, \ldots, \Phi_N)$ comes from all the constraints we got (particularly Lemma 4.1, Lemma 4.3), then $\forall i \in \{1, \ldots, N\}$:

$$\left\{ \begin{array}{l} \Phi_i < \gamma_i \mu_{i,1} \left( 1 + \sum_{n=1}^{K} \frac{\mu_{i,n}}{\mu_{i,1}} \prod_{r=1}^{n-1} p_{i,r} \right)^{-1}, \\
\Phi_i > \frac{(\gamma_i + \mu_i,1)}{\alpha \mu_{i,1}} \sum_{j \in A(i)} \lambda_j > 0, \\
\sum_{i=1}^{N} \Phi_i = \Delta. \end{array} \right.$$  \hspace{1cm} (47)

Clearly, this function (i.e. (47)) is separable: $f(\Phi_1, \ldots, \Phi_N) = \sum_{i=1}^{N} f_i(\Phi_i)$. And each function $f_i$ is decreasing with $\Phi_i$. Thus, one must increase each $\Phi_i$ to decrease function $f$.

Therefore, if $\Delta$ is at least equal to the sum of the upper bounds on $\Phi_i$, the optimal solution is the upper bound and some panels are not used. Otherwise, we need a method to allocate the panels to the cells. We use two methods: one based on an heuristic and another on Gradient descent algorithm. Both methods consist in 3 steps. In step 1) we calculate the bounds of the solution domain for each cell. In step 2) we verify that the problem has at least one solution and that $\Delta$ is at least equal to the sum of the lower bound of cells. Also if $\Delta$ is equal to or greater than the sum of the upper bound of the cells, then the solution is the upper bound of the panels in each cell as previously mentioned. Step 3) is an iterative improvement of the solution. We start with the lower bound of panels in each cell (Lemma 4.3). Then at each iteration, we add a panel until we assign all the panels available (i.e. $\Delta$). In the heuristic algorithm, we set the new panel to the most loaded cell which has not reached its upper bound. In the gradient descent approach, the new panel will be assigned to the cell that minimizes (47). Therefore the main difference between the heuristic and the Gradient descent is the decision made in step 3.2). Gradient descent algorithm performs the optimal local decision at each step (but we do not have proof that it is the global optimum).

\textbf{Algorithm 1 Heuristic: Panels assignment algorithm}

**Require:** $\Delta$ the number of panels.

**Ensure:** An assignment of the $\Delta$ panels.

\textbf{Step 1) Calculation of lower and upper bound of $\Phi_i$ for each cell $i$.} (Lemma 4.1 and 4.3).

\textbf{Step 2) Verification of bounds:}

\textbf{Step 2.1) If} there is a cell whose upper bound of the number of panels is lower then the lower bound, then print "No solution for such parameters", the algorithm stops. Otherwise go to Step 2.2)

\textbf{Step 2.2) If} $\Delta$ is strictly lower then the sum of all lower bounds, then print "$\Delta$ is too low", the algorithm stops. Otherwise go to Step 2.3)

\textbf{Step 2.3) If} $\Delta$ is equal or greater than the sum of all upper bounds, then the solution is the upper bound of each cell, the algorithm stops. Otherwise, go to step 3)

\textbf{Step 3) Constructing progressively the solution:}

\textbf{Step 3.1) We} construct a first solution, which corresponds to the lower bound of each cell.

\textbf{Step 3.2) We} add a panel in the most loaded cell that can receive a panel.

\textbf{Step 3.3) If} we achieved $\Delta$ panels in the network, than go to step 4). Otherwise go to Step 3.2)

\textbf{Step 4) Return} the solution and the corresponding delay.

\textbf{B. Numerical example}

We consider a complete binary tree EPN topology with $N = 7$ cells. Cells 1, 2, 3 and 4 are leaves, while cells 5 and 6 are inter-cellular cells and the root cell is 7. All the cells send
Table I

<table>
<thead>
<tr>
<th>Δ</th>
<th>Φ₁</th>
<th>Φ₂</th>
<th>Φ₃</th>
<th>Φ₄</th>
<th>Φ₅</th>
<th>Φ₆</th>
<th>Φ₇</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>21</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>23</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>29</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5 &amp; 4</td>
<td>5 &amp; 4</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>31</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>6 &amp; 5</td>
<td>4 &amp; 5</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>33</td>
<td>2</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>35</td>
<td>2 &amp; 3</td>
<td>4</td>
<td>6 &amp; 5</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>37</td>
<td>3</td>
<td>4</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>3 &amp; 4</td>
<td>6 &amp; 5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the leakage rate is the same for all EP queues: \( \forall i, \gamma_i = 5 \). Service times are distributed according to an Erlang process with \( K = 4 \) phases, (i.e. \( \forall i \in \{1, \ldots, N\} \) \( \mu_{i,n} = \mu_i \) and \( \forall n \in \{1, \ldots, K-1\} \) \( p_{i,n} = 1, \mu_i = 50 \) and \( p_{i,K} = 0 \). The external arrivals of DP only occur to the leaves. Arrival rates of DPs in each cell are \( \{1, 2, 3, 4, 0, 0, 0\} \) per time unit. EPs rate generated by a single solar panel is \( \alpha = 2 \) per time unit and the number of panels is \( \Delta = 41 \). All these parameters lead a set of feasible solutions which is not empty.

Fig. 2 illustrates the evolution of the average end to end delay of a DP in the network while increasing the number of panels \( \Delta \). The average end to end delay was calculated using the solution proposed by the two methods. We first observe that the average end to end delay decreases with the increase in the number of panels in the network (as the number of panels is at the denominator of the function to minimize (47)). We also notice that the solutions proposed by the two methods are quasi similar. In particular, for this example we have 22 experiments (from \( \Delta = 20 \) to \( \Delta = 41 \)) of which 16 experiments generate the same solution. In the 6 other experiments the delay of the heuristic is around \( 10^{-3} \) and \( 10^{-4} \) above the Gradient descent. In Table I we present the solutions proposed by both algorithms for some values of \( \Delta \). In this table, an entry \( 'x1k\times x2' \) means that the heuristic answers \( x1 \) panels while Gradient descent proposes \( x2 \) while a single number (say \( 'x' \)) means that both algorithms assign the same number of panels \( x \) to the corresponding cell. It is worthy to note that both methods are very fast. We have also verified numerically that the gradient descent algorithm generates the optimal solution, i.e. by comparing its solution to an exhaustive algorithm. Since the exhaustive method is exponential in execution time, then we can perform this exhaustive analysis up to \( \Delta = 50 \) panels.

V. Conclusion

We analyse mathematically an EPN using Markov chains and balance equations. We prove a product form solution for the steady-state probability from which we compute end to end delays and energy packet loss rates. In order to optimise end to end delays, we proposed and compare an heuristic and a Gradient descent algorithm. In future work we are aiming to study the end to end delays and energy consumption in a unique function to minimize.

Acknowledgment

Youssef Ait el mahjoub is supported by Labex Digicosme (ANR 11-LABX-0045) PHD grant program (Perfeco project).

References

Concept Drift and Avoiding its Negative Effects in Predictive Modeling of Failures of Electricity Production Units in Power Plants

1st Marek Moleda  
TAURON Wytwarzanie S.A.  
Silesian University of Technology  
Gliwice, Poland  
ORCID: 0000-0002-6556-5083

2nd Alina Momot  
Department of Applied Informatics  
Silesian University of Technology  
Gliwice, Poland  
ORCID: 0000-0002-9308-8077

3rd Dariusz Mrozek  
Department of Applied Informatics  
Silesian University of Technology  
Gliwice, Poland  
ORCID: 0000-0001-6764-6656

Abstract—Ensuring the required accuracy of predictive models operating on time series is very important for industrial diagnostics systems. It is especially visible if there are a lot of models covering hundreds of devices and thousands of measurements operating under varying conditions in changing environments. In this work, we analyze the concept drift phenomenon in the context of actual measurements and predictions of the diagnostic system of boiler feed pump working in coal-fired power plants. In the practical part, we adapt algorithms and techniques operating on time series to obtain better results and reduce the negative effects of the concept drift. The results of our experiments show that the application of drift handling methods brings improvement in the effectiveness of the fault prediction process.

Index Terms—concept drift, predictive maintenance, power plant, time series

I. INTRODUCTION

In the field of machine learning, it can often be noticed that data and its statistical properties change over time, while predictive models usually assume a static relationship between input and output variables. The problem of the changing underlying relationships in the data, known as concept drift [24], causes less accurate predictions and deterioration of predictive performance as time passes. Changes in the hidden context usually induce changes in the target concept, and due to this fact, effective predictive models should be able to detect such changes and adapt to them quickly. Additionally, models should be able to distinguish between actual concept drift and noise. Therefore the ideal predictive models should combine robustness to noise and sensitivity to concept drift [24].

First systems capable of handling concept drift were built in the last century, e.g. STAGGER [20], IB3 [2], FLORA [24], PECS [19], SPLICE [10]. In general, dealing with changing concepts methods are based on instance selection and use time-window over the data stream that moves over recently arrived instances, assuming that the most recent instances are the relevant ones. The decision models use mostly only the instances that are included in the window, and that is why the key difficulty is setting the appropriate window size. This size can be fixed or adaptive, where the size is adjusted to the current extent of concept drift, e.g. “Adaptive Size” [13], ADWIN [4] or FLORA2 [24].

Manufacturing companies, such as power plants, require production infrastructure reliability, which is deprived of unscheduled downtimes. Disturbances in proper plant functioning lead to deterioration in performance, declines in quality, and serious productivity losses. Thus maintenance is a critical element for production effectiveness [5]. It is essential to select an appropriate maintenance strategy that relies on determining the consequences of a failure of every item and determining a suitable strategy for each one [11]. The results of a study of the relationship between maintenance strategies and performance are presented in [21]. Moreover, it is needed to provide the required reliability for production at the lowest cost [23].

In our research, presented in the paper, we have applied the mechanisms and algorithms used in streaming data to detect and eliminate the effects of concept drift. Based on actual production data from the boiler feed pump [16], we compare the results obtained before and after the application of improving algorithms. The production environment in a power plant creates very variable conditions, which makes stable operation of predictive models very difficult [17]. Regular repairs, breakdowns, and faults have a significant impact on the state of the device and, thus, on the values of gathered measurements. In designing a solution in the area of predictive maintenance (e.g., early failure detection task), it is important to be able to distinguish between changes in accuracy caused by failure or gradual degradation of the device (true positives in our model) and those caused by other factors causing a decrease in model efficiency (false positives).
In the last 50 years, the detection of changes and concept drift have attracted much attention. Initially, concept drift was studied in the context of finite samples, with the system STAGGER [20], FLORA family of algorithms [24] and SPLICE system for identifying context changes [10]. A few useful review articles have been published since then, e.g., an overview of incremental data mining model maintenance and change detection under block evolution [8], a survey on change diagnosis algorithms in evolving data streams [1] or a glimpse of concept drift applications [26]. Reviews over 130 high-quality publications in concept drift related research areas are presented in [14].

In recent years, concept drift is mainly analyzed in the context of streamed data. Three active learning strategies for streaming data that explicitly handle concept drift are presented in [25]. The principal methodologies and approaches for detecting concept drift and supporting data-driven decision-making in big streamed data can be found in [15]. A critical review of stream mining with a focus on stream mining challenges, discussing both well established and recent concept drift detection techniques as well as gaps in the existing literature, particularly relating to current evaluative measures and the availability of public datasets for benchmarking is presented in [22].

III. ENVIRONMENT AND OBJECT OF RESEARCH

In this work, we investigate the idea of concept drift in a regression-based failure prediction model for a boiler feed pump. The feed pump is an essential installation in a coal-fired power plant, supplying water to the boiler continuously. In order to supply water to allow the unit to operate under nominal conditions, two pumps must be operated in parallel. To reduce the impact of unforeseen failures the pumps operate in units consisting of three pumps, one of the pumps is used as a backup. Typical malfunctions and service operations during pump operation are oil filter change, cooler cleaning, water and oil leaks, engine overheating, and bedding. To prevent breakdowns, the pump is checked daily and is subject to diagnostic review over a longer period.

The input set for the prediction model used in our research contains raw historical measurements obtained offline from a Power Generation Information Manager (PGIM) system. The PGIM system is a data repository for signals from a Distributed Control System (DCS) used in power plants to control the production process. Data include measurements of temperature, pressure, power, flow, and many other variables from various sensors located on the monitored pumping unit. Due to a large number of measurements and the occurrence of significant signal correlations, we limited our research to four representative signals: water output pressure, motor temperature, air temperature before the motor and the temperature of bearing no.5. The data range covers the period from January 2013 to August 2017 with the sampling period of one minute.

IV. MANAGING CONCEPT DRIFT IN PREDICTIVE ANALYTICS

A. Identification of the concept drift

Concept drift means that the concept about which data is being collected may shift from time to time, each time after some minimum permanence [6]. The influence of external factors and, above all, the time imprint on the characteristics of the system makes changes imminent. Data streams evolving over time make changes in the data distribution and whole input structure. Changes in data can be characterized as changes in the components of this relation [9], [12]. In other terms, the prior probabilities of classes $p(y)$ may change, the class conditional probabilities $p(X|y)$ may change, and as a result, the posterior probabilities of classes $p(y|X)$ may change, affecting the prediction.

Changes in the distribution of data may have various effects on the returned results. Figure 1 shows the different forms the drift concept can take. Changes can occur abruptly, which is characteristic of a sudden change of state. In our case, such changes may occur after repair or replacement of an important part of the water supplying installation. Slow changes over time characterize gradual and incremental drift. In the case of incremental drift, these changes are close to linear, whereas, in the case of gradual drift, the changes take the form of progressive deregulation. Incremental drift can be a symptom of component wear, e.g., increasing bearing heat or radiator efficiency pairs. Similar factors can cause gradual drift. In this case, it is most likely to be influenced by progressive changes in the input or the emergence of new factors not included in the training set. Recurring drift is a typical example of a fault. A change of state indicates an imminent (and ongoing) failure, while a return to the initial state takes place after the failure removal. This case, in a narrower time range, may also indicate anomalies (e.g., sensor damage) or start-up/turn off states.

![Concept drift types](image)

Fig. 1. Concept drift types
B. Analysis of concept drift types on actual data

Analyzing the prediction results for real data obtained using a stationary model trained with data for 2013, we can observe all the above mentioned types of concept drifting. Figures 2, 3, 4 show the results (i.e., mean absolute error in regression) obtained for the model for the period 2014-2017.

Fig. 2. Example of the concept drift in prediction of oil temperature in front of the cooler

Fig. 3. Concept drift in engine temperature prediction

Fig. 4. Concept drift in output pressure prediction

Figure 2 shows the prediction error for bearing temperature. The error increases successively over a long period. In this case, there have indeed been many reports of increased temperature (the temperature of the bearing has remained high for the whole period 2013-2017), and maintenance works for this bearing. These maintenance works were probably the cause of the increasing deviation of the estimated value for a stationary model.

Figures 3 and 4 show the results for engine temperature and water output pressure. In these cases, we can observe sudden temporary increases in errors, which is a clear reflection of the failures recorded during this time. At a later stage, a permanent error is caused by changes caused by repairs. Occasional cases of recurrent drifting can also be caused by interference from faults of other parts.

C. Concept drift handling

To classify the methods of handling concept drift effect, we will use the taxonomy presented in Figure 5. Data management includes methods related to the selection and weighting of input data. One of the mechanisms defined here is the use of time-window over the stream of examples. Detection methods cover the techniques and mechanisms for drift detection. The two main approaches are related to monitoring the evolution of performance indicators and monitoring distributions on two different time-windows. Adaptation methods consist of retraining or refreshing the model to new or more recent data.

1) Adaptation methods: The adaptation methods manage adaptation of the model. We can distinguish two types of approaches: blind and informed.

- Blind methods update the model with new data at regular intervals without considering whether changes have actually occurred (see fig. 6). There are also improvements to the method by giving the data more significant weight to the recent data and methods of adjusting the window size.
- Informed methods trigger model updates only if a change was detected. They work together with the detection model.

2) Adaptive window algorithm: The ADWIN (adaptive sliding window) [3], [4] is an algorithm that we can use to detect changes in stream and also to estimate values using
adaptive size of sliding window. As opposed to the traditional sliding window, ADWIN adjusts the length of the window so that the average partial values inside the exit window do not differ significantly from the average window. As a result of the algorithm, we get a subset of most recent items with consistent average values, which is done by dropping the oldest fragments if their average differs from the rest part of the window. The parameters of the algorithm are confidence value $\delta \in (0, 1)$ and the sequence of stream data $S$, consisting of $x_t \in S$ input at instant $t$. The algorithm uses a sliding window $W$ with the most recent data. The main idea of the algorithm is as follows: whenever two “large enough” subwindows of $W$ exhibit “distinct enough” averages, one can conclude that the corresponding expected values are different, and the older portion of the window is dropped [6]. "Large enough" and "distinct enough" thresholds are defined by the Hoeffding bound statistic, testing whether the average of the two subwindows is larger than $\varepsilon_{cut}$, given by the following formula:

$$\varepsilon_{cut} := \frac{1}{2m} \ln \frac{4|W|}{\delta},$$

where $|W|$ denotes the length of window $W$ and $m$ is the harmonic mean of $|W_0|$ and $|W_1|$ (lengths of two adjacent subwindows), computed as follows:

$$m := \left(\frac{1}{|W_0|} + \frac{1}{|W_1|}\right)^{-1}.$$

The pseudo code of the ADWIN algorithm is shown in Algorithm 1, assuming $\bar{\mu}_W$ denote the average of examples contained within $W$, $n$ is the length of $W$ and $n_0$ and $n_1$ are the lengths of $W_0$ and $W_1$ respectively, such that $n = n_0 + n_1$.

The algorithm can be applied in two ways:

- to detect abrupt changes – ADWIN shrinks its window only if a significant change was detected.
- to estimate value in time series – the predicted value is the moving average value calculated for the time window since the last significant change, without the oldest values with different average.

The ADWIN automatically detects changes and adjusts itself to them. The only parameter we set is the confidence value $\delta$, indicating how accurate should be output results. An example of fitting the length of a window is shown in Figure 7.
Algorithm 2: The modified ADWIN algorithm

\begin{verbatim}
Input : W : input data set; step : subset size;
Set \( i_{\text{max}} := 0 \); \( res_{\text{max}} := 0 \); \( k := \lceil |W|/\text{step} \rceil \);
for \( i \leftarrow 1 \) to \( k \) do
  \( \mu W_0 := \text{average of the range}[1 : i] \);
  \( \mu W_i := \text{average of the range}[i + 1 : k] \);
  compute \( m \);
  if \( |\mu W_0 - \mu W_i| \times m > res_{\text{max}} \) then
    \( i_{\text{max}} := i \);
    \( res_{\text{max}} := |\mu W_0 - \mu W_i| \times m \);
end
Output: \( i_{\text{max}} \times \text{step} \)
\end{verbatim}

D. Drift detection

In this section, we present the techniques and mechanisms of concept drift detection, describing in detail algorithms implemented in research with a description of the changes applied. Many algorithms implement this objective; generally, we can classify them as [7]:

- methods based on sequential analysis,
- methods based on control charts,
- methods based on differences between two distributions,
- heuristic methods.

Drift detection mechanisms are used to detect a change that should be followed by refreshing the model. We could apply this approach with certainty about the nature of the change - whether it is a temporary malfunction or a permanent change in the state of the device.

E. CUSUM algorithm

The cumulative sum algorithm (CUSUM) is an algorithm presented and practiced for some time [18]. It is generally used for change detection. The objective of the algorithm is to raise the alarm when the average input value is significantly higher than zero. The CUSUM input can be any filter residual – in our case, it will be an error in estimating the model of a particular signal. The algorithm is based on the following equations:

\[
g_0 = 0, \\
g_t = \max(0, g_{t-1} + (r_t - v)),
\]

where \( r_t \) is the currently observed value, \( v \) is the allowed magnitude of change and \( t \) is the current time. When the value of \( g_t \) exceeds a fixed threshold of \( \lambda \), the alarm is triggered, and the value of \( g_t \) is set to zero. The formula as presented detects only alarms in a positive direction, and in our experiments, that’s enough. However, if it is necessary to detect negative changes, the operator "max" could be changed to "min". The CUSUM algorithm is not complicated and can successfully operate on data streams. The accuracy of the algorithm depends on the proper setting the parameters \( \lambda \) and \( v \). The set thresholds have an influence on the detection speed and the number of false alarms, with better sensitivity and faster detection being reflected in a higher number of false alarms. In the context of our research, this algorithm can be used to filter out fixed or slowly growing errors (resulting from estimation drifting) and to detect outliers or abrupt changes.

V. RESULTS AND DISCUSSION

A. Stationary vs. sliding window

In the case of regression, the quality of the prediction model is determined by the ability to reproduce values, which can be measured by coefficients such as mean absolute error (MAE), root mean squared error (RMSE) or the coefficient of determination (R2). However, one should not forget to maintain sufficient accuracy to detect anomalies, in the case of systems in the area of predictive maintenance. In the conducted experiment, we compared the results of the failure prediction process obtained for the stationary method and different adaptive methods.

- The stationary model, where data of the training set was taken from the whole year 2013 and the test set contains data of the range 2014-2017.
- The adaptive model with fixed window size, where the 2014-2017 data set was divided into subsets of 160,000 samples each (approximately 40,000 samples correspond to a month of continuous work). In the first variant, the training set included the first 120,000 samples, and the next 40,000 samples were included in the test set (the step was 40,000). In the second variant, we used a smaller training set - the first 40,000 samples, and the test set was equal to the next 120,000 samples (the step was 120,000).
- The adaptive model with adjusted window size, where the algorithm is almost identical to the one with fixed window size, the added function is to reduce the length of the window to the size at which the most significant error change was detected. This will allow us to achieve the maximum set of training set based on the recent data bridge, maintaining a consistent distribution. The algorithm used is shown in the section Algorithm 2. The length of the window (window size) is selected from 5 to 80 thousand samples.

Table I shows the results obtained for the stationary model and table II for the adaptive models mentioned above. Experiments were conducted for the following datasets:

- \( T_{\text{engine}} \) – engine temperature,
- \( T_{\text{air}} \) – air temperature before the engine,
- \( T_{\text{bearing}} \) – bearing temperature,
- \( P_{\text{output}} \) – water pressure on output.

Coefficients allowing to compare the quality of prediction were: mean absolute error (MAE) and root mean squared error (RMSE), where the symbols \( MAE \) and \( RMSE \) describe the parameters calculated for the whole dataset, \( MAE_0 \) and \( RMSE_0 \) – obtained for the part of dataset for which the failure condition was found, \( MAE_1 \) and \( RMSE_1 \) – obtained for the part of dataset for normal operation state. In order to determine the ability to detect faults/anomalies, we labelled the data set with time marks in which service events were
recorded for the device. Then we calculated the \( R_{\text{MAE}} \) and \( R_{\text{RMSE}} \) coefficients, which relate the ratio of MAE and RMSE values obtained for the emergency period to the failure-free period, i.e., \( R_{\text{MAE}} = \frac{\text{MAE}_1}{\text{MAE}_0} \) and \( R_{\text{RMSE}} = \frac{\text{RMSE}_1}{\text{RMSE}_0} \).

Analyzing the obtained results, we can observe that by applying periodic refreshing model with the use of fixed window size, we obtain an overall lower mean absolute error (MAE), while maintaining slightly smaller but still satisfactory coefficients \( R_{\text{MAE}} \) and \( R_{\text{RMSE}} \). Comparing the results for different sliding window variants, we can see that using a shorter step (comparing the 160/120 window to the 160/40 one), we obtained better MAE and RMSE parameters, but significantly worse \( R_{\text{MAE}} \) and \( R_{\text{RMSE}} \) parameters. Using the adaptation step in training set length improves the \( R_{\text{MAE}} \) and \( R_{\text{RMSE}} \) parameters while MAE and RMSE parameters remain at a comparable level in a short step (i.e., window size 160/120).

### TABLE I

**ERROR RATES OBTAINED USING A STATIC MODEL**

<table>
<thead>
<tr>
<th></th>
<th>( T_{\text{engine}} )</th>
<th>( T_{\text{air}} )</th>
<th>( T_{\text{bearing}} )</th>
<th>( T_{\text{output}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAE</td>
<td>0.31</td>
<td>0.34</td>
<td>2.08</td>
<td>0.42</td>
</tr>
<tr>
<td>RMSE</td>
<td>0.43</td>
<td>0.45</td>
<td>2.35</td>
<td>0.54</td>
</tr>
<tr>
<td>MAE(_1)</td>
<td>2.36</td>
<td>1.81</td>
<td>7.40</td>
<td>0.47</td>
</tr>
<tr>
<td>RMSE(_1)</td>
<td>2.98</td>
<td>3.42</td>
<td>8.37</td>
<td>0.84</td>
</tr>
<tr>
<td>MAE(_2)</td>
<td>1.23</td>
<td>0.98</td>
<td>4.48</td>
<td>0.44</td>
</tr>
<tr>
<td>RMSE(_2)</td>
<td>2.02</td>
<td>2.32</td>
<td>5.88</td>
<td>0.69</td>
</tr>
<tr>
<td>( R_{\text{MAE}} )</td>
<td>7.63</td>
<td>5.83</td>
<td>3.55</td>
<td>1.09</td>
</tr>
<tr>
<td>( R_{\text{RMSE}} )</td>
<td>6.91</td>
<td>7.63</td>
<td>3.56</td>
<td>1.56</td>
</tr>
</tbody>
</table>

### TABLE II

**ERROR RATES OBTAINED USING ADAPTIVE MODELS**

**Fixed window size 160/120**

<table>
<thead>
<tr>
<th></th>
<th>( T_{\text{engine}} )</th>
<th>( T_{\text{air}} )</th>
<th>( T_{\text{bearing}} )</th>
<th>( T_{\text{output}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAE</td>
<td>0.28</td>
<td>0.24</td>
<td>0.85</td>
<td>0.25</td>
</tr>
<tr>
<td>RMSE</td>
<td>0.40</td>
<td>0.45</td>
<td>1.16</td>
<td>0.43</td>
</tr>
<tr>
<td>MAE(_1)</td>
<td>0.87</td>
<td>0.74</td>
<td>0.92</td>
<td>0.81</td>
</tr>
<tr>
<td>RMSE(_1)</td>
<td>2.49</td>
<td>2.95</td>
<td>1.53</td>
<td>0.59</td>
</tr>
<tr>
<td>MAE(_2)</td>
<td>0.91</td>
<td>0.86</td>
<td>0.95</td>
<td>0.85</td>
</tr>
<tr>
<td>RMSE(_2)</td>
<td>1.70</td>
<td>2.01</td>
<td>1.34</td>
<td>0.51</td>
</tr>
<tr>
<td>( R_{\text{MAE}} )</td>
<td>5.04</td>
<td>4.40</td>
<td>1.33</td>
<td>1.55</td>
</tr>
<tr>
<td>( R_{\text{RMSE}} )</td>
<td>6.31</td>
<td>7.33</td>
<td>1.32</td>
<td>1.39</td>
</tr>
</tbody>
</table>

**Fixed window size 160/40**

<table>
<thead>
<tr>
<th></th>
<th>( T_{\text{engine}} )</th>
<th>( T_{\text{air}} )</th>
<th>( T_{\text{bearing}} )</th>
<th>( T_{\text{output}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAE</td>
<td>0.33</td>
<td>0.45</td>
<td>1.26</td>
<td>0.29</td>
</tr>
<tr>
<td>RMSE</td>
<td>0.44</td>
<td>0.58</td>
<td>1.61</td>
<td>0.45</td>
</tr>
<tr>
<td>MAE(_1)</td>
<td>2.41</td>
<td>1.35</td>
<td>2.99</td>
<td>1.28</td>
</tr>
<tr>
<td>RMSE(_1)</td>
<td>3.58</td>
<td>2.40</td>
<td>4.41</td>
<td>1.66</td>
</tr>
<tr>
<td>MAE(_2)</td>
<td>1.27</td>
<td>0.86</td>
<td>2.04</td>
<td>0.74</td>
</tr>
<tr>
<td>RMSE(_2)</td>
<td>2.42</td>
<td>1.67</td>
<td>3.19</td>
<td>1.17</td>
</tr>
<tr>
<td>( R_{\text{MAE}} )</td>
<td>7.21</td>
<td>3.02</td>
<td>2.36</td>
<td>4.32</td>
</tr>
<tr>
<td>( R_{\text{RMSE}} )</td>
<td>6.91</td>
<td>7.63</td>
<td>3.56</td>
<td>1.56</td>
</tr>
</tbody>
</table>

**Adjusted window size 160/(5-80)**

<table>
<thead>
<tr>
<th></th>
<th>( T_{\text{engine}} )</th>
<th>( T_{\text{air}} )</th>
<th>( T_{\text{bearing}} )</th>
<th>( T_{\text{output}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAE</td>
<td>0.26</td>
<td>0.25</td>
<td>0.96</td>
<td>0.27</td>
</tr>
<tr>
<td>RMSE</td>
<td>0.35</td>
<td>0.38</td>
<td>1.25</td>
<td>0.41</td>
</tr>
<tr>
<td>MAE(_1)</td>
<td>1.26</td>
<td>1.19</td>
<td>1.66</td>
<td>0.46</td>
</tr>
<tr>
<td>RMSE(_1)</td>
<td>2.33</td>
<td>2.57</td>
<td>2.52</td>
<td>0.78</td>
</tr>
<tr>
<td>MAE(_2)</td>
<td>1.59</td>
<td>1.75</td>
<td>1.93</td>
<td>0.60</td>
</tr>
<tr>
<td>RMSE(_2)</td>
<td>4.87</td>
<td>5.20</td>
<td>1.72</td>
<td>1.72</td>
</tr>
<tr>
<td>( R_{\text{MAE}} )</td>
<td>6.57</td>
<td>6.73</td>
<td>2.02</td>
<td>1.90</td>
</tr>
</tbody>
</table>

### B. CUSUM filtering

The use of the CUSUM algorithm, in the assumptions, should detect changes in the generated results by filtering the constant error value resulting from concept drift. The parameter \( v \) we used for the task comprised the moving average and the standard deviation of two weeks in a sliding window:

\[
v = \text{AVG}_{2 \text{weeks}} + 3 \cdot \text{STD}_{2 \text{weeks}}
\]

The visual result of the algorithm on the example of bearing temperature is shown in Figure 8; in the upper part, there are peaks indicating alarms, while in the lower part, there is the prediction error level. By analyzing the chart, it can be concluded that the use of the CUSUM filter solves the problem of incremental drifting. In order to confirm this theory, the points of alarm occurrence were compared with probable causes, i.e., registered service events during this period. The results are shown in table III.

### TABLE III

**CUSUM ALARMS AND CORRELATION WITH REAL EVENTS**

<table>
<thead>
<tr>
<th>Date</th>
<th>( P_{\text{output}} )</th>
<th>( T_{\text{engine}} )</th>
<th>( T_{\text{air}} )</th>
<th>( T_{\text{bearing}} )</th>
<th>Probable cause</th>
</tr>
</thead>
<tbody>
<tr>
<td>2014-01-23</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Faulty thrust bearing temperature measurement</td>
</tr>
<tr>
<td>2014-03-24</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Leakage between engine and Voith</td>
</tr>
<tr>
<td>2014-06-26</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Unknown</td>
</tr>
<tr>
<td>2014-07-19</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Contaminated oil cooler</td>
</tr>
<tr>
<td>2014-08-11</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Valve leakage</td>
</tr>
<tr>
<td>2014-09-23</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Contaminated oil cooler</td>
</tr>
<tr>
<td>2015-04-25</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Safety valve leakage</td>
</tr>
<tr>
<td>2015-05-05</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Contaminated oil cooler</td>
</tr>
<tr>
<td>2015-08-12</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Unknown</td>
</tr>
<tr>
<td>2015-08-28</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Oil leakage</td>
</tr>
<tr>
<td>2015-10-05</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>Overhaul of electric motors</td>
</tr>
<tr>
<td>2015-11-10</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>high bearing vibrations</td>
</tr>
<tr>
<td>2015-11-26</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>high bearing vibrations</td>
</tr>
<tr>
<td>2015-12-02</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>high bearing vibrations</td>
</tr>
</tbody>
</table>
VI. CONCLUSION

In the experiments described in the article, we showed the application of various techniques of change detection and adaptation of the fault prediction model of electricity production equipment in power plants. The theory and knowledge associated with time series are consistent with the problems we face in the area of predictive maintenance tasks. The use of adaptive methods based on periodical training of the predictive model showed the results of prediction are more accurate but the difference between failure states and regular operation decreases. In addition, there is a risk that slowly increasing changes such as material wear and tear or a decrease in performance over a period of even a few years may be overlooked. The use of the change detection algorithm has produced promising results. Triggered alarms overlap with recorded events, but due to a large number of recorded events, the final assessment requires more analytical engineering input.

REFERENCES

Non-Asymptotic Performance Analysis of
Size-Based Routing Policies

Eitan Bachmat
Department of Computer Science
Ben-Gurion University
Beer-Sheva, Israel, 84105.
ebachmat@cs.bgu.ac.il

Josu Doncel
Department of Mathematics
University of the Basque Country, UPV/EHU
Leioa, Spain. 48940
josu.doncel@ehu.eus

Abstract—We investigate the performance of two size-based routing policies: the Size Interval Task Assignment (SITA) and Task Assignment based on Guessing Size (TAGS). We consider a system with two servers and Bounded Pareto distributed job sizes with tail parameter 1 where the difference between the size of the largest and the smallest job is finite. We show that the ratio between the mean waiting time of TAGS over the mean waiting time of SITA is unbounded when the largest job size is large and the arrival rate times the largest job size is less than one. We provide numerical experiments that show that our theoretical findings extend to Bounded Pareto distributed job sizes with tail parameter different to 1.

Index Terms—Size-based Routing, Parallel Servers, Heavy-tailed distributions.

I. INTRODUCTION

Many existing routing policies to parallel queues are included in the SQ(d) framework, where for each incoming job, \( d \geq 2 \) servers are picked uniformly at random to observe their states and the job is routed to the server in the best state among the observed ones. It is known that the performance of this kind of systems is very good (see [1], [2], [3], [4]), however the author in [5] showed that when the variability of the job size distribution is high this family of policies are not optimal. This is, in fact, the regime where the size-based routing policies outperform the routing policies that belong the SQ(d) family of policies.

In this work, we study two size-based policies: the Size Interval Task Assignment (SITA) [6] and the Task Assignment with Guessing Size (TAGS) [7]. In the former policy, short jobs and long jobs are executed in different servers and, therefore, it is assumed that the size of incoming tasks is known. In the latter policy, all the jobs are executed in one server and, if a job does not end its service before a given deadline, it is stopped and enqueued in the other server, where it starts service from scratch. From the above definitions, it follows clearly that the mean waiting time of jobs in the SITA system is smaller than in the TAGS system. Recently, the authors in [8], [9] show for Bounded Pareto distributed job sizes that, when the difference between the smallest and the largest job size tends to infinity and the total load of the system is less than one, the ratio of the mean waiting time of TAGS over the mean waiting time of SITA is upper bounded by 2.

We consider a system with two servers and we compare the mean waiting time of both systems in a non-asymptotic regime where the difference between the size of the largest and the smallest job is finite. We show that the ratio of the mean waiting time of the TAGS system over that of the SITA system is unbounded for the Bounded Pareto distribution with tail parameter 1 when the largest job size is large and the arrival rate times the largest job size is less than one. Our numerical experiments show that the performance ratio of both systems is unbounded other values of the tail parameter of the Bounded Pareto distribution and it can be large even if the arrival rate is not small.

II. MODEL DESCRIPTION

A. Notation

We consider a system with two servers with equal capacity. We assume that the servers are FCFS queues. Arriving jobs follow a Poisson distribution with rate \( \lambda \). The size of the jobs is given by a sequence of i.i.d. random variables denoted by \( X \).

We denote by \( F(s) = \mathbb{P}[X < s] \) the cumulative distribution function of the job size distribution. We assume \( F(\cdot) \) to be differentiable and we write \( f(s) = \frac{dF(s)}{ds} \). We assume that the size of the smallest job is one and the size of the largest job is \( r > 1 \). The load of the system is denoted by \( \rho \) and to ensure stability we assume that \( \rho < 1 \).

B. The TAGS System

We focus on the TAGS routing (see Figure 1). Let \( s \in [1, r] \). In this policy, all incoming jobs are sent to Server 1. If a job has been served before \( s \) units of time in Server 1, it leaves the system; otherwise, when the execution time equals \( s \), it is stopped and sent to the end of the queue of Server 2, where the execution starts from scratch. Jobs that are executed in Server 2 are always executed until completion.

For a given value \( s \), we denote by \( W_{TAGS}(s) \) the random variable of the waiting time of incoming jobs under TAGS policy. We define as \( s_T \) the value of \( s \) such that the mean waiting time of the TAGS system is minimized, i.e.,

\[
s_T = \arg \min_s \mathbb{E}[W_{TAGS}(s)].
\]
From the above description, it follows that
\[
E[W_{TAGS}(s_T)] = E[W_1^{TAGS}(s_T)] + \left( \int_{s_T}^{r} f(x)dx \right) s_T + \left( \int_{s_T}^{r} f(x)dx \right) E[W_2^{TAGS}(s_T)],
\]
where \( E[W_{TAGS}(s_T)] \) is the mean waiting time of jobs in Server \( i \) when the threshold value is \( s_T \).

C. The SITA System

We focus on the SITA routing (see Figure 2). This policy assumes that the size of incoming jobs is known [10], [11]. Let \( s \in [1, r] \). Under the SITA policy with cutoff \( s \), jobs whose size is smaller than \( s \) are sent to Server 1, whereas jobs whose size is larger than \( s \) to Server 2. The random variable of the waiting time of incoming jobs under the SITA policy with cutoff \( s \) is denoted by \( W_1^{SITA}(s) \). We define as \( s^* \) the value of \( s \) such that the mean waiting time of the SITA system is minimized, i.e.,
\[
s^* = \arg \min_s E[W_1^{SITA}(s)].
\]

The mean waiting time of jobs under the SITA policy with cutoff \( s^* \) is given by
\[
E[W_1^{SITA}(s^*)] = \left( \int_{1}^{s^*} f(x)dx \right) E[W_1^{SITA}(s^*)] + \left( \int_{s^*}^{r} f(x)dx \right) E[W_2^{SITA}(s^*)],
\]
where \( E[W_i^{SITA}(s^*)] \) is the mean waiting time of jobs in Server \( i \).

D. The Bounded Pareto Distribution

We now present the Bounded Pareto distribution. A distribution \( X \) is said to be Bounded Pareto with parameters \( 1, r \) and \( \alpha \) if its density has the following form: if \( 1 \leq x \leq r \), then
\[
f(x) = \frac{\alpha x^{-\alpha-1}}{(1-(1/r)^{\alpha})},
\]
and \( f(x) = 0 \) otherwise. Besides, the cumulative distribution function of the Bounded Pareto distribution is given by the following expression:
\[
F(x) = \begin{cases} 0, & x \leq 1, \\ \frac{1-(1/x)^{\alpha}}{1-(1/r)^{\alpha}}, & 1 \leq x \leq r, \\ 1, & x \geq r. \end{cases}
\]

Besides, when \( \alpha \neq 1 \), we have that the mean of the distribution is given by \( \frac{\alpha}{\alpha-1} \frac{1-1/(r)^{\alpha}}{1-1/(r)^{\alpha}} \) whereas when \( \alpha = 1 \) it coincides with the uniform distribution on the interval \([1, r]\). Besides, the most favorable distribution of SITA and TAGS is given when \( \alpha = 1 \) [8]. This is, indeed, the case we study in the next section.

III. BOUNDED PARETO DISTRIBUTION WITH \( \alpha = 1 \)

We consider the Bounded Pareto distribution with \( \alpha = 1 \) and we aim to compare the mean waiting time of a system with two queues operating under the TAGS routing with that of a system with two queues operating under the SITA routing. In the following result, we provide a lower-bound of the performance of the TAGS system.

Lemma 1. For the Bounded Pareto distribution with \( \alpha = 1 \), if \( \lambda r < 1 \),
\[E[W_{TAGS}(s_T)] > \lambda r.\]

Proof. We aim to study the optimal performance of the TAGS system with two queues. It follows from (1) that
\[
E[W_{TAGS}(s_T)] = E[W_1^{TAGS}(s_T)] + \left( \int_{s_T}^{r} f(x)dx \right) s_T.
\]
For the Bounded Pareto distribution with $\alpha = 1$, we have that
\[
\mathbb{E}[W_1^{\text{TAGS}}(s_T)] + \left( \int_{s_T}^{r} f(x) dx \right) s_T =
\frac{\lambda(s_T - 1)}{2(1 - \rho)(1 - \frac{1}{r})} + \frac{1 - \frac{x_T}{r}}{1 - \frac{1}{r}} \geq
\frac{\lambda(s_T - 1)}{2(1 - \frac{1}{r})} + \frac{1 - \frac{x_T}{r}}{1 - \frac{1}{r}} = s_T(\lambda - \frac{1}{r}) + 2 - 2\lambda
\frac{2(1 - \frac{1}{r})}{2(1 - \frac{1}{r})}.
\]
If $\lambda r < 1$, then \( \frac{x(s - \frac{1}{r}) - 1}{2(1 - \frac{1}{r})} \) decreases with $x$. Thus,
\[
\frac{s_T(\lambda - \frac{1}{r}) + 2 - 2\lambda}{2(1 - \frac{1}{r})} \geq \frac{\lambda(r - 2) + 1}{2(1 - \frac{1}{r})} > \frac{\lambda(r - 2) + \lambda r}{2(1 - \frac{1}{r})} = \lambda r.
\]
\[\square\]

It is important to note that, unlike in the previous work [8], we do not need to assume Poisson arrivals to all the servers in the above result. In the following result, we characterize the mean waiting time of the SITA system.

**Lemma 2.** For the Bounded Pareto distribution with $\alpha = 1$, when $\lambda r < 1$ and $r$ is large,
\[
\mathbb{E}[W_1^{\text{SITA}}(s^*)] \leq \frac{\lambda(\sqrt{r} - 1)^2}{\sqrt{r}(1 - \frac{1}{r})^2}.
\]

**Proof.** We first note that the load of each server is upper bounded by $\rho$. Therefore, for the Bounded Pareto distribution with $\alpha = 1$, we have that
\[
\mathbb{E}[W_1^{\text{SITA}}(s^*)] \leq \frac{\lambda(s^* - 1)}{2(1 - \rho)(1 - (1/r))}
\]
and
\[
\mathbb{E}[W_2^{\text{SITA}}(s^*)] \leq \frac{\lambda(r - s^*)}{2(1 - \rho)(1 - (1/r))},
\]
where $\rho = \lambda \ln(r)$. We now note that,
\[
\lambda r < 1 \iff \lambda \ln(r) < \frac{\ln(r)}{r}.
\]
Since when $r$ is large, $\ln(r)$ tends to zero and $\rho = \lambda \ln(r)$, it follows that $\rho$ tends to zero when $\lambda r < 1$. As a result,
\[
\mathbb{E}[W_1^{\text{SITA}}(s^*)] \leq \frac{\lambda(s^* - 1)}{2(1 - (1/r))}
\]
and
\[
\mathbb{E}[W_2^{\text{SITA}}(s^*)] \leq \frac{\lambda(r - s^*)}{2(1 - (1/r))}.
\]

We know from Section 3.2.4 of [13] that, for the Bounded Pareto distribution with $\alpha = 1$, $s^*$ balances the load of both servers and, therefore, it can be obtained as follows:
\[
\int_{1}^{s^*} f(x) dx = \int_{s^*}^{r} f(x) dx \iff s^* = \sqrt{r}.
\]
As a result,
\[
\mathbb{E}[W_1^{\text{SITA}}(s^*)] \leq \frac{\lambda(\sqrt{r} - 1)}{2(1 - (1/r))}
\]
and
\[
\mathbb{E}[W_2^{\text{SITA}}(s^*)] \leq \frac{\lambda(r - \sqrt{r})}{2(1 - (1/r))}.
\]
Therefore, from (2), it follows that
\[
\mathbb{E}[W_1^{\text{SITA}}(s^*)] \leq \left( \int_{1}^{s^*} f(x) dx \right) \frac{\lambda(\sqrt{r} - 1)}{2(1 - (1/r))}
+ \left( \int_{s^*}^{r} f(x) dx \right) \frac{\lambda(r - \sqrt{r})}{2(1 - (1/r))}
= \left( \int_{1}^{\sqrt{r}} f(x) dx \right) \frac{\lambda(\sqrt{r} - 1)}{2(1 - (1/r))}
+ \left( \int_{\sqrt{r}}^{r} f(x) dx \right) \frac{\lambda(r - \sqrt{r})}{2(1 - (1/r))}
= \frac{\lambda(1 - \frac{1}{\sqrt{r}})(\sqrt{r} - 1)}{2(1 - (1/r))^2}
+ \frac{\lambda(\frac{1}{\sqrt{r}} - \frac{1}{r})(r - \sqrt{r})}{2(1 - (1/r))^2}
= \frac{(\sqrt{r} - 1)^2}{\sqrt{r}(1 - (1/r))^2},
\]
From the above lemmas, we have that when $\lambda r < 1$ and $r$ is large, the ratio between the mean waiting time of TAGS over the mean waiting time of SITA is upper bounded by
\[
\frac{\lambda r}{\sqrt{r}(1 - (1/r))^2} = \frac{(\sqrt{r} + 1)^2}{\sqrt{r}}
\]
where the equality is obtained by simplifying the derived expression. Interestingly, the last expression does not depend on $\lambda$. Besides, we now show that it is increasing with $r$.

**Lemma 3.** The function \( \frac{(\sqrt{r} + 1)^2}{\sqrt{r}} \) is increasing with $r$.

**Proof.** We aim to show that \( \frac{(\sqrt{r} + 1)^2}{\sqrt{r}} \) is increasing with $r$, which is true if and only if
\[
((\sqrt{r} + 1)^2)' \sqrt{r} - (\sqrt{r})'(\sqrt{r} + 1)^2 > 0 \iff 2(\sqrt{r} + 1) \sqrt{r} - \frac{1}{2\sqrt{r}}(\sqrt{r} + 1)^2 > 0 \iff \sqrt{r} + 1 - \frac{1}{2\sqrt{r}}(\sqrt{r} + 1)^2 > 0 \iff 1 - \frac{1}{2\sqrt{r}}(\sqrt{r} + 1) > 0 \iff 2\sqrt{r} - (\sqrt{r} + 1) > 0 \iff \sqrt{r} - 1 > 0
\]
\[\square\]
From the above results and using that \((\sqrt{\pi}+1)^2\) tends to infinity when \(r \to \infty\), it follows that when \(\lambda r < 1\), the ratio between the mean waiting time of the TAGS system and the mean waiting time of the SITA system is lower bounded by a function that is unbounded and the following result follows.

**Theorem 4.** The ratio of the mean waiting time of the TAGS system and the mean waiting time of the SITA system is unbounded.

## IV. Numerical Experiments

We present our numerical work, which focuses on the ratio between the mean waiting time of the TAGS system and the mean waiting time of the SITA system for the Bounded Pareto distribution. We investigate the evolution of this ratio when we vary \(r\) from 10 to 1000 for different values of \(\lambda\) and different values of \(\alpha\). In Figure 3, we consider that \(\alpha = 1\) and we depict the ratio between the mean waiting time of the TAGS system and the mean waiting time of the SITA system as a function of \(r\) for different values of \(\lambda\). We observe that the maximum of this ratio is 4 when \(\lambda = 0.05\), 8 when \(\lambda = 0.01\), 11 when \(\lambda = 0.005\) and when \(\lambda = 0.001\) the performance ratio is increasing with \(r\) for the considered values of \(r\). This illustration shows that the maximum over \(r\) of the performance ratio under consideration increases when we decrease \(\lambda\) and also that it can be large even if the value of the arrival rate is not very small. In Figure 4, we consider an arrival rate of 0.001 and we study the evolution of the ratio between the mean waiting time of the TAGS system and the mean waiting time of the SITA system over \(r\) for several values of \(\alpha\) larger than one. We observe that the performance ratio under study is increasing with \(r\) for the considered values of \(r\). Similar results have been obtained for this performance ratio for different values of \(\alpha\) which are smaller than one, that is, ratio between the mean waiting time of TAGS and the mean waiting time of SITA is also increasing with \(r\) when \(\alpha\) is smaller than 1. We omit this illustration due to lack of space.

**REFERENCES**


