# Table of Contents

Message from the General and Program Co-Chairs .............................................. iv
Organizing Committee ................................................................................................ v
Program Committee ................................................................................................. vi
Additional Reviewers ............................................................................................... vii
Sponsors .................................................................................................................... ix

## 1 Session 1

Deep Learning Models for Automated Identification of Scheduling Policies ........ 1
  *Yichong Chen and Giuliano Casale*

S4BXI: the MPI-ready Portals 4 Simulator .............................................................. 9
  *Julien Emmanuel, Matthieu Moy, Ludovic Henrio and Grégoire Pichon*

Energy-Efficiency Comparison of Common Sorting Algorithms ........................... 17
  *Norbert Schmitt, Supriya Kamthania, Nishant Rawtani, Luis Mendoza, Klaus-Dieter Lange and Samuel Kounev*

An "A+" Heuristic for Dispatching in Large-Scale Systems with Unknown Server Speeds . 25
  *Cole Stephens and Kristen Gardner*

OLTP In Real Life: A Large-scale Study of Database Behavior in Modern Online Retail . 33
  *Nosayba El-Sayed, Zhuoran Sun, Ke Sun and Ricardo Mayerhofer*

## 2 Session 2

Scaling Up The Performance of Distributed Key-Value Stores With In-Switch Coordination .............................................................. 41
  *Hebatalla Eldakiky and David Hung-Chang Du*

Simulation Modeling of Urban E-Scooter Mobility .................................................. 49
  *Rachel Mclean, Carey Williamson and Lina Kattan*
3 Session 3

Precomputed Ionospheric Propagation for HF Wireless Sensor Transmission Scheduling... 81
   Terry Koziniec, David Murray and Michael Dizon

Efficient Brain-Inspired Hyperdimensional Learning with Spatiotemporal Structured Data 89
   Jiseung Kim, Hyunse Lee, Mohsen Imani and Yeseong Kim

Sparse Matrix-Vector Multiplication Cache Performance Evaluation and Design
   Exploration. ................................................................. 97
   Javen Cui, Liu Sheng and Kai Lu

A Mixture Density Network Approach to Predicting Response Times in Layered Systems. 104
   Zifeng Niu and Giuliano Casale

Automated performance prediction of microservice applications using simulation......... 112
   Clement Courageux-Sudan, Anne-Cécile Orgerie and Martin Quinson

4 Session 4

Performance Evaluation of Automated Tape Library Systems. ........................... 120
   Ilias Iliadis, Linus Jordan, Mark Lantz and Slavisa Sarafjanovic

Does XORing Pointers Save Bitflips for NVRAM. ......................................... 128
   Arockia David Roy Kandalai and Thomas Schwarz

Understanding Energy Efficiency of Databases on Single Board Computers for Edge
   Computing. ....................................................................... 134
   Jian Liu, Kefei Wang and Feng Chen

A Multiple Snapshot Attack on Deniable Storage Systems. .......................... 142
   Kyle Fredrickson, Austen Barker and Darrell Long

Towards Efficient I/O Scheduling for Collaborative Multi-Level Checkpointing. .... 150
   Avinash Maurya, Bogdan Nicolae, M. Mustafa Rafique, Thierry Tonellot and Franck
   Cappello
5 Session 5

A Case for SDN-based Network Virtualization ...................................................... 158
Gyeongsik Yang, Changyong Shin, Yeonho Yoo and Chuck Yoo

REBAL : Channel Balancing for Payment Channel Networks .............................. 166
Nitin Awathare, Suraj Suraj, Akash Akash, Vinay Ribeiro and Umesh Bellur

Exact and Efficient Protective Jamming in SINR-based Wireless Networks ........... 174
Dominik Bojko, Marek Klonowski, Dariusz R. Kowalski and Mateusz Marciniak

Diffusion Analysis Improves Scalability of IoT Networks to Mitigate the Massive Access Problem .......................................................... 182
Erol Gelenbe, Mert Nakip, Dariusz Marek and Tadeusz Czachorski

Capri: Achieving Predictable Performance in Cloud Spot Markets .................... 190
Bogdan Ghit and Asser Tantawi

Characterizing Machine Learning I/O Workloads on Leadership Scale HPC Systems ... 198
Arnab K. Paul, Ahmad Maroof Karimi and Feiyi Wang
It is our honor to welcome all the participants to the 29th International Symposium on the Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2021). MASCOTS brings together researchers from academia and industry to advance the state-of-the-art in the areas of performance evaluation of computer systems and networks as well as in related areas. Experimental, modeling and simulation studies are all within the scope of the conference, including work focusing on novel performance evaluation methods or providing insights on design and runtime management tradeoffs.

This year we received 76 submissions and accepted 25 papers (32.9% acceptance rate). For the second time, MASCOTS will be conducted in a fully virtual mode this year due to the COVID-19 pandemic. Despite this, we were able to assemble a very exciting program.

While the pandemic forces us to conduct MASCOTS as an online event, this also presents great opportunities to bring together researchers worldwide while bypassing travel restrictions and reducing the cost of participation.

We thank the authors who have submitted their work to MASCOTS and extend our congratulations to the authors of accepted papers. We also extend our gratitude to the program committee members and the reviewers. The success of this year's MASCOTS would not be possible without their hard work and the focus on maintaining a high standard for the technical program. Finally, we also express sincere thanks to our co-organizing committee members:

- Tadeusz Czachorski (Finance Chair)
- Ricardo Lent (Publication Chair)
- Krzysztof Grochla (Publicity Chair)

November 3, 2021
Houston, Texas, USA

Albert Cheng, General Chair
Jehan-François Paris and Erol Gelenbe, Program Co-Chairs
Organizing Committee
MASCOTS 2021

General Chair
Albert Cheng, University of Houston

Program Co-Chairs
Jehan-François Paris, University of Houston
Erol Gelenbe, IITIS, Polish Academy of Sciences

Publicity Chair
Krysztof Grochla, IITIS, Polish Academy of Sciences

Publication Chair
Ricardo Lent, University of Houston

Finance Chair
Tadeusz Czachórski, IITIS, Polish Academy of Sciences
<table>
<thead>
<tr>
<th>Name</th>
<th>Affiliation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jonatha Anselmi</td>
<td>INRIA</td>
</tr>
<tr>
<td>Gilles Bernot</td>
<td>University of Nice Sophia Antipolis</td>
</tr>
<tr>
<td>Deepavali Bhagwat</td>
<td>IBM</td>
</tr>
<tr>
<td>Louiza Bouallouche-Medjkoune</td>
<td>Research Unit LAMOS, University of Bejaia</td>
</tr>
<tr>
<td>Jalil Boukhobza</td>
<td>Univ. Bretagne Occidentale</td>
</tr>
<tr>
<td>Andre Brinkmann</td>
<td>Johannes Gutenberg-Universität Mainz</td>
</tr>
<tr>
<td>Maria-Carla Calzarossa</td>
<td>Universita di Pavia</td>
</tr>
<tr>
<td>Valeria Cardellini</td>
<td>Universita degli Studi di Roma &quot;Tor Vergata&quot;</td>
</tr>
<tr>
<td>Albert Cheng</td>
<td>University of Houston</td>
</tr>
<tr>
<td>Tadeusz Czachorski</td>
<td>ITIIS PAN, Polish Academy of Sciences</td>
</tr>
<tr>
<td>Sofiene Djahel</td>
<td>Manchester Metropolitan University, UK</td>
</tr>
<tr>
<td>Joanna Domanska</td>
<td>Institute of Theoretical and Applied Informatics</td>
</tr>
<tr>
<td>Lorenzo Donatiello</td>
<td>University of Bologna</td>
</tr>
<tr>
<td>Derek Eager</td>
<td>University of Saskatchewan</td>
</tr>
<tr>
<td>Edgar Gabriel</td>
<td>University of Houston</td>
</tr>
<tr>
<td>Erol Gelenbe</td>
<td>ITIIS</td>
</tr>
<tr>
<td>Krzysztof Grochla</td>
<td>Institute of Theoretical and Applied Informatics of PAS</td>
</tr>
<tr>
<td>Yassine Hadjadj-Aoul</td>
<td>University of Rennes 1</td>
</tr>
<tr>
<td>Khaled Hussain</td>
<td>Assiut University</td>
</tr>
<tr>
<td>Ilias Iliadis</td>
<td>IBM Zurich Research Laboratory</td>
</tr>
<tr>
<td>Alain Jean-Marie</td>
<td>INRIA</td>
</tr>
<tr>
<td>Rainer Keller</td>
<td>HS Esslingen</td>
</tr>
<tr>
<td>Samuel Kounev</td>
<td>University of Wuerzburg</td>
</tr>
<tr>
<td>Ricardo Lent</td>
<td>University of Houston</td>
</tr>
<tr>
<td>Darrell Long</td>
<td>University of California</td>
</tr>
<tr>
<td>Aniket Mahanti</td>
<td>The University of Auckland</td>
</tr>
<tr>
<td>Luisa Massari</td>
<td>University of Pavia</td>
</tr>
<tr>
<td>Daniel Menasco</td>
<td>George Mason University</td>
</tr>
<tr>
<td>Jehan-Francois Paris</td>
<td>University of Houston</td>
</tr>
<tr>
<td>Nihal Pekergin</td>
<td>Univ. Paris-Est-Creteil</td>
</tr>
<tr>
<td>Riccardo Pincirolli</td>
<td>Gran Sasso Science Institute</td>
</tr>
<tr>
<td>Gerardo Rubino</td>
<td>INRIA</td>
</tr>
<tr>
<td>Thomas Schwarz</td>
<td>Marquette University</td>
</tr>
<tr>
<td>Daniele Tessera</td>
<td>Universita Cattolica del Sacro Cuore</td>
</tr>
<tr>
<td>Benny Van Houdt</td>
<td>University of Antwerp</td>
</tr>
<tr>
<td>Peter Varman</td>
<td>Rice University</td>
</tr>
<tr>
<td>Marco Vieira</td>
<td>University of Coimbra</td>
</tr>
<tr>
<td>Umberto Villano</td>
<td>University of Sannio</td>
</tr>
<tr>
<td>Carey Williamson</td>
<td>University of Calgary</td>
</tr>
</tbody>
</table>
Additional Reviewers
MASCOTS 2021

Abdelhaf ez, Amr
Abdelham ed, Abdelrahman
Afifi, Mahmoud

Bar bierato, Enrico
Bel, O ceane
Benhamida, Nadjette
Bononi, Luciano
Bounouni, Mahdi

Campos, João R.
Catillo, Marta

Dar wish, Mohamed
Davoli, Renzo
D’abruzzo Pereira, José

Eismann, Simon

Ferretti, Stefano
Fonseca, José

Goncal ves, Charles
Grohmann, Johannes

Herrnleben, Stefan

Kaiser, Dennis
Kalouptsoglou, Llias

Leppich, Robert

Marzolla, Moreno
Mohammed, Mostafa
Moktefi, Mohand

Nakip, Mert

Polys, Konrad

Solorio, Thamar
Straesser, Martin
Taj-Eddin, Islam
Velusamy, Gandhimathi
Verma, Rakesh
Wang, Lan
Sponsors
MASCOTS 2021

Technically Co-Sponsored by the IEEE Computer Society

IEEE
COMPUTER
SOCIETY

IITiS
Deep Learning Models for Automated Identification of Scheduling Policies

Yichong Chen
Imperial College London, UK
yichong.chen119@imperial.ac.uk

Giuliano Casale
Imperial College London, UK
g.casale@imperial.ac.uk

Abstract—Queueing network models are commonly used as performance models of distributed software applications and service-based systems. Although several methods exist for learning their parameters, such as demand estimation methods, little research has been carried out in the literature on automatically identifying scheduling policies from empirical datasets. Scheduling policies and their parameters have an impact on the model’s stationary distribution in general, thus their correct determination is important for model accuracy. They are particularly relevant for correctly estimating percentiles and higher-order moments of performance indexes such as response times. We propose a deep learning technique based on transformer models - a common technique in natural language processing, to address the lack of methods for this parameter identification problem. From a sample path of the joint network state, or an aggregate thereof, our approach can classify the scheduling policy of the stations in a queueing network. We show that the transformer model delivers good-classification precision and recall, improving significantly over support vector machines or simpler recurrent neural networks.

I. INTRODUCTION

Queueing network models are a type of stochastic model that is commonly used to simulate congestion in software applications and service-based systems [1]. The identification of artificial intelligence and machine learning (AI/ML) methods that can help to automatically identify a model from empirical data is a natural question that arises in conjunction with the current growth of AI technology.

Over the last decade, a comprehensive range of studies have investigated estimating service demands, which are important drivers of model prediction accuracy [2] and one of the main challenges in the performance model identification field. For example, in [3], the authors apply maximum likelihood estimation to predict demands in closed queueing networks. Similarly, the work in [4] tackles the same problem via Gibbs sampling and Monte Carlo integration [5]. Prior art in parameter estimation also includes open models, such as [6] that estimates service and arrival rates limiting distribution based on the number of jobs in the node. AI/ML are also used in some prior arts. For example, in [7], the authors apply a fluid approximation to the queueing network [8] to model the system dynamics and use a recurrent neural network to predict the queue length and encode the transition probability matrix and the service rate of the queueing network. The work in [9] predict the waiting time of jobs in a queueing system with a Gaussian mixture model in which parameters are generated from a fully-connected neural network. The applicability of these methods often depends on the precise assumptions of the scheduling policy at play, making it critical to identify the scheduling policy in use inside a system with high certainty. For example, [10] proposes various algorithms for estimating missing data inside a finite sample path of a queueing network, depending on the scheduling policy. Therefore, the application of this body of work on demand estimation effectively requires to know the scheduling policy at every node before applying the proposed algorithm to the dataset, motivating the use of specialized methods to learn such scheduling characteristics from measured data.

The widespread diffusion of cloud-hosted services makes model classification useful for infrastructure providers to estimate the topology and resource consumption of an application from passive measurements (e.g., network packet sniffing), is another motivator for the scheduling identification problem we consider. Learning scheduling policies in this setting may in principle help in better understanding the scheduling characteristics of a distributed system, supporting performance prediction.

Motivated by the above considerations, this paper proposes a transformer model [11] to automatically identifying the scheduling policies of a network of resources, modeled as queueing stations, assuming that partial measurements are obtained for their state. To the best of our knowledge, the present work is the first one to develop an AI method for learning scheduling characteristics from partial monitoring data and accounting for the presence of queueing phenomena. We show that transformer models, commonly used in natural language translation problems, can be effectively adapted to the queueing model identification problem under study. Experiments with simulation data show that it is possible to recover with relatively good accuracy the scheduling characteristics of the system that generated the data using a supervised learning approach. The proposed method can result in accuracy and F1-score that both exceed 0.80. Satisfactory accuracy is observed also in sensitivity analysis experiments, where we reduce the amount of information available to the method, or include job priorities in the model.

The paper is organized as follows: Section 2 introduces the classes of models studied throughout this paper. Section 3 overviews certain AI/ML models applicable to the problem at hand. Section 4 introduces the proposed transformer model.
Table I

<table>
<thead>
<tr>
<th>Scheduling</th>
<th>State vector</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCFS, LCFS, HOL</td>
<td>((c_1, \ldots, c_R, s_1, \ldots, s_R)) (\sum_s s_r = 1)</td>
<td></td>
</tr>
<tr>
<td>PS</td>
<td>((s_1, \ldots, s_R))</td>
<td>None</td>
</tr>
<tr>
<td>SIRO</td>
<td>((b_1, \ldots, b_R, s_1, \ldots, s_R)) (\sum_s s_r = 1)</td>
<td></td>
</tr>
</tbody>
</table>

Section 5 gives experimental results, followed by conclusions.

II. Background

Queueing network models have been extensively studied in the scientific literature and applied to many computer and communication systems [12]. According to certain statistical distribution, service times have a random duration. Jobs may either visit the system and then leave (open classes), or perpetually cycle in the network (closed classes). Routing is probabilistic and governed by a discrete-time Markov chain.

At each station, a scheduling policy determines which job will be selected for processing when a server is idle. Some of the most commonly used scheduling policies include first-come first-served (FCFS), last-come first served (LCFS), processor sharing (PS), and service in random order (SIRO) [1]. If jobs are assigned a static priority, the FCFS rule is typically replaced by the Head-of-Line (HOL) policy, in which the jobs with the highest priority are served first and then in order of arrival within that group.

Let us consider a model with \(M\) stations and where jobs can belong to one of \(R\) service classes. Class switching is not allowed and service times are assumed to be exponentially distributed. When we represent such systems as a continuous-time Markov chain, the state vector for a given single-server station can be represented as in Table I, based on the following symbols:

- \(b\): number of jobs in the waiting buffer in the current state.
- \(c_r\): class of job waiting at the \(j\)th position in the waiting buffer.
- \(n_r\): Number of class-\(r\) jobs in the station.
- \(b_r\): Number of class-\(r\) jobs in the buffer.
- \(s_r\): Number of class \(r\) jobs running in the server.

It would be fairly simply to classify scheduling by observing the random variables shown in Table I. The problem we consider here is however to do so by observing aggregate state vectors

\[
x(t) = (x_{11}(t), x_{12}(t), \ldots, x_{1R}(t), \ldots, x_{MR}(t))
\]

where \(x_{ij}(t)\) is the random variable denoting the number of jobs in class \(j\) at station \(i\) at the measurement time \(t\). The sample path of aggregate states has a more compact representation than using the exact one in Table I. It is also easier to collect and monitor in real-world systems in practice, since it does not require to track the state of individual jobs, but rather how any jobs for each class are inside the system. Thus, it is a quantity that can be estimated by input-output measurements of the system. The problem we consider here is to classify scheduling policies based on the finite realization.

III. Sequence Classification Methods

Classifying the scheduling policy in a single-server station is easy if the state space is small and the station states can be continuously observed. However, in many real systems, such as in a web server, hundreds of thousands of requests received in a matter of seconds, making precise tracking of the system state costly. Indeed, tracing methods for distributed systems are forced for this very same reason to adopt sampling methods. Besides, the system may leverage resources across a cluster and thus require models with multiple queueing stations. Therefore, collecting all the states at every node in a reliable fashion becomes more difficult. Furthermore, some data may be absent, further complicating the process.

Secondly, in the presence of aggregation, it is difficult to discern distinct traits that distinguish policies from sample paths. In our experiments, we discovered that samples paths across policies appear appear qualitatively comparable when inspected visually. Faced with this challenge, we propose the use of AI/ML methods in solving the model identification problem.

In this section, we introduce three possible architectures that may be used for this problem, leveraging SVMs [13], RNNs [14], and transformers [11]. As our experimental validation later demonstrates, the transformer outperforms the other two techniques, we expand on this in Section 4, since it is relatively little understood within the performance evaluation community.

A. Support Vector Machines

The support vector machine (SVM) is a well-established machine learning model introduced in [13]. The SVM draws a linear decision boundary which maximize the gap between two classes so that to separate two classes. The model may fail if the data are not linearly separable. We can address this by projecting the data to a higher dimension where the data are linearly separable with a chosen kernel function. The radial basis function (RBF) is commonly used in classification problems and also used throughout the present work.

To apply SVM to time series data, we consider the states of a station throughout a period as input data, which may be seen as a matrix of size \(n \times m\) where \(n\) is the sequence length and \(m\) is the dimension of the aggregate state vector. This matrix is transformed to an input vector of size \(nm\). Thus, SVMs can be applied to time series data by increasing the number of input features.

B. Recurrent Neural Networks

In deep learning (DL) and natural language processing (NLP), recurrent neural networks (RNN) are often used to
The transformer model [11] has made outstanding achievements on solving NLP problems like machine translation [15] and language model construction [16]. To apply this model to the problem at hand, we introduce a modified transformer architecture as shown in Figure 1; details of this architecture are described later in the next section.

IV. BACKGROUND ON TRANSFORMER MODEL ARCHITECTURE

A. Overview

The architecture seen in Figure 1 is described in detail in this section. The meaning of the blocks in the model architecture is as follows. The transformer model is an encoder-decoder model. The encoder maps the state sequence to a new sequence of representations that is passed as one of the input of the decoder. The decoder then generates the output with \( n_{\text{query}} \) station queries that identify the scheduling characteristics. These queries are initially zero vectors that are modified by the decoder as they pass through the blocks shown in Figure 1. Therein, the positional encoding blocks embed the order of the sequence and the multi-head attention blocks process the correlation between each states within a sequence. Finally, using a linear feed-forward network and the softmax function, the data is processed so that the output of the model represents the probability of a given scheduling policy being used at each station.

B. Attention

The attention mechanism is the crucial part of the transformer model. We here review the essential concepts of the application of the method in [11] to the queueing domain. This mechanism can process the relation between any two states in the sequence, and its output is a vector \( a \in \mathbb{R}^{n_x d_v} \), where \( d_v \) is a parameter denoting the dimension of the key vectors. This layer is defined on the attention function, which is a function that delivers a high contribution to the output of the corresponding query if two states are highly correlated, and vice-versa if their correlation is negligible. The attention function takes three inputs, which are queries and key-value pairs, to produce the output. The contents of the key-value pairs are described in the next sub-sections, we here focus on the definition of attention and for the moment these may be treated as arbitrary.

In more details, suppose the queries \( Q \) and keys \( K \) are in \( \mathbb{R}^{n_x d_k} \), and the values \( V \) is in \( \mathbb{R}^{n_x d_v} \), where \( n \) is the number of states in the trace. The equation for calculating the output of the attention function is as follows

\[
a = a(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V.
\]

From this we can compute the correlation between the query and the key. More precisely, this is used in the transformed model to compute the correlation between any two states \( x_i \) and \( x_j \).

In our model, each layer contains multiple attention functions. Each attention function is referred to as attention head and it concentrates on different features of the sequence. These attention heads form the multi-head attention layer which is written as follows [11],

\[
\text{MH}(Q, K, V) = \text{Concat}(a_1, \ldots, a_h)W^Q
\]

\[
a_i = A(QW_{i}^Q, KW_{i}^K, VW_{i}^V), \quad i = 1, \ldots, h.
\]

That is, in order to have different attentions, we project the queries and key-value pairs linearly for \( h \) times with learnable parameter matrices \( W_{i}^Q \in \mathbb{R}^{d_{model} \times d_k} \), \( W_{i}^K \in \mathbb{R}^{d_{model} \times d_k} \) and \( W_{i}^V \in \mathbb{R}^{d_{model} \times d_v} \), where \( d_k = d_v = d_{model}/h \), so that the projected queries and key-values pairs contain different
information that the attention function should focus on. All the attention outputs are concatenated together. To be more specific, if we take the attention outputs $a_i \in \mathbb{R}^{n \times d_e}$, all the matrices are stacked together to form a large matrix with more columns so that the shape of the concatenated matrix is $(n, bd_e)$. The output is again projected with $W^O \in \mathbb{R}^{bd_e \times d_{model}}$ to process the output of different attention heads and get the final result.

C. Transformer Encoder-Decoder

Suppose the input state sequence is $x \in \mathbb{R}^{n \times d}$ where $n$ is the number of states, and $d$ is the dimension of a single state. The input is first projected to a higher dimension with a feed-forward network (FFN), so that $x' \in \mathbb{R}^{n \times d_{model}}$. In our experiment, adding this FFN can significantly improve the performance of the model.

Unlike the RNN model which inherently takes sequence order into account, the encoder and decoder of the transformer model only contain the multi-head attention and the FFN, neither of these networks takes sequence order into account. As a result, a positional encoding $P$ is added to the input to embed the order of the sequence. In [11], the sine and cosine functions are used as the positional encoding. The functions are defined as follow

$$P_{t,2i} = \sin \left( \frac{t \cdot 2i}{10000^{\frac{2i}{d_{model}}} } \right)$$
$$P_{t,2i+1} = \cos \left( \frac{t \cdot 2i+1}{10000^{\frac{2i}{d_{model}}} } \right),$$

where $t = 0, 1, \ldots, (n-1)$. The positional encoding matrix $P$ determined not by the input values but by the $d_{model}$ and the position. Therefore, the order of a state can be represented by a unique vector $P_t$ and add to the projected state vector $x'_t$ (the row vector of $x'$), so that the input contains the information of the order.

In the encoder, the encoded state vectors become the queries and key-value pairs of the multi-head attention network. The output of the multi-head attention network is added with the input of the attention network, so that the loss can pass through the attention network or directly back-propagate to the previous layer [17]. Finally, in order to accelerate the training and minimize the covariate shift, we apply batch normalization to the resulted output [18].

In machine translation, the decoder of the transformer model takes the output of the encoder and the output of the previous prediction of the model. However, in our model, we classify the scheduling policy of all the stations simultaneously. To address this problem we use a matrix $z$ of zeros with shape $(n_{query}, d_{model})$ as one of the inputs of the decoder. Since the model produces $n_{query}$ different predictions, the input must be different before sending it into the decoder. As a result, a learned positional encoding is applied, and the input is combined with the positional encoding, just like in the encoder.

In the decoder, the multi-head attention network takes the output of the previous layer as the query and the output of the encoder as the key-value pairs so that the network considers both the input of the decoder and the output of the encoder.

D. Prediction FFN and loss function

In the final step, the prediction is computed with an FFN with softmax as the activation function. By using the softmax function, the probability of each scheduling policy used in the station can be computed.

To measure the performance of the model, we use cross-entropy (7) as the loss function. As we mention above, we set $M \leq n_{query}$ and use the "empty" label to represent the unused stations. The amount of "empty" labels can be larger than the other scheduling policies, so the model is easier to learn to classify the "empty" label. However, we hope that the model can also perform well on other policies. As a result, the loss of each class needed to be weighted. The empty class should have a significantly lower weight than the others. So the loss function is

$$\text{Loss}(\text{output}, y) = - \sum_{i=1}^{N} \sum_{j=1}^{C} \text{Weight}(y_{ij})(y_{ij} \log(\text{output}_{ij}))$$

where $y_{ij}$ is the true scheduling policy of sample $i$ at station $j$ and output$_{ij}$ is the corresponding predicted probability of using policy $y_{ij}$. Weight($y_{ij}$) is the weight of policy $y_{ij}$. We need to find a model to minimize the cross-entropy in order to maximize the probability of giving the correct classification. We employ Adam [19] in our experiments.

V. EXPERIMENT

A. Data collection

A large number of sample paths of a closed network has been used to train the considered prediction models. We have used traces sampled from stochastic simulation of the continuous-time Markov chain underpinning the queueing model. The sampling has been performed using the MATLAB toolbox LINE [20], [21]. The queueing network routing matrices are randomly generated. Randomized parameters include the number of nodes, the number of job classes, the number of jobs per class, the scheduling policy of each node, and the transition probability of the network. Table II summarizes the levels of each parameter that is used to generate a queueing network. The routing matrix is generated using three methods: circular routing, circular with self-loops (referred here as looped routing), and fully random routing, as shown in Figure 2. In the circular routing, the nodes are connected sequentially to form a closed chain. The extension in circular routing adds self-loops at each node. The probability of the jobs going to the next station is generated from a uniform distribution between 0 and 1. In the random link structure, routing probabilities are randomly generated from a uniform distribution between 0 and 1.

Each network also has an infinite server node, referred to as the Delay throughout the study. If the Delay is not added to the network, queueing stations systematically operate in heavier loads and thus state space explosion may occur in the CTMC.
in the presence of multiclass FCFS and HOL nodes, making it more difficult to carry out comparisons with the exact model. All jobs are initialized at the Delay station.

Overall, 25000 queueing network models are generated and simulated for 5000 steps each. A step here indicates that the state of the network has changed from state $i$ to state $j$, whereby state $i$ and state $j$ are directly connected states in the sense that they differ only for a single job having changed its station. Therefore, each network has 5000 states.

The input of the model is a sequence of aggregated states as shown in (1). Each state is concatenated with the standardized time

$$
\hat{t}_i = \frac{t_i - t_{\text{min}}}{t_{\text{max}} - t_{\text{min}}}
$$

Therefore, the input matrix can be written as follow

$$
x = \begin{bmatrix}
\hat{t}_1 & n_{11}^{(1)} & n_{12}^{(1)} & \ldots & n_{1C}^{(1)} & \ldots & n_{NC}^{(1)} \\
\hat{t}_2 & n_{11}^{(2)} & n_{12}^{(2)} & \ldots & n_{1C}^{(2)} & \ldots & n_{NC}^{(2)} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\
\hat{t}_m & n_{11}^{(m)} & n_{12}^{(m)} & \ldots & n_{1C}^{(m)} & \ldots & n_{NC}^{(m)}
\end{bmatrix}
$$

where $n_{ij}^{(t)}$ is the number of jobs of class $j$ at station $i$ at time $t$.

### B. Evaluation setting

Three measurements are used to evaluate the performance of the model, which are accuracy, macro-F1 score and weighted-F1 score [22]. Accuracy is commonly used to represent the performance of a classifier. However, if the test labels are imbalanced, the accuracy cannot provide a valid result to judge the performance of the model. The F1 score is widely used to solve this problem.

The macro-F1 score computes the mean of the F1-scores of all classes, but in our setting, Delay station and “empty” class are easy to classify and exceed the number of other classes, and we do not care as much about them, so we lower their contribution to the final result in comparison to the others. As a result, the weighted-F1 score is calculated, with each class’s weight factored into the final F1 score. The weight for Delay and empty class are set to $1/12$ and the weight for the remaining classes are set to $1/6$.

#### C. Model selection

In sections III and IV, we have presented three types of models that may be used for the problem under study: the baseline model SVM, the RNN model with LSTM cells and the transformer model. In this section, we compare the performance of these models with different length of continuous observations and sampled observations, then pick the best one to use in the rest of our experiment.

The RNN model is made up of ten layers of LSTM cell with a hidden dimension of 128. We consider two compression methods which are taking the mean of all outputs or only considering the final output. After checking the performance of transformer models with different structure, we used 3 encoders and 3 decoders to construct a transformer model and the hidden dimension is also set to 128 with 8 heads in the multi-head attention layers. A zeros matrix with the shape (4, $B$, 128) is used as the input of the decoder.

1) Continuous observations: To check the effect of the sequence length, we try three different sequence length as the input to train the models. At first, only the first 50 time steps are considered as input data, followed by the first 500 and 1000 steps. The models are trained with 200 epochs and the batch-size and learning rate are set to 32 and 0.0001, respectively. The results are shown in Table III

<table>
<thead>
<tr>
<th>Model</th>
<th>Sequence length</th>
<th>Accuracy</th>
<th>Macro-F1</th>
<th>Weighted-F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (SVM)</td>
<td>50</td>
<td>0.6736</td>
<td>0.5252</td>
<td>0.4499</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.7332</td>
<td>0.6164</td>
<td>0.5572</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.7426</td>
<td>0.6328</td>
<td>0.5785</td>
</tr>
<tr>
<td>RNN-mean</td>
<td>50</td>
<td>0.7524</td>
<td>0.7371</td>
<td>0.6973</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.6504</td>
<td>0.6244</td>
<td>0.5715</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.6458</td>
<td>0.6252</td>
<td>0.5831</td>
</tr>
<tr>
<td>RNN-last</td>
<td>50</td>
<td>0.4066</td>
<td>0.2937</td>
<td>0.1975</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.4043</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.4488</td>
<td>NaN</td>
<td>NaN</td>
</tr>
<tr>
<td>Transformer</td>
<td>50</td>
<td>0.8104</td>
<td>0.7975</td>
<td>0.7654</td>
</tr>
<tr>
<td></td>
<td>500</td>
<td>0.9018</td>
<td>0.8937</td>
<td>0.8767</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>0.9042</td>
<td>0.8968</td>
<td>0.8801</td>
</tr>
</tbody>
</table>

The dataset is available at https://zenodo.org/record/5550315#YVxf75pBxPY

---

1The dataset is available at https://zenodo.org/record/5550315#YVxf75pBxPY
The performance of the RNN model with mean compression decreases as the sequence length increases. Again, we find that the mean compression becomes less useful when the length of the sequence is too long in the RNN model, because it loses too much information by taking the mean. The reason for the mean compression method outperforms the last state compression is that the model considers the output of all state, which means that the error might propagate not only backwards in time but also directly to every state. However, it can only outperform the baseline SVM model with a sequence length of 50.

The transformer can produce the best performance among the three models. It performs much better in long sequence data with 90.42% of accuracy, 0.8968 of the macro-F1 score and 0.8801 of the weighted-F1 score compared to the one trained with the sequence length of 50 which accuracy, macro-F1 score and weighted-F1 score are 81.04%, 0.7975 and 0.7654 respectively. Furthermore, from Figure 3, the transformer model can clearly classify every type of scheduling policy.

2) Sampled observations: Next, we consider the situation that the state data is no longer continuous, but we observe the state periodically. Hundreds of thousands of requests are pushed into a web service system, thus tracking every network change is costly. Instead of memorizing all the states, the sensor can record the states periodically; for example, the sensor will save the state if the state of the network has changed every 5 times or 10 times. Another approach is to record the state of the network randomly. Both of these two approaches can possibly be used in a real system with a large number of requests. In this experiment, the RNN model with mean compression and transformer will be trained and tested with 4 types of data which are getting the states for every 5 steps (per5) or 10 steps (per10) and randomly taking 500 (rand500) or 1000 (rand1000) states from the 5000 states. The training parameters are identical to those used in the prior experiment. The results are shown in Table IV.

Transformers trained using periodic data perform better than those trained with data taken at random from 5000 states. Although the models trained with the periodic sequences seem to have higher evaluation scores, we think that the models trained with the random sequence can also provide valuable results on the periodic data because the periodic data is the subset of the random selection. To prove this, the models trained with periodic data are tested on the random sequence. Meanwhile, the models trained with the random sequence are used to classify the scheduling policy on the periodic data.

From Table V, the model trained with the random 1000 states data can also perform well on data record the state every 5 steps with 85.75% of accuracy, 0.8497 of the macro-F1 score and 0.8249 of the weighted-F1 score. On the contrary, the model trained with periodic data cannot give a satisfying classification on the randomly selected data. Therefore, the transformer trained with randomly selected data is more robust than the one trained with periodic data. In the following experiments, the models are trained with the random selected setting. We notice that the performance of the model is also affected by the sequence length. We test the transformers which are trained by the 500 and 1000 randomly selected states data on the test data with different sequence length from 50 to 1600.

As shown in Figure 4, the accuracy and F1 scores reach their maximum when the sequence length of the test data is the same as the training data. When the sequence length of the test data increases, the performance drops marginally, but the model can still give reasonably good classification. However, if the sequence length of the test data decreases, the accuracy and F1 scores drop dramatically. Therefore, we believe the model can handle sequences longer than the training sequence, but it fails to give an useful classification for sequences shorter
D. Training with missing data

In this experiment, we study the robustness of the transformer model by training the model with missing values. There are two different types of missing values. The first type is that the number of jobs $n_{ij}$ is missing with some probability, which means the sensor recording the state of the network is failed by chance. To simulate this situation, we can set the number of jobs at each station to $-1$ with probability $p$ to represent the missing values. The other type is called missing features, which means the sensor cannot monitor some classes of jobs at some stations. In this type of missing value, the columns of the missing features are set to $-1$, and, for each training sample, the missing features are randomly chosen. In the next experiment, we call the data set no masked data if it does not contain a missing value.

1) Missing values: For the first type of missing values, each value $n_{ij}$ is missing with probability $p$, which equals to 0, 0.1, 0.2 and 0.3. The data set is masked with 4 different values of $p$ and used to train the model separately. The training data are selected randomly with 1000 states, just like we did in Section V-C2.

From Figure 5, we can find that the models give the best performance when the test data are masked by the corresponding probability. If the probability of missing data in the test set is different from the training set, the accuracy and the F1 score of the model will decrease. We can also discover that the model trained with the complete data (blue curve) significantly decreases when the probability of missing value increases, which means this model only provides valid classification if the test data does not contain the missing value. The reason behind this is that the training set does not contain -1, which represents the missing value. Therefore, if the test set has -1, the model cannot process the information of missing value.

On the other hand, the models trained with missing data are more robust than the model trained with complete data. For example, the model trained with 10% of missing data (orange curve) gives a useful classification when the test data are masked with the probability between 0.025 to 0.175 where the accuracy and the mean F1 score are around 0.8 and the weighted-F1 score is greater than 0.7. If the probability of missing value is outside this interval, the performances drop dramatically. The models which are trained with 20% and 30% of missing data have a similar property. The result of the models can only perform well in a specific interval of the missing probability that can be due to the fact that the distribution of missing value on the test data is different from the training data. The models do not have knowledge on other distribution of missing value.

To prove this, we choose the probability of missing value in a sample from an uniform distribution with low bound equal to 0 and upper bound equal to $2p$. Therefore, we have more uncertainty on the missing value and get the result as shown in Figure 6. The interval where the model can give promising classification becomes larger, and we can conclude that the model can provide good performance if the testing data stay in similar distribution of the training data.

2) Missing features: In our experiment, the maximum number of stations in a queueing network is 4, and the maximum number of class of jobs is 3, so the total number of features in a state vector is 12 plus the time. For the second type of missing...
values, the number of missing features is set between 0 and 10 except the time, which means the time will never miss. More precisely, if we select $m$ missing features, all the training data are miss $m$ features. Unlike the first type of missing values, we can easily observe how many features are missing in the test data. Therefore, we can select the right trained model according to the test data. So the models are evaluated by the test data with the corresponding number of missing features. The data selection and training settings are the same as the previous experiment.

Figure 7 shows the accuracy, mean F1 score and weighted-F1 score of the model trained by the corresponding number of missing features and tested by the test data set. All three curves have a similar trend and the scores decrease linearly when the number of missing features is increased. Models that were trained with 0 to 6 missing features can nevertheless perform reasonably well. We think that the missing features are randomly selected so the padding features can be selected and there will not be a significant effect on the model if these features are missing. When the number of missing features is greater than 7, the mean F1 score and weighted-F1 score drop faster than the accuracy. The reason is that the models cannot give a precise classification on FCFS, LCFS, SIRO, PS and HOL queue, but they can still classify the Delay and empty stations. This is the case considering the fact that every sample data must have a Delay station and the total number of empty stations is much larger than the other types of queue.

VI. CONCLUSION

This paper focuses on a novel problem of estimating the scheduling policies of a queueing network from aggregate sample path traces. The transformer model is shown to be an effective approach to solving the problem when an extra input is provided to the decoder in the transformer model. This input provides a set of learnable parameters for the decoder to learn a particular part of the key-value pairs passed on by the encoder. The transformer model has been successfully used to classify scheduling policies from a sequence of sampled, non-continuous, states of the queueing network with much greater accuracy than SVMs or RNNs.

Future work may explore the application of this work to actual system measurements and the challenges arising with extended queueing features not considered in this paper.

REFERENCES

S4BXI: the MPI-ready Portals 4 Simulator

Julien Emmanuel∗†, Matthieu Moy∗, Ludovic Henrio∗, Grégoire Pichon†

∗ Univ Lyon, EnsL, UCBL, CNRS, Inria, LIP, F-69342, LYON Cedex 07, France
first.last@ens-lyon.fr
† Atos, Échirolles, France
first.last@atos.net

Abstract—We present a simulator for High Performance Computing (HPC) interconnection networks. It models Portals 4, a standard low-level API for communication, and it allows running unmodified applications that use higher-level network APIs such as the Message Passing Interface (MPI). It is based on SimGrid, a framework used to build single-threaded simulators based on a cooperative actor model. Unlike existing tools like SMPI, we rely on an actual MPI implementation, hence our simulation takes into account MPI’s implementation details in the performance. This paper also presents a case study using the BullSequana eXascale Interconnect (BXI) made by Atos, which highlights how such a simulator can help design space exploration (DSE) for new interconnects.

Keywords—Simulation, SimGrid, HPC, MPI, Interconnect, Portals 4, BXI

I. INTRODUCTION

While High Performance Computing (HPC) clusters are often used to run models of real-world objects in a lot of scientific fields, these systems are themselves so complex that there is valuable information to be learned by simulating them. In particular, having models of performance for HPC hardware enables performing some optimizations in the code or the configuration of the applications using simulation, in order to make the execution on a real cluster as fast as possible. This is important since clusters are limited resources that are expensive to use, both from an economic and ecologic point of view, so performing many test runs on real machines is often problematic. Another important use of simulation results of HPC systems is for the co-design of new hardware: since it is not realistic to create many physical prototypes at every stage of the development of a new chip, the best way to make design decisions is to evaluate new ideas in simulation.

Many simulators already exist for HPC systems (we present some of them in Section II), especially for network interconnects which allow the machines to communicate. This is especially important since clusters keep getting larger, with many servers (nodes) connected together, and a big amount of time is spent in communications. To get an accurate timing for communication, we need to model both low-level network primitives (typically hardware-accelerated) and algorithms used to implement high-level communication primitives. Our simulator, S4BXI†, models the Portals 4 low-level API, a standard network API specified by Sandia and entirely implemented in hardware by the BXI interconnect developed by Atos [1], used in some of the fastest European supercomputers [2]. S4BXI allows running applications using the Message Passing Interface (MPI) standard API, that offers in particular a wide range of collective operations while Portals 4 only offers point-to-point semantics. We do not require any modification to the application, and take into account the implementation details of the Atos implementation of MPI running on top of BXI.

The closest related works are [3], which provides a model of Portals 4 as implemented in BXI, and SMPI [4]. [3] allows simulating unmodified applications using the Portals API directly, but does not allow running MPI applications. Typical HPC applications call MPI primitives, and rely on this very widely used library to call the lower-level Portals primitives, hence this is an important limitation to run real-life applications on the simulator. SMPI allows running unmodified applications using MPI. It does so by catching calls to MPI primitives and abstracts away the details of the hardware, leading to a lack of accuracy for some workloads. Also, it only allows a choice between a fixed set of MPI models corresponding to particular MPI implementations. Our work is based on [3], and follows a path different from SMPI to offer an MPI model. Instead of providing MPI models running on top of the SimGrid [5] network model, we allow a specific real-world MPI implementation to run directly on top of a more precise Portals 4 model, itself relying on SimGrid. We achieved this by providing a patch for OpenMPI to make it compatible with our simulator. The patch is relatively small because the API of the portals simulator we designed mimics faithfully the interface of the real network interconnect. We also adapted SMPI’s performance model for computation to be usable in our simulator, which allows us to get accurate timing when modeling realistic applications.

After presenting related work in Section II, we present the details of how we implemented MPI support in Section III, and the results of our validation experiments in Section IV. Finally, Section V presents some ongoing experiments conducted with our simulators to help the design of next generations of hardware for the BXI interconnect.

II. RELATED WORK

Network models for HPC can generally be categorized in three groups: packet-level models, flow models, and analytical...
models, which all have their own purpose.

Packet-level models are usually close to emulating the hard-
ware, as they model the processing of each individual packet. While this makes for a very accurate model, it is usually very slow to run, which means that it is very impractical, if at all possible, to simulate several machines running a realistic application. These simulators are well suited to help the design of specific circuits when creating new networking hardware, and they are usually written in frameworks like SystemC [6], Omnet++ [7], ns3 [8], or the Structural Simulation Toolkit (SST) [9].

At the other end of the spectrum, analytical models enable very fast simulations by using very abstract methods for timing network transfers. This allows them to model thousands of processes very quickly, while sacrificing a lot of accuracy. Indeed these models often ignore important aspects of the network, which have a huge impact on performance, such as congestion. For this reason, they are more suited to simulate very large scale applications in order to get a very quick but rough estimate of their performance. They include all the LogP family of models, such as LogGOPSIm [10] for example. Another use of these types of models is worst-case timing analysis, which is enabled by techniques such as Network Calculus [11].

The last category of simulators tries to find a middle ground between extreme precision and very good speed, by using a flow model to represent resource sharing: this enables building simulators at message-level, with different messages sharing the bandwidth of the links that they go through. This way a wide spectrum of effects can be taken into account, depending on the purpose of the simulator. In this category, the simulation framework SimGrid [5] and its MPI simulator SMPI [4] are widely used. While SMPI has been shown to give very good results when modeling real-world applications on large-scale clusters, it has a few issues:

1. Since SMPI is a full re-implementation of the MPI standard, it only gives good results when modeling an MPI variant that is supported. While this isn’t a problem when using an MPI implementation already available in SMPI (such as OpenMPI or MPICH), it cannot model faithfully an MPI version that has been tuned to use different collective algorithms for example.

2. Similarly, since function calls are intercepted by the simulator at the MPI level, SMPI doesn’t allow any fine tuning of the MPI implementation itself, neither in its code nor in most of the numerous parameters that MPI typically has. Therefore, while SMPI is a good tool for developers of MPI applications, it isn’t usable by developers who work on optimizing MPI itself, or by advanced MPI users which might want to fine-tune their implementation’s parameters. It isn’t usable to explore any lower-level changes either, such as different hardware behaviors, and therefore it can’t be used to simulate interconnection networks that are not available.

3. While SMPI models completely the network topology of the interconnect, the intra-node communications are modeled in a very simplistic way. In particular, transfers between the memory and the the network controller (NIC) can only be accounted for using multiplying factors applied to the latencies or bandwidths of network links. This isn’t a problem when modeling a relatively slow interconnect, because intra-node transfers’ overhead is then very small, and modeling them as a small latency is a good heuristic. On the other hand, on HPC clusters this model isn’t as good. Indeed most NICs use a PCIe network to communicate with the host memory, and the inter-node cables have speeds of the same order of magnitude as PCIe (usually somewhere between 100 and 200 Gbits/s). Therefore having a more detailed model of the PCIe network is better for the accuracy of the simulator, and enables studying various effects (such as congestion) on this network too.

4. Because SMPI’s model is quite simple, a complex tuning procedure is mandatory to find bandwidth and latency factors that should be applied for different message sizes. Thankfully most of this is automated in various scripts [12], but it is still an additional step that is required to get realistic results, and it is not entirely trivial either.

While most of these characteristics contribute to making SMPI more performant, it leaves some space for new simulators that provide a lower-level model, and therefore a better accuracy (at the cost of some performance).

Even though network communications are important when simulating an HPC application, it isn’t possible to get a good performance estimate without a model for computation phases too. The approaches can be categorized similarly to network models: some simulators go for cycle-accurate models, which emulate very precisely (but slowly) the hardware. These are usually made with SystemC [6] or gem5 [13] for example. On the other hand, some simulators have faster but more approximate ways of timing computations: for example, by default SMPI benchmarks the time between network operations as the application gets executed on the host machine (that is running the simulation), and the measured time is then injected into the simulated world (potentially with a constant multiplying factor). Another option available in SMPI is to disable this automatic benchmarking, and to manually describe computation time in the application’s code itself, which has the downside that the application no longer runs unmodified in simulation.

**S4BXI’s approach:** SMPI’s model of computations is sufficient for our needs, and we reuse it with very little modifications. On the other hand, our network model is different: instead of reimplementing high-level MPI primitives in the model like SMPI, we rely on a real-world MPI implementation, which uses our simulated Portals implementation as a transport, solving points 1. and 2.. Thus, for communication operations, our simulator will run the same code as machines on the real-world clusters, as shown in Figure 1. To address point 3., our simulator includes a simplified model of the PCI network inside each machine [3]. Finally, our simulator still requires some configuration (in particular the bandwidth and latency of PCI and inter-node links), but it is easier to setup than SMPI, since it doesn’t rely as extensively on empirical coefficient to adapt the speed of operations (point 4.). As a
result, our simulator is more specific than SMPI because it specifically targets interconnects compliant with the Portals API, but it is more accurate. It is also slower, but still faster than most packet-level simulators.

III. MPI SIMULATION

To model MPI applications on top of our simulator we used Atos’ version of OpenMPI, which adds an optimized transport (Byte Transfer Layer, abbreviated BTL) for the BXI interconnect to the community version of OpenMPI. This is the version that is traditionally used in production on BXI clusters. As shown in Figure 1, the S4BXI’s workflow is similar to the real-world execution. The main advantage, additionally to the accuracy of the simulation, is that the solution can easily be adapted to a new implementation of MPI, to a new communication library (preliminary experiments on OpenSHMEM [14] gave promising results), or to a new hardware design of the network transport layer (see experiments in Section V). As shown in the figure, S4BXI required small adaptations of the OpenMPI library. We present these adaptations below.

Fig. 1. High level view of simulations workflows and real-world execution

Initializing the MPI library for simulation: Although S4BXI is able to run classical applications written using Portals unmodified, allowing a library such as MPI to run on top of it is still challenging. The main reason for this is that during its initialization, OpenMPI exchanges meta-data about the different processes in the job through the Process Management Interface (PMI). These communications don’t use Portals (since they are used to setup the Portals interface), and they usually occur across an out-of-band network, which can be Ethernet for example. While S4BXI provides a Portals implementation, it doesn’t have a PMI implementation compatible with the simulated world, nor a model for the out-of-band network. Even though making an implementation of PMI in simulation seems feasible, it would be very time-consuming, which is why we directly modified OpenMPI’s code to remove PMI calls, and instead inject values directly from the simulator. This data mainly includes the rank associated with each process, and other identifiers that processes need to communicate with each other (Network IDentifier and Process IDentifier in the case of Portals). These PMI calls are required in the execution on a real-world cluster, and they influence the duration of MPI’s initialization, but on realistic applications the timing of this phase is negligible compared to the actual computations performed by the application.

Modifying OpenMPI components: A simplified view of OpenMPI’s architecture and our modifications is depicted in Figure 2. Because MPI is a very large library with many components, this representation is very simplified, and many components that are not commonly used with Atos’ version are not depicted. The darker boxes are components that run completely unmodified, and lighter ones are those which have been modified to some extent (or that are completely implemented by S4BXI). These modifications are really minor: in total around 400 lines of code were modified, although OpenMPI is composed of several hundred thousand lines of C code. Also, for the vast majority of them they only change the initialization of MPI, which means that they are very easy to maintain, in particular as other developers work on the components that are truly interesting to study: the Collective components, the Point-to-point Management Layer (abbreviated PML) and the BTL. This was confirmed as rebasing our modifications on top of the work of the MPI team at Atos was always trivial and without conflict, for several months now.

Potential optimizations and experimental setup: Our simulation is slower compared to SMPI, this is partially because we have an accurate simulation of the network layer. Our approach consists in running the MPI library modified only where necessary, we thus decide not to try to gain performance by modifying this library. However, we can still obtain different trade-offs between accuracy and performance by modifying the network transport layer. For that purpose, several options exist in S4BXI to simplify the model. These options include “quick acknowledgements (ACK)”, which makes ACK events instantaneous instead of requiring a very small transfer on
the network, and also several levels of detail for the PCI model (to ignore small commands for example). Using these options we can improve the performance of the simulator of up to 30%, while loosing some accuracy but remaining on average significantly more accurate than SMPI. These different trade-offs need to be further investigated and next section will present our experimental validation without using the existing shortcuts (in other words we focus on the gain in accuracy in our experiments).

The fact that all OpenMPI primitives are able to run on top of our simulator with such minor changes demonstrates the versatility of S4BXI, and shows that it is possible to model APIs that have a Portals transport with a relatively small effort.

**IV. EXPERIMENTAL VALIDATION**

Since our low-level Portals model has already been validated for point-to-point operations in [3], our experiments focus on collective operations using MPI. We show here experimental results, first in Section IV-A on OSU Micro-Benchmarks [15], and then in Section IV-B on a realistic application, LULESH [16]. In our experience we will evaluate the accuracy of our simulator and the duration of the simulation, we will also compare our simulator to SMPI (the closest existing simulator). We couldn’t perform a comparison with a packet-level simulator since none exist for the BXI interconnect.

**A. OSU Benchmarks**

OSU Micro-Benchmarks are a complete suite of tests that are used to evaluate the performance of individual MPI primitives. Because there are a lot of collective operations (a few dozens), we do not present every single one of them, but we instead synthesize our results into a few different categories.²

Each graph that we present shows our simulation results in terms of simulated timing, as well as a comparison with the same benchmark executed on a real-world cluster equipped with BXI v2 hardware and AMD EPYC™ 7763 64-Core processors. Even though our simulator supports running several processes on each simulated node, we focus on simulations with one process per node only, to maximize the usage of the BXI network and have a better estimate of how our network model performs. Experiments were executed five times each (for both simulations and the real-world comparison), and error bars show the minimal and maximal values obtained in each case (although the difference is often so small that they merge with the median data point).

The graphs also show the comparison with the same benchmark simulated in SMPI, which has been calibrated using tools described earlier (point 4, in Section II).

²Data and graphs for all benchmarks are available at https://framagit.org/s4bxi/s4bxi-mpi-paper-data
In this case SMPI seems like a better solution than S4BXI, since it simulates faster and has a lower memory footprint.

On the other hand, when the algorithm used for a collective operation has been finely tuned by Atos, and results in different choices than OpenMPI’s community version, SMPI would model a different algorithm than the one used in our implementation, at least for some message sizes and node numbers. On the other hand, S4BXI models the correct algorithm by design, since the complete real-world MPI implementation runs in simulation. This leads to a better accuracy of S4BXI as can be seen in Figure 4, which shows a Scatter benchmark on 16 machines. Scatter spreads data from a root process to every other processes in the job. For this operation, OpenMPI uses two algorithms: basic, which is the simplest implementation where the root node sends a message to every other process in the job, and binomial tree, where data is passed along a tree structure, which causes bigger data transfers but lowers congestion. Experiments show that for small job sizes, the default choice of using mostly basic is not optimal on BXI, and that the binomial tree algorithm is optimal for most message sizes. SMPI does not properly model the switch from basic to binomial tree implemented in Atos’ version of OpenMPI.

Unfortunately there is a last category of benchmarks where both simulators struggle to give accurate results, which corresponds to asynchronous collective operations (benchmarks which start with an “i”). In this case S4BXI gives better results than SMPI in the majority of benchmarks, in particular thanks to the algorithm selection explained previously, but the accuracy of both simulators is not as good as for synchronous collective operations, as can be seen on Figure 5, which shows the IAllgather benchmark running on 16 machines. IAllgather allows all processes to send a piece of data to all the other processes in the job in an asynchronous way.

Thankfully, these operations seem more rarely used than synchronous ones, and we haven’t come across realistic applications that make a heavy use of them. In particular, the application that we study in the next section, LULESH, doesn’t use them at all.

Finally, the relative error of S4BXI and SMPI is depicted for all benchmarks on Figure 6 (for asynchronous collective operations the value is the average of the compute error and the communication error, since these benchmarks measure both, as well as the overlap between communication and computation). We can see that for the most common MPI operations S4BXI performs better than SMPI, and that the asynchronous collectives are indeed the worst case for us, as S4BXI produces approximately the same error as SMPI.

B. LULESH

While OSU micro-benchmarks are useful to evaluate individual primitives, they are not the most representative workload of a real-world application. We now present results on LULESH [16], an hydrodynamics proxy application commonly used in HPC. This application is an interesting case study, as it alternates between intensive computation phases and collective operations for data exchanges, which will stress both our network and computation models.

Our setup is as follows: we run the benchmark on a variable number of nodes (with one process per node), with different problem sizes (which has a big impact on the execution time of the application). We were able to run the real-world execution on two clusters: the system that we used for OSU benchmarks, equipped with AMD CPUs, as well as a cluster with more machines, equipped with Intel®’s Knight Landing CPUs (Xeon Phi™ 7250), which allows our results to go up to 27 nodes (a requirement of LULESH is that the number of processes must be the cube of an integer).

Our results are shown on Table I for the AMD cluster (including a comparison with SMPI), and on Table II for the Intel cluster (with no SMPI comparison because we don’t have a proper calibration of SMPI for this cluster). No error is reported on the tables because the results are extremely consistent across multiple executions, and we always obtain the same values both in simulation and real-world executions (which might be in part due to the fact that LULESH only gives us two significant digits for performance).

Several conclusions can be drawn from this data. First, the executions on one node do not perform any communication,
and therefore we used them to calibrate the computation models for both simulators. For the smaller AMD cluster (Table I), we can see that both simulators are very accurate, with a small advantage for S4BXI (average relative error of 5%) over SMPI (average relative error of 8%). On the bigger cluster (Table II), we can also see that the accuracy of our simulator is very good for bigger problems (Problem size = 30), and for small problems when they are not too distributed (Problem size = 30, Nodes = 8), and it is still reasonable (around 25% longer in simulation than real-life) in unfavorable cases where a lot of small computations are distributed on the largest number of nodes (Problem size = 10, Nodes = 27). Our explanation for this is that when the application runs very fast (in less than ten seconds) on many machines, it puts a lot of stress on the network model, as there is little computation between calls to MPI collective operations. Therefore it makes sense that we get an error that is comparable with what we obtain for OSU benchmarks (which are designed to stress the network). Even though we will investigate this difference in the future, to provide a model as accurate as possible, the error is not too concerning: this type of workload is not very representative of a realistic use of HPC clusters (as such short executions would probably never run on many nodes).

Overall, the accuracy of our simulator on LULESH application is convincing: on realistic workloads we provide a very accurate estimation, and our simulator is always more accurate than SMPI.

### C. A Word on Performance

We have shown that our simulator can accurately model a variety of workloads, but another important aspect of simulation is its performance. Indeed, as explained in Section II, one of the most important characteristics of a simulator is the trade-off between accuracy and performance.

In this regard, we observe different behaviors for network intensive applications like OSU benchmarks, and more realistic ones like LULESH: in the first case, our detailed Portals model is costly compared to other flow models like SMPI, both in terms of execution speed and memory usage of the simulation. On an Intel® Core™ i9-10850K CPU with 4 simulations running in parallel (on different CPU cores) we can simulate all 31 OSU benchmarks in 12 minutes with S4BXI, and about 3 minutes with SMPI. The detail for each benchmark is shown on Figure 7, where the slowdown factor of each simulator is depicted (relative to the real-world execution). We can see that S4BXI is significantly slower than SMPI, as expected, with similar simulation times in the best cases, and one order of magnitude slower in the worst cases. Regarding memory, S4BXI requires around 300MB of memory for each simulated process, regardless of the OSU benchmark, which means that on a very powerful machine with a lot of memory we can expect to be able to simulate up to a few hundred processes, but not thousands as SMPI has been shown to support [17]. To scale past this point in the future, it will be necessary to reduce MPI’s memory consumption. Making the simulation distributed is not realistic because of the sequential nature of SimGrid.

### TABLE I

**Timing Accuracy of Simulators for LULESH (AMD Cluster)**

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Nodes</th>
<th>S4BXI</th>
<th>SMPI</th>
<th>Real-world</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>0.13s</td>
<td>0.13s</td>
<td>0.15s</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>0.39s</td>
<td>0.37s</td>
<td>0.41s</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>3.5s</td>
<td>3.5s</td>
<td>3.5s</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>7.7s</td>
<td>8.4s</td>
<td>8.0s</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>19s</td>
<td>20s</td>
<td>18s</td>
</tr>
<tr>
<td>30</td>
<td>8</td>
<td>41s</td>
<td>44s</td>
<td>40s</td>
</tr>
</tbody>
</table>

### TABLE II

**Timing Accuracy of S4BXI for LULESH (INTEL Cluster)**

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Nodes</th>
<th>S4BXI</th>
<th>Real-world</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>1.6s</td>
<td>1.7s</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>4.8s</td>
<td>4.6s</td>
</tr>
<tr>
<td>10</td>
<td>27</td>
<td>12s</td>
<td>9.5s</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>41s</td>
<td>44s</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>91s</td>
<td>99s</td>
</tr>
<tr>
<td>20</td>
<td>27</td>
<td>150s</td>
<td>150s</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>240s</td>
<td>230s</td>
</tr>
<tr>
<td>30</td>
<td>8</td>
<td>490s</td>
<td>500s</td>
</tr>
<tr>
<td>30</td>
<td>27</td>
<td>720s</td>
<td>790s</td>
</tr>
</tbody>
</table>

### TABLE III

**Performance of Simulators for LULESH (AMD Cluster)**

<table>
<thead>
<tr>
<th>Problem size</th>
<th>Nodes</th>
<th>S4BXI</th>
<th>SMPI</th>
<th>Real-world</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>1</td>
<td>0.25s</td>
<td>0.11s</td>
<td>0.15s</td>
</tr>
<tr>
<td>10</td>
<td>8</td>
<td>6.9s</td>
<td>3.2s</td>
<td>0.41s</td>
</tr>
<tr>
<td>20</td>
<td>1</td>
<td>3.1s</td>
<td>2.9s</td>
<td>3.5s</td>
</tr>
<tr>
<td>20</td>
<td>8</td>
<td>51s</td>
<td>45s</td>
<td>8.0s</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>16s</td>
<td>16s</td>
<td>18s</td>
</tr>
<tr>
<td>30</td>
<td>8</td>
<td>250s</td>
<td>250s</td>
<td>40s</td>
</tr>
</tbody>
</table>
greatly optimized [12], but it requires both a deep knowledge of the modeled application, and the modification of the application code. Such analysis could be re-used in S4BXI, as we used the same model for computation as SMPI.

V. CO-DESIGN OF NEXT-GENERATION HARDWARE

Since Portals is implemented directly in hardware in the BXI interconnect, our simulator is a good tool to experiment with new potential designs for the next-generation Network Controllers (NIC). Because we use a flow model at message-level (and therefore have no representation of individual packets), it isn’t a substitute for more detailed models that are typically built with SystemC for example, but it is complementary because it enables experiments on the processing at message level to be evaluated at a larger scale.

The study that we present is based on flow control, which is a feature that is lacking in the current generation of BXI hardware. There are many ways to implement flow control [18], so the approach we are studying is to limit the number of messages in-flight between each pairs of machines (node-level flow control), or each pair of processes (process-level flow control) at the sender side. We implemented this feature in simulation, which is tunable using environment variables at runtime to test various scenarios. This required little effort, as it represents only about 250 lines of modified code (the total size of the simulator is around 5500 lines of C++ code), which shows how much easier it is to implement message-level processing in our model than in a low-level emulator.

The experimental setup that we use to validate the behavior of our model is as depicted in Figure 8: our simulated platform consists of two switches connected together, with two machines connected to each. This way machines connected to switch 1 that communicate with machines connected to switch 2 will share a common BXI link. While this is a very simple topology, it is representative of real-world scenarios as pruning is very common in HPC clusters, which means that most of the time there will be shared links of this sort between switches (this is especially common in fat-tree topologies). The workload that we simulate is simple: a pair of machines runs eight processes each, which will flood the network by sending as many 1MB messages as possible to each other. While this isn’t what a realistic application would do, it does emulate realistic situations where an application running on many nodes would have an intense communication phase. On the other hand, the remaining pair of machines simply runs one process each, which sends a fixed amount of 1MB messages sequentially. We measure the latency between the second pair of machines (top pair of Figure 8), which gives us an estimate of the congestion on the shared BXI link.

We run this experiment in two scenarios: node-level flow-control and process-level flow control. The result is shown on Figure 9, where the average latency of a message is represented as a function of the number of messages authorized inflight, and the horizontal dashed line represents the latency when flow control is completely disabled.

We can see that the results are as expected: as the flow control gets more strict, the nodes flooding the network are slowed down, which allows the other node pair to exchange
messages with a far lower latency. When the flow control configurations reaches the maximum number of messages that nodes are able to send in parallel, the latency becomes constant as the control-flow has no effect. We can also see that small values of flow control have a greater impact at the node-level than process-level, which is also expected since there are eight processes on the machines that flood the network.

These results confirm that flow control can be a useful feature in the presented type of workload, and they help us quantify the decrease in congestion that we could expect if it was implemented in the future generation of interconnect. We also executed LULESH with different levels of flow-control, and we observed that this configuration has a negligible influence on the performance of the application. This shows that different workloads can be affected very differently by this feature. To go further, it is now necessary to use lower-level simulators in order to take the final decision and decide how to implement this feature.

VI. CONCLUSION AND FUTURE WORK

We presented our approach for MPI simulation, using the low-level Portals model that S4BXI provides. We showed that it gives an accurate prediction for a variety of workloads, whether on micro-benchmarks or on more realistic applications. We also quantified the cost of this accuracy in terms of performance, and compared our results to a state-of-the-art simulator, SMPI. Finally, we described a practical use of our simulator to study potential improvements in the next-generation BXI hardware, by implementing hardware-level flow control. This allowed us to quantify the benefits of using different algorithms, with a small effort in the adaptation of our simulator.

The next step to improve our simulation is to focus on the performance of our model while ensuring a good accuracy of the simulation. A first set of optimisations of the network transport layer are already available in S4BXI (see Section III). We plan to extend these options further, so that the performance of our lowest-precision model is as close as possible to SMPI’s performance. We also need to investigate variable-precision simulation, where part of the network is modeled using a precise but performance-costly model, and the rest uses a faster but more abstract model, which could be especially useful for workloads that heavily rely on a “root” process doing more work than the others.

Another strength of our simulator that hasn’t been described in detail is its versatility: since it models a low-level communication API, it should be usable under any high-level network API that has a Portals transport. In particular, we are working on running OpenSHMEM [14] (a Partitioned Global Address Space library, which is used to transparently share memory across machines) on top of S4BXI. Most primitives already work with few efforts, so we should be able to get performance results soon using this workflow.

Acknowledgements: We would like to end by thanking the BXI low-level and MPI teams at Atos for their help in understanding Portals and OpenMPI, as well as the SimGrid team for their valuable help throughout the development of our simulator, especially Tom Cornebize for his help on the tuning of SMPI, and Arnaud Legrand for his advices.

REFERENCES


Energy-Efficiency Comparison of Common Sorting Algorithms

Norbert Schmitt  
Chair of Software Engineering  
University of Würzburg  
Würzburg, Germany  
norbert.schmitt@uni-wuerzburg.de

Supriya Kamthania  
Hewlett Packard Enterprise  
Bengaluru, India  
supriya.kamthania@hpe.com

Nishant Rawtani  
Hewlett Packard Enterprise  
Bengaluru, India  
nishant.rawtani@hpe.com

Luis Mendoza  
Hewlett Packard Enterprise  
Houston, USA  
luis.mendoza@hpe.com

Klaus-Dieter Lange  
Hewlett Packard Enterprise  
Houston, USA  
klaus.lange@hpe.com

Samuel Kounev  
Chair of Software Engineering  
University of Würzburg  
Würzburg, Germany  
samuel.kounev@uni-wuerzburg.de

Abstract—With the rising demand for information technology comes an increase in energy consumption to power it. Namely, cloud computing is constantly growing, in part due to a rising number of devices using the cloud to provide certain functionality. This growth leads to an increase in energy consumption in data centers and is estimated to climb to over 1PWh in 2030. Hardware manufacturers counter the rising demand for energy in cloud data centers by providing techniques to make servers more energy-efficient. However, the advances cannot fully compensate for the growth. To further increase energy efficiency, software needs to be addressed as well but is often neglected by the developers.

In this paper, we compare six sorting algorithms, a common task in most programs, against each other in terms of energy efficiency to allow developers to select the best solution to their problem. We selected well-known algorithms in two variants, and two implementation languages, C and Python. We ran each algorithm on two, out of four, state-of-the-art server systems with different CPUs.

Index Terms—Energy Efficiency, Sorting Algorithms, Power Consumption, Software Efficiency, Cloud, Data Center, Green Computing, Performance

I. INTRODUCTION

A growing number of services are deployed in the cloud. Additionally, an increasing number of devices are using the cloud to provide functionalities and enabling data centers to grow. This growth, in turn, results in a rise in energy demand with a total energy consumption of 263 TWh annually in 2020 expected to grow to an estimated 1137 TWh by 2030 [1]. Moreover, while advances in hardware increase their energy efficiency for sustainable growth, they cannot fully compensate for the increase in energy demand in data centers [2]. Therefore, it is essential to include software as a significant part of conserving energy and increasing energy efficiency.

Fortunately, cloud data centers can be made more efficient. Many technologies exist to increase the energy efficiency of data centers on the hardware level, like dynamic voltage and frequency scaling (DVFS), and benchmarks to create an incentive for manufacturers to improve energy efficiency, such as the SPECpower_ssj® 2008 [3]. For software, energy can be conserved by intelligently placing or consolidating deployed software [4] or using only the minimal amount of resources to satisfy user demands through auto-scaling [5]. However, the software itself needs to be designed and implemented with energy efficiency in mind.

In a 2016 study by Pang et al. about software energy consumption, out of 122 programmers, only 18% answered that they take energy consumption into account when writing software. More positively, 14% considered minimizing energy consumption a requirement, and 21% responded they already changed the software to consume less energy [6]. This survey shows that optimizing energy consumption is not widely considered and adapted even if it does not mean trading energy consumption for lower performance, as Capra et al. showed by testing two different enterprise resource planning (ERP) systems. While both had similar performance differing by about 5%, their energy efficiency varied by about 50% [7]. Pinto et al. also argue that energy consumption is becoming a more critical aspect of software engineering and development, but there is a lack of knowledge on achieving less energy consumption by software [8].

Closing this knowledge gap is essential. One important aspect is selecting a good or even the best algorithm for a given task. One task that probably is part of most programs is sorting data. Sorting algorithms are well known, and some research exists focusing on conserving energy by looking at those algorithms [9]–[11]. However, they put their focus either on battery-powered mobile devices [9], [10] or on comparing different data types (floating-point versus integer) of a small problem size on desktop computers [11].

Our contribution in this paper is the analysis of the energy efficiency of six common sorting algorithms. We use state-of-the-art server systems, two different programming languages, and two different implementation variants for Python and C to analyze which or if an approach is more energy-consuming.

...
than another. We also compiled a list of guidelines for energy efficient sorting. Industry practitioners in software development then could utilize our results to make informed decisions about the most suitable sorting algorithms for their problem at hand.

The remainder of this paper is structured as follows. We first give an overview of related work in Section II, followed by the description of our testbed setup, selected algorithms, and approach on the measurements in Section III. We then evaluate the energy efficiency of common sorting algorithms in Section IV and present the compiled guidelines in Section V. Afterwards threats to validity are discussed in Section VI and we conclude our work in Section VII.

II. RELATED WORK

Much research has been conducted regarding power consumption, energy consumption, and energy efficiency. In this section, we will give an overview of works related to energy consumption, decreasing energy wastage, and increasing energy efficiency, as well as the energy efficiency of algorithms. This section is based partially on previous work [12].

For mobile devices power and energy consumption are pressing issues, primarily if the devices are powered by batteries. In [13] Li et al. present an approach to measure power consumption on a source line level. The authors combine hardware-based power measurements with program analysis and statistical modeling. While executing an application, the approach measures the energy and derives the executed parts of the application using path profiling. The per-line consumption is then evaluated using static and regression analysis as presented in the authors’ previous works [14]. However, this approach requires a significant amount of resources to perform the profiling, reducing accuracy.

Pathak et al. introduce eprof, a fine-grained energy profiler for smartphone apps [15]. When applied to several apps, eprof exposes energy drainers like third-party advertisements or pinpoints wake lock bugs in the code. Next, bundles are introduced to help developers optimize their app’s energy drain by presenting the app’s I/O energy consumption.

Another tool-assisted approach by Zhang et al. [16] introduces two tools. PowerBooter constructs a power model without using a power meter. PowerTutor is a tool that, using online analysis, shows developers the implications of their design choices on power consumption. Unlike [13], this approach is not based on the code line level but instead on the component level. Thus, the model is built using the consumption of the CPU, LCD, GPS, Wi-Fi, 3G, and audio components.

For Java-based software systems, Seo et al. [17] present the first iteration of a framework for estimating power consumption. This approach focuses on the interaction among distributed components. Thus, it allows developers to estimate their systems’ power consumption at design time. However, the component-based approach is coarse, making informed decisions for developers of single components complicated due to the high abstraction.

Banerjee et al. [18] generate an event flow graph of a mobile application and trace the execution to find pieces of code related to high energy consumption. They distinguish between energy hotspots, shorter peaks, and energy bugs that result in increased energy consumption over extended periods of time. This approach aids the developers in showing them that energy hotspots and bugs are present and locates them in the code.

These works on mobile devices have in common that they do not directly use CPU performance counters but use either the sensors of a mobile device, tracing, profiling, or Java’s bytecode. While this might work for the specific device or language and gives insight into where energy is wasted, it is complicated to derive accurate, practical guidance without intensive measurement setups [15], [18] for developers to decrease energy wastage and increase energy efficiency.

Other works shift their focus away from modeling the power and energy consumption of devices and relate to the energy-efficiency of software without regard to what type of hardware is executing the application. As a good example, Capra et al. measured and analyzed complete software systems: two ERP systems, two customer relationship management (CRM) systems, and four database management systems (DBMS).

The authors could show that software can differ considerably (50%) in energy efficiency without significant differences in performance (5%) in extreme cases. Further, they found that this discrepancy stems from using hibernation during external requests and that algorithmically more efficient software also can be more energy-efficient [7].

Consequently, Aggarwal et al. [19] developed GreenAdvisor based on their previous work [20] about system-calls as their primary source to predict power consumption. GreenAdvisor is a tool that analyzes the appearance of system calls in an application and maps them to the corresponding function call inside the application. The change of system call behavior across multiple versions is tracked to determine if a significant change in energy consumption has occurred.

Stier et al. modeled the power consumption of software systems with the Palladio Component Model on an architectural level, achieving an error of less than 5.5% [21]. Nevertheless, modeling the power consumption, an abstraction itself, on a high level can lead to a better architecture but is too coarse to make informed decisions for developers of single components of a software system, similar to Seo et al. [17]. Hence, it can achieve only an improvement on the architecture level but not on an algorithmic level. For this reason, we selected sorting algorithms for our analysis as we argue that sorting is a problem common among many applications.

Bunse et al., as in our work, used measurements of six sorting algorithms, with some in both iterative and recursive implementation, to define a trend function, allowing the developers to choose between performance and energy consumption. They continued by using this trend function to increase the battery lifetime of mobile devices [10]. Rashid et al. compared sorting algorithms on an ARM platform, popular for mobile devices, showing that the algorithm and language do affect the energy consumption [9]. While an interesting
and promising approach, in our opinion, mobile devices often are connected to the cloud to fulfill heavy computational tasks, that while conserving energy for a longer battery lifetime, they shift the energy consumption into data centers.

Chandra et al. conducted a basic study on the power consumption, energy consumption, and runtime of sorting algorithms for their work. They found that the energy consumed is related directly to its time complexity and that integer sorting takes less energy than sorting floating-point data [11]. Again, their focus is not on server systems but desktop machines and a small problem size of just 10,000 integer or floating-point numbers.

It can be seen that the focus of energy-efficiency research is on mobile devices, and there is a gap of knowledge on data center hardware. Therefore, we selected state-of-the-art server systems for our measurements. Additionally, most works are concerned with modeling the power or energy consumption and finding the location inside an application where energy is wasted. The actual improvements in energy efficiency often are left to the compiler leveraging compiler performance optimizations. The following related work is taken and adapted from our previous work [22].

Compilers usually are not targeting energy efficiency with the available optimizations; their main target is performance. Hence, to optimize for energy efficiency, the correct optimization settings must be known to adapt a compiler to improve energy efficiency [23], or there must be adaptation points added during compilation. We also took a first look at two different properties, programming language and application domain, of the SPEC CPU 2017 benchmark suite and which property could be responsible for higher energy efficiency when different compiler optimizations are used [24].

Nobre et al. [25] are changing the order in which performance optimizations are performed during compilation. They have shown that while an increase in energy efficiency is possible, the sequence of optimization is dependent on the application and not necessarily transferable to other applications. Kandemir et al. [26] focused on six loop optimizations of a compiler and tested them on five applications and simulations with mixed results. Most loop optimizations increased the power consumption in the embedded systems.

The tool Socrates from Gadioli et al. [27] not only tries to find suitable compiler optimizations but also weaves in code that can be used for tuning the software to different targets, such as power consumption or performance. The additional instructions or parameters are used, for example, to adapt the number of OpenMP threads. Hsu et al. [28] inserted instructions to control the Dynamic Voltage Scaling (DVS). The approach selects program parts suitable for running with lower voltage and frequency without degrading performance above a user-selectable value resulting in energy savings of up to 28% and performance degradation of just 5%, increasing energy efficiency.

Our work extends the related work by aiming not at mobile devices but cloud servers. Additionally, our work is intended as a guideline or help for practitioners to select a better choice of sorting algorithm for energy efficiency before compilation and to leave the fine-tuning to approaches that leverage compiler optimizations.

III. APPROACH

We first describe our used system under test (SUT) and setup, followed by the measurement analysis in which we compare six sorting algorithms in two variants against each other. In a second step we take a look at two different state-of-the-art server systems to analyze if the energy efficiency is transferable or is an attribute of the server. We make all measurements with two different programming languages, C and Python.

A. System Under Test and Setup

As our SUTs, we selected four different state-of-the-art HPE ProLiant servers, each with a different CPU size. All servers are equipped with the same 480GB SSD storage drive. CentOS 8 is used as the operating system. These servers are, in our opinion, a good representation of standard cloud servers that are not purpose built and do not contain any additional acceleration units. Each server’s power supply is connected to a Yokogawa WT210 power analyzer for power measurements. The power analyzer samples internally at about 10kHz and aggregates each sample over one second. The setup is in accordance with the power and performance benchmarking methodology described by Kounov et al. [29]. For our analysis, we do not use the internal sampling rate but the aggregated sampling rate of one second.

TABLE I
Systems Under Test.

<table>
<thead>
<tr>
<th>Name</th>
<th>Cores/Threads</th>
<th>Clock</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server A</td>
<td>128/256</td>
<td>2.25GHz</td>
<td>16x16GB</td>
</tr>
<tr>
<td>Server B</td>
<td>56/112</td>
<td>2.20GHz</td>
<td>12x16GB</td>
</tr>
<tr>
<td>Server C</td>
<td>32/64</td>
<td>3.00GHz</td>
<td>16x16GB</td>
</tr>
<tr>
<td>Server D</td>
<td>36/72</td>
<td>2.00GHz</td>
<td>12x16GB</td>
</tr>
</tbody>
</table>

B. Selected Sorting Algorithms

We have selected six commonly known sorting algorithms, listed in Table II. The algorithms were chosen as they have different runtime behaviors, with a range of \( n^2 \), \( n \), and \( n \log(n) \) for their best, average, and worst case respectively. While Merge and Heap Sort have identical time complexity of \( n \log(n) \), they differ in space complexity. Stability of an algorithm has not been considered. We also did not consider hybrid sorting algorithms like Tim Sort or Intro Sort that combine techniques from algorithms already in our list.

The authors self-implemented each algorithm in two variants with their differences shown in Table III to avoid bias of a specific implementation variant, and two programming languages, Python and C. We selected Python and C as they are well known representatives for interpreted and compiled...
languages\textsuperscript{1,2}. Yet it must be kept in mind that the results might not be transferable to other languages without further measurements.

<table>
<thead>
<tr>
<th>Name</th>
<th>Time Complexity</th>
<th>Space Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>$n \log(n)$</td>
<td>$n$</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>$n \log(n)$</td>
<td>$n \log(n)$</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>$n \log(n)$</td>
<td>$n^2 \log(n)$</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>$n \log(n)$</td>
<td>$n^2$</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>$n^2$</td>
<td>1</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>$n^2$</td>
<td>1</td>
</tr>
</tbody>
</table>

To stress the SUT, each algorithm had to sort a number of integers, which we refer to as the problem size. For each algorithm, we selected three problem sizes shown in Table IV. The problem size is split across the available threads of the SUT listed in Table I. The problem sizes shown in these tables for each algorithm were calibrated separately based on the average time complexity characteristics. Setting common problems sizes across algorithms, servers, and implementation variants proved unfeasible because setting small problem sizes for algorithms with linear time complexity, like Heap Sort, reduces run times and not giving sufficient time for data collection from the power analyzer. Similarly, setting large problem sizes for algorithms with quadratic time complexity, like Bubble Sort, increases the runtime to impractical lengths. The problem sizes used for C and Python are different as we would expect a performance contrast between them. To allow for stable measurements of the C algorithms that are roughly equal in length than the Python measurements, the problem size would have to be larger than the memory and would distort the measurement due to disc IO. We, therefore, took actions, such as reducing the problem size, to achieve a stable measurement while at the same time keeping disc IO to a minimum.

The total problem sizes used for the SUTs are kept constant. For example, if we calibrated the Python implementations of Insertion Sort to a total of 10,240,000 (see Table IV), all integers that must be sorted are distributed across the cores equally. If we want to sort 10,240,000 integers on server A with 256 threads and server B with 112 threads, each thread of server A is given 40,000 (10,240,000/256) integers and each thread of server B is given 91,428 (10,240,000/112) integers. Pseudo-random numbers were generated by reading from the \texttt{urandom} file for the C implementations and using the \texttt{random} library for the Python implementations. All problem sizes fit in the memory of the SUTs.

### C. Analysis

Measurements are taken on four SUTs. For each sorting algorithm, two variants in two different programming lan-


\textsuperscript{2}https://www.tiobe.com/tiobe-index/. Accessed: June 18th, 2021

The energy consumption is calculated by integrating over the measured power consumption. For our sampling interval of one second, this is the sum of the power consumption $P$ in Equation 1. To calculate energy efficiency shown in Equation 2, the energy consumption is calculated by dividing the problem size $p$ (total number of sorted integers) by the total energy consumption $E$, resulting in the number of integers sorted by Joule. For the number of bytes sorted by Joule, we multiply $p$ by four, the size of an integer on the SUTs. We count a variant as preferable if the 95% confidence intervals do not overlap.

$$E = \sum P$$

$$\text{Eff} = \frac{p \times 4}{E}$$

### IV. Analysis

First, we will take a look at the algorithms that do not have a logarithmic runtime behavior. It is clear from the results in Table V and VI that the three sorting algorithms that have an average time complexity of $n^2$, Bubble Sort, Insertion Sort, and Selection Sort, are much less energy efficient than the algorithms with $n \log(n)$ as expected no matter the implementation variant.

Regarding the three algorithms with $n \log(n)$ time complexity, Heap Sort, Merge Sort, and Quick Sort, the Heap Sort algorithm is outperformed across almost all implementation variants, problem sizes, and SUTs (see Table V). The only exception is Quick Sort on server A, variant 2 on a small problem size with 62MB per Joule compared to 64MB per Joule for Heap Sort. We can also determine that even an inefficient implementation to generate the max heap for Heap Sort is not the main reason for this behavior as variant 1 and 2 achieve similar energy efficiency values on all problem sizes and both SUTs. Merge Sort maintains a stable energy efficiency across the problem sizes for each server. One exception is variant 2 on server A with a medium problem size close to 62MB per Joule but still in the same range of Quick Sort and outperforming Heap Sort given this configuration. For Heap Sort on server A with a higher thread count, inlined code (variant 1) has higher efficiency. At the same time, not inlining and calling additional functions seems beneficial for servers with smaller core counts. Quick Sort scales well on server A and becomes more energy-efficient the larger the problem size grows. Although, on server B, it seems to plateau as the configurations for the problem sizes are similar by the 95% confidence interval. In general, Quick Sort also has a higher energy efficiency than Merge Sort for server B.
### TABLE III
IMPLEMENTATION VARIANTS FOR PYTHON.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Python Variant 1</th>
<th>Python Variant 2</th>
<th>C Variant 1</th>
<th>C Variant 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>Recursive and code inlined</td>
<td>Recursive and code distributed</td>
<td>Recursive and without dynamic memory allocation</td>
<td>Recursive and with dynamic memory allocation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>across multiple functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Heap Sort</td>
<td>Efficient iterative loop to build</td>
<td>Inefficient iterative loop to</td>
<td>Memory based swapping</td>
<td>Pointer based swapping</td>
</tr>
<tr>
<td></td>
<td>max heap</td>
<td>build max heap</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Quick Sort</td>
<td>Iterative</td>
<td>Recursive</td>
<td>Memory based swapping</td>
<td>Pointer based swapping</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>no separate function for sorting</td>
<td></td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>Iterative</td>
<td>Recursive</td>
<td>Memory based swapping</td>
<td>Pointer based swapping</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>but additional function to perform sort operation</td>
<td></td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>Iterative</td>
<td>Recursive</td>
<td>Memory based swapping</td>
<td>Pointer based swapping</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selection Sort</td>
<td>Explicitly type checking in return</td>
<td>Dynamic return type (no explicit</td>
<td>Memory based swapping</td>
<td>Pointer based swapping</td>
</tr>
<tr>
<td></td>
<td>function to ensure the consistency</td>
<td>type checking)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>between the element types going</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>in and out of the sorting function</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### TABLE IV
CALIBRATED PROBLEM SIZES FOR PYTHON.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Small</th>
<th>Medium</th>
<th>Large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Python</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Merge Sort</td>
<td>1,024,000,000</td>
<td>1,152,000,000</td>
<td>1,280,000,000</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>1,280,000,000</td>
<td>1,408,000,000</td>
<td>1,536,000,000</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>1,536,000,000</td>
<td>1,664,000,000</td>
<td>1,792,000,000</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>10,240,000</td>
<td>11,520,000</td>
<td>12,800,000</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>10,240,000</td>
<td>11,520,000</td>
<td>12,800,000</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>10,240,000</td>
<td>11,520,000</td>
<td>12,800,000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>5,760,000,000</td>
<td>6,120,000,000</td>
<td>6,480,000,000</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>5,760,000,000</td>
<td>6,120,000,000</td>
<td>6,480,000,000</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>5,760,000,000</td>
<td>6,120,000,000</td>
<td>6,480,000,000</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>3,600,000</td>
<td>7,200,000</td>
<td>10,800,000</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>3,600,000</td>
<td>7,200,000</td>
<td>10,800,000</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>3,600,000</td>
<td>7,200,000</td>
<td>10,800,000</td>
</tr>
</tbody>
</table>

Essentially, better energy efficiency for a larger thread count can be observed between both SUTs for Python. These results are expected. We assume that this stems from the measurement setup, measuring wall power at the power supply for the complete server. As server A has a higher thread count but, apart from memory, similar hardware, the base power consumption is similar, resulting in a lower energy efficiency score. This lower score is neither attributable to the CPU or the algorithm.

For the C implementations, shown in Table VI, we take a look at only the three algorithms with a time complexity of $n \cdot \log (n)$. It can be observed that Merge Sort outperforms all other sorting algorithms by a large margin if the memory is not dynamically allocated in variant 1. Nevertheless, if the memory is dynamically allocated with variant 2, its energy efficiency is reduced by a factor ranging from 7.3 on the medium problem size and server C up to 13.4 on the large problem size and server D.

Quick Sort is the next most energy-efficient implementation in C and should be preferred over a Merge Sort with dynamically allocated memory. Both variants of Quick Sort are more energy efficient than variant 2 of Merge Sort. Yet, in all but one case (medium problem size on server C), Quick Sort becomes less energy efficient when it is implemented with pointer based swapping. This single case for Quick Sort is probably due to the large confidence intervals and we assume that additional measurement runs would resolve this outlier.

Regarding Heap Sort, the difference between pointer and memory based swapping is not as clear. There are two measurements on server C, the small and large problem size, that do overlap on the 95% confidence interval. Given that Quick Sort has a better space complexity of $\log (n)$ compared to Heap Sort with $n$, it is expected that Heap Sort is less susceptible to changes on how the swapping in memory is handled.

### V. GUIDELINE

We will take the findings from our measurements in Section IV before and provide some basic guidelines for practitioners for each sorting algorithm. We left out Bubble Sort, Insertion Sort, and Selection Sort on purpose as a much higher energy efficiency can be achieved and we would recommend, given the choice, implementing a different algorithm with a better time complexity. We also found that there are major differences even for algorithms of similar time complexity. We marked in front of each guideline whether it is applicable to both implementation languages (Both), only for Python, or only for C.

**Merge Sort**

*Both* Merge Sort is preferable over a Heap or Quick Sort algorithm when memory is no constraint.

*Python* Use Merge Sort for up to 1.2 billion integers (medium problem size).

*Python* In case of a very large amount of sorting data, a Quick Sort algorithm might be a better choice.
C Preallocate memory if possible. If the memory must be allocated dynamically, a Quick Sort implementation should be preferred.

Heap Sort

Both Implement a Quick Sort algorithm instead when possible and the space complexity is no hard constraint.

Python Max heap generation can be neglected in terms of energy efficiency.

Python Does not scale as good with the growing size of the sorting problem.

Quick Sort

Python Implement iterative variant if possible to achieve a better energy efficiency.

Python It scales better with the growing size of the sorting problem.

C Pre-allocate memory if possible.

VI. Threats to Validity

The threats to validity are discussed in the order of severity rated by the authors.

a) Lower number of repetitions: All measurements were repeated five times. However, a higher number of repetitions might reduce the variance and, therefore, some confidence intervals might not overlap, which will negate the conclusions drawn from the measurements. As this work is intended as a guideline, a selection of a good algorithm instead of the optimal still can be of value.

b) Selection of sorting algorithms: The selection of algorithms to choose for this work has been based on the time and space complexity of the algorithms. We are aware that alternatives with similar time and space complexity might exist and argue that, given the assumption that they were implemented with care and perform as intended, they should be interchangeable. Stability, on the other hand, was not

<table>
<thead>
<tr>
<th>Server</th>
<th>Problem Size</th>
<th>Variant 1 Mean</th>
<th>95% CI</th>
<th>Variant 2 Mean</th>
<th>95% CI</th>
<th>V1 and V2 Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Merge Sort</td>
<td>A Small</td>
<td>95,827.81</td>
<td>2219.27</td>
<td>89,306.57</td>
<td>320.12</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>93,869.7</td>
<td>702.76</td>
<td>61,848.02</td>
<td>1050.81</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>93,251.27</td>
<td>531.87</td>
<td>86,783.91</td>
<td>305.66</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>B Small</td>
<td>42,599.25</td>
<td>255.33</td>
<td>45,074.79</td>
<td>177.47</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>42,401.23</td>
<td>147.29</td>
<td>44,907.01</td>
<td>178.77</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>42,149.87</td>
<td>162.97</td>
<td>45,016.84</td>
<td>200.9</td>
<td>×</td>
</tr>
<tr>
<td>Heap Sort</td>
<td>A Small</td>
<td>64,238.06</td>
<td>1888.36</td>
<td>64,072.1</td>
<td>2055.87</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>49,276.52</td>
<td>3061.87</td>
<td>49,179.08</td>
<td>3273.6</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>50,795.02</td>
<td>3537.18</td>
<td>49,664.16</td>
<td>3817.94</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>B Small</td>
<td>31,540.17</td>
<td>70.99</td>
<td>31,561.86</td>
<td>46.96</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>31,514.58</td>
<td>77.78</td>
<td>31,447.73</td>
<td>86.82</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>31,350.92</td>
<td>59.84</td>
<td>31,331.65</td>
<td>52.29</td>
<td>✓</td>
</tr>
<tr>
<td>Quick Sort</td>
<td>A Small</td>
<td>85,200.41</td>
<td>500.45</td>
<td>61,514.99</td>
<td>833.26</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>87,689.18</td>
<td>395.88</td>
<td>62,400.97</td>
<td>1038.0</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>119,615.0</td>
<td>670.43</td>
<td>77,710.85</td>
<td>255.19</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>B Small</td>
<td>72,838.23</td>
<td>514.72</td>
<td>49,790.54</td>
<td>361.29</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>72,994.33</td>
<td>715.83</td>
<td>49,872.73</td>
<td>338.38</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>73,519.13</td>
<td>709.49</td>
<td>49,995.9</td>
<td>310.38</td>
<td>✓</td>
</tr>
<tr>
<td>Insertion Sort</td>
<td>A Small</td>
<td>585.35</td>
<td>2.82</td>
<td>42.03</td>
<td>0.12</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>400.73</td>
<td>3.4</td>
<td>35.41</td>
<td>1.6</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>447.42</td>
<td>3.99</td>
<td>31.76</td>
<td>0.06</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>B Small</td>
<td>102.05</td>
<td>0.32</td>
<td>2120.12</td>
<td>75.03</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>90.15</td>
<td>0.17</td>
<td>7.64</td>
<td>0.92</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>81.16</td>
<td>0.06</td>
<td>6.63</td>
<td>0.95</td>
<td>×</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>A Small</td>
<td>263.49</td>
<td>17.22</td>
<td>274.77</td>
<td>10.95</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>186.99</td>
<td>22.5</td>
<td>184.75</td>
<td>21.4</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>173.83</td>
<td>15.49</td>
<td>175.83</td>
<td>15.11</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>B Small</td>
<td>67.61</td>
<td>5.8</td>
<td>192.27</td>
<td>6.64</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>41.34</td>
<td>0.1</td>
<td>42.13</td>
<td>0.17</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>37.09</td>
<td>0.08</td>
<td>32.51</td>
<td>14.55</td>
<td>✓</td>
</tr>
<tr>
<td>Selection Sort</td>
<td>A Small</td>
<td>414.76</td>
<td>48.77</td>
<td>429.13</td>
<td>45.98</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>394.48</td>
<td>2.23</td>
<td>390.1</td>
<td>4.08</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>410.4</td>
<td>17.12</td>
<td>414.7</td>
<td>11.83</td>
<td>✓</td>
</tr>
<tr>
<td></td>
<td>B Small</td>
<td>100.39</td>
<td>0.26</td>
<td>102.08</td>
<td>0.36</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Medium</td>
<td>88.82</td>
<td>0.36</td>
<td>90.46</td>
<td>0.22</td>
<td>×</td>
</tr>
<tr>
<td></td>
<td>Large</td>
<td>79.44</td>
<td>0.38</td>
<td>81.35</td>
<td>0.12</td>
<td>×</td>
</tr>
</tbody>
</table>
TABLE VI
ENERGY EFFICIENCY FOR BOTH C IMPLEMENTATION VARIANTS IN SORTED K B PER JOULE.

<table>
<thead>
<tr>
<th>Server Size</th>
<th>Variant 1</th>
<th>Mean</th>
<th>95% CI</th>
<th>Variant 2</th>
<th>Mean</th>
<th>95% CI</th>
<th>Overlap</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Small</td>
<td>3 057 140.43</td>
<td>68 234.82</td>
<td>401 219.46</td>
<td>1867.08</td>
<td>401 219.46</td>
<td>1867.08</td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td>2 874 856.35</td>
<td>607 613.69</td>
<td>393 527.45</td>
<td>20 993.1</td>
<td>393 527.45</td>
<td>20 993.1</td>
<td></td>
</tr>
<tr>
<td>Large</td>
<td>5 406 575.21</td>
<td>109 261.19</td>
<td>376 680.4</td>
<td>110 104.92</td>
<td>376 680.4</td>
<td>110 104.92</td>
<td></td>
</tr>
<tr>
<td>C Small</td>
<td>3 554 848.68</td>
<td>702 507.71</td>
<td>322 805.22</td>
<td>1190.23</td>
<td>322 805.22</td>
<td>1190.23</td>
<td></td>
</tr>
<tr>
<td>Medium</td>
<td>3 946 797.65</td>
<td>572 099.29</td>
<td>318 950.61</td>
<td>3432.17</td>
<td>318 950.61</td>
<td>3432.17</td>
<td></td>
</tr>
<tr>
<td>Large</td>
<td>4 263 224.3</td>
<td>261 687.77</td>
<td>317 969.18</td>
<td>3100.83</td>
<td>317 969.18</td>
<td>3100.83</td>
<td></td>
</tr>
</tbody>
</table>

Problem Variant 1 | Variant 2 | V1 and V2
--- | --- | ---
Merge Sort | | |
C Small | 156 507.34 | 9793.82 | 156 538.77 | 2647.3 | ✓ |
| Medium | 159 005.12 | 491.7 | 156 300.54 | 1164.83 | x |
| Large | 3 156 421.5 | 2314.56 | 153 358.12 | 4113.29 | x |

Heap Sort | | |
C Small | 112 808.83 | 165.06 | 107 845.36 | 110.66 | x |
| Medium | 112 207.69 | 350.81 | 107 072.05 | 135.36 | x |
| Large | 111 542.68 | 403.48 | 106 421.56 | 474.03 | x |

Quick Sort | | |
C Small | 712 085.62 | 2818.19 | 581 176.35 | 49584.84 | x |
| Medium | 544 565.18 | 776.15 | 420 779.36 | 3926.93 | x |
| Large | 689 021.31 | 74343.39 | 604 550.75 | 6370.04 | x |

Insersion Sort | | |
C Small | 1571.49 | 23.95 | 1299.39 | 13.67 | x |
| Medium | 1360.63 | 11.74 | 851.6 | 5.85 | x |
| Large | 819.89 | 5.01 | 600.31 | 2.01 | ✓ |

Selection Sort | | |
C Small | 368.83 | 2.2 | 349.39 | 2.87 | ✓ |
| Medium | 192.95 | 0.88 | 181.93 | 1.16 | ✓ |
| Large | 217.86 | 0.52 | 121.3 | 1.31 | x |

Buble Sort | | |
C Small | 571.07 | 3.53 | 901.45 | 5.29 | x |
| Medium | 314.75 | 1.36 | 537.04 | 3.14 | ✓ |
| Large | 211.29 | 1.31 | 369.8 | 0.79 | x |

Selection Sort | | |
C Small | 497.21 | 963.3 | 837.31 | 23.62 | ✓ |
| Medium | 264.64 | 1.39 | 477.16 | 4.57 | x |
| Large | 176.83 | 1.09 | 320.51 | 1.09 | x |

Considered and the results might not be generalizable to algorithms with different stability properties.

c) Programming language selection: We selected the programming languages Python and C as representatives for an interpreted and compiled language, respectively. Both languages are publicly available for extended periods of time, well-known, and widely used. We are aware that the compilation can introduce optimizations but argue that this is not a threat as compiler optimizations, ahead-of-time or just-in-time, also might be used in a real-world scenario.

d) Problem size selection: We took care to select three different problem sizes for each algorithm that allowed for a stable measurement while keeping the measurement feasible timewise. However, due to the nature of the time complexity of slow algorithms, not all algorithms have been calibrated to identical problem sizes. This might invalidate our conclusions when absolute energy consumption results are compared but not for the normalized energy efficiency.

e) Limited number of server configurations: As measuring takes time and the number of possible server configurations is vast, considering all configurations becomes practically infeasible. We, therefore, opted for four configurations of state-of-the-art servers as representative of a cloud data center scenario. As the four options consider both major x86 CPU manufacturers, we see this as a good starting point for this analysis, although, it is possible that servers with a lower computational power might yield different results. Other instruction set architectures have not been considered as, in our opinion, cloud hardware predominantly uses x86 servers, so our generalizability is limited to x86 only.

VII. Conclusion

Data centers consume large amounts of energy today, and their energy demand will grow in the future. Energy efficiency needs to be improved to reduce energy consumption in data
centers. While many advances are made for the hardware used, they cannot compensate for the data center growth. The software, on the other hand, often is not considered by developers. There is also some lack of knowledge on how to improve the energy efficiency of software. One aspect is to select a suitable algorithm for the problem. Yet most work on energy-efficient development is concerned with battery-powered mobile devices. Automated approaches using compiler optimizations can be used to improve energy efficiency, but an inefficient algorithm can be optimized only to a certain degree. We therefore analyzed six common sorting algorithms for energy consumption to compile lessons learned.

For our analysis, we selected common sorting algorithms and implemented two variants for each algorithm. Each implementation variant is then measured on two state-of-the-art servers five times with three different problem sizes. Our results show that, even with identical average time complexity of \( n \log(n) \), there are differences between algorithms in how energy efficient they are. We compiled the first version of guidelines for developers and plan to extend and generalize these guidelines in the future, including additional programming languages, SUTs, and algorithms. Additional future work is to confirm our experiments on a real-world application.

REFERENCES

An “A+” Heuristic for Dispatching in Large-Scale Systems with Unknown Server Speeds

Cole Stephens  
Department of Computer Science  
Amherst, MA, USA

Kristen Gardner  
Department of Computer Science  
Amherst College  
Amherst, MA, USA  
kgardner@amherst.edu

Abstract—How to dispatch jobs to servers is a question of critical importance in today’s computer systems. While there is a long history of literature on dispatching, much of this work focuses on settings where servers are homogeneous in their speeds—an unrealistic assumption in modern computer systems. Dispatching policies that are heterogeneity-aware typically assume that the dispatcher has access to detailed information about all servers’ speeds; this may also be unrealistic. In this paper, we study the setting in which server speeds are heterogeneous and the dispatcher has no information about server speeds. We propose a novel heuristic, called server accomplishment, that can be used as a proxy for detailed server speed information in heterogeneity-aware dispatching policies. Using extensive simulation studies, we demonstrate that our accomplishment-based dispatching policies are able to bridge most of the performance gap between heterogeneity-unaware policies, such as JSQ(d), and policies that make use of detailed server speed information.

Index Terms—Dispatching; Load balancing; Power-of-\(d\); Heterogeneity

I. INTRODUCTION

How to dispatch jobs to servers is a question of critical importance in today’s computer systems. There is a long history of literature that strives to answer this question, beginning with the canonical Join-the-Shortest-Queue (JSQ) policy, under which an arriving job is dispatched to the server with the fewest jobs in its queue. JSQ is known to minimize mean response time in certain settings, and its performance has been analyzed in a variety of regimes [7], [13], [17], [18]. Unfortunately, JSQ is an impractical choice for modern systems for two key reasons. First, today’s systems operate at large scale. In systems consisting of hundreds or thousands of servers, querying all servers for their queue lengths upon every job’s arrival is prohibitively expensive. Second, today’s systems are heterogeneous, meaning that different servers may operate at different speeds. In this regime, JSQ is no longer optimal because it does not account for differing server speeds.

To reduce the communication overhead in large-scale systems, researchers have proposed a number of alternative approaches. In the “power-of-\(d\)” approach, the dispatcher queries a small number of servers upon a job’s arrival, and dispatches the job to one of the queried servers. A well-known policy in this category is Join-the-Shortest-Queue(\(d\)) (JSQ(\(d\))) [10], [16]. In the “pull-based” approach, servers periodically provide the dispatcher with updates of their statuses. One policy in this category is Join-Idle-Queue (JIQ) [9]. Both the “power-of-\(d\)” and the “pull-based” approaches require far less communication between servers and dispatcher than JSQ, making them suitable for large-scale systems. However, the canonical policies within these approaches assume that server speeds are homogeneous, often leading to poor performance in heterogeneous settings [8], [12], [14], [19].

More recently, new policies have been proposed that account for server speed heterogeneity. Heterogeneity-aware descendants of JSQ(\(d\)) include Shortest-Expected-Delay(\(d\)), Balanced Routing [3], Hybrid SQ(\(d\)) [12], and JSQ(\(d_F, d_S\)) [6]. The HALO class of policies provides heterogeneity-aware adaptations of JSQ(\(d\)), random, and round-robin dispatching [5]. Pull-based policies such as JIQ also admit heterogeneity-aware variants. In all cases, the heterogeneity-aware variants produce lower mean response times than their heterogeneity-unaware analogues. Yet most such policies require the dispatcher to know some information about the servers’ speeds, ranging from the coarse (e.g., heterogeneity-aware JIQ requires only an ordering of servers by their speeds) to the exceedingly detailed (e.g., JSQ(\(d_F, d_S\)) and HALO require the exact speed of each server). While there are a few exceptions that do not require detailed speed information [2], [4], the metric of interest in these papers is the stability region, rather than response time.

In this paper, we study the setting in which server speed information is unavailable to the dispatcher. This situation may arise, for example, in a cloud computing setting where the exact server speeds depend on the underlying physical machine and resource contention experienced by a virtual machine. Unfortunately, most of the heterogeneity-aware policies proposed in the literature are not feasible in this setting.

Our primary contribution is a novel heuristic, called accomplishment, that helps the dispatcher estimate server speeds. A server’s accomplishment at time \(t\) is defined as the number of jobs that have been dispatched to the server by time \(t\). Clearly, the dispatching policy affects the servers’ accomplishments: under random or round-robin all servers have the same accomplishment, whereas policies that do not treat all servers symmetrically will yield higher accomplishments for the favored servers. When server speeds are unknown, accomplishment is a useful heuristic if faster servers tend to become more accomplished. The dispatching policy can then favor more accomplished servers as a proxy for favoring faster servers, thereby effectively implementing a heterogeneity-aware policy.
Because the dispatching policy’s decisions affect the servers’ accomplishments, which in turn affect later dispatching decisions, it is not at all obvious that using accomplishment as a proxy for speed will actually yield dispatching policies that perform well. We will show how four heterogeneity-aware dispatching policies, SED(d), BR, JIQ, and JSQ(d_F, d_S), can be adapted to incorporate the accomplishment heuristic, yielding four new “Accomplishment+” policies: A+SED(d), A+BR, A+JIQ, and A+JSQ(d_F, d_S). Our extensive simulation results demonstrate that, in many cases, the A+ policies are capable of closing most of the performance gap between heterogeneity-unaware policies, such as JSQ(d) and JIQ, and their heterogeneity-aware descendants. As an added benefit, accomplishment is a low-communication heuristic, requiring no more communication between servers and dispatcher than the baseline heterogeneity-aware policies.

The remainder of this paper is organized as follows. In Section II we introduce our model and formally define the accomplishment heuristic. In Section III we define the four “A+” policies that we consider and empirically evaluate their performance in systems with two server speed classes and exponentially distributed service times. In Section IV we address three generalizations aimed at moving to more realistic system settings: general service times, more than two server speed classes, and server speeds that may change over time. Finally, in Section V, we conclude.

II. MODEL AND PRELIMINARIES

We consider a system that consists of \( k \) servers with heterogeneous speeds, where \( k_F \) of the servers are “fast” and \( k_S = k - k_F \) of the servers are “slow” (for more than two classes of servers, see Section IV-B). Let \( q_F = \frac{k_F}{k} \) \((q_S = \frac{k_S}{k})\) denote the fraction of servers that are fast (slow). Service times are exponentially distributed with rate \( \mu_F \) at fast servers and rate \( \mu_S < \mu_F \) at slow servers (for general service times, see Section IV-A), and define \( \tau = \frac{\mu_S}{\mu_F} \). The overall capacity of the system is \( \mu_F k_F + \mu_S k_S = 1 \). Each server has its own dedicated queue and processes the jobs in its queue in first-come-first-served (FCFS) order.

Jobs arrive to the system as a Poisson process with rate \( \lambda c \). Upon arrival, a job is dispatched to a single server according to some dispatching policy. We assume that the dispatcher has limited information, both about jobs and about servers: the dispatching decision can be based on queue length information (for which the dispatcher can query individual servers), but it cannot be based on any information about job sizes or server speeds. In particular, we consider the setting in which the dispatcher does not have any information about server speeds.

We consider several dispatching policies, beginning with two well-studied heterogeneity-unaware policies. Under Join-the-Shortest-Queue(d) (JSQ(d)), when a job arrives the dispatcher queries \( d \) servers uniformly at random. The job is dispatched to the queried server with the fewest jobs in its queue (including the job in service, if there is one). Under Join-Idle-Queue (JIQ), the dispatcher maintains a queue of idle servers. When a job arrives, it is dispatched to an idle server (in FIFO order) if there are any, and to a busy server chosen uniformly at random otherwise.

The dispatching policies that we propose are based on a novel heuristic, called accomplishment, that allows the dispatcher to use server heterogeneity information without knowing server speeds a priori or attempting to measure them.

Definition 1: A server \( i \)'s accomplishment at time \( t \), denoted by \( a_i(t) \), is the number of jobs that have been dispatched to server \( i \) by time \( t \). Server \( i \) is more accomplished than server \( j \) at time \( t \) if \( a_i(t) > a_j(t) \). When the time is clear from context, we will drop the \( t \) in our notation and write \( a_i \).

We use the idea of server accomplishment to augment several existing heterogeneity-aware dispatching policies. In our “A+” version of each policy, we replace server speed information with server accomplishment, making the “A+” policies suitable for settings where server speeds are unknown. In the sections that follow, we will review each of these existing policies and then define the accomplishment-based policy augmentations. Figure 1 gives a hierarchy of the dispatching policies studied in this paper.

III. PERFORMANCE EVALUATION

In this section, we describe how to apply the accomplishment heuristic to four heterogeneity-aware dispatching policies: Join-Idle-Queue, Balanced Routing, Shortest-Expected-Delay(d), and Join-the-Shortest-Queue(d_F, d_S). We define the “A+” version of each policy and compare its mean response time, \( E[T] \), to that of the baseline heterogeneity-unaware policy and the heterogeneity-aware “parent” policy. Throughout, we present simulation results for a system with \( k = 1000 \) servers and we set \( d = 4 \) for JSQ(d) and its descendants; 95% confidence intervals were all less than 2%.

A. Accomplishment+Join-Idle-Queue

We begin by incorporating the accomplishment heuristic into a heterogeneity-aware version of JIQ. Under all versions of JIQ, the dispatcher maintains a queue of idle servers. An arriving job is dispatched to an idle server if there is one, and to a server chosen uniformly at random if not. The versions differ in how ties are broken among multiple idle servers. Under Join-Idle-Queue-Known-Speeds (JIQ-KS), ties are broken in favor of the fastest idle server. Under Accomplishment+JIQ (A+JIQ), ties are broken in favor of the most accomplished idle server, thereby approximating JIQ-KS.

Figure 2 compares \( E[T] \) under JIQ, JIQ-KS, and A+JIQ. As \( \lambda \) increases, \( E[T] \) under JIQ—which breaks ties among
idle servers in FIFO order—improves. When arrivals are more frequent, a busy fast server is likely to return to the idle queue before a busy slow server. This means that fast servers are more frequently at head of the idle queue, so jobs are dispatched to fast servers disproportionately often, in turn decreasing $E[T]$. On the other hand, $E[T]$ under JIQ-KS remains constant as long as $k \lambda < k_F \mu_F$, i.e., if the fast servers have enough capacity to maintain stability without needing the slow servers. Once $\lambda$ is high enough that the slow servers are needed to ensure stability, $E[T]$ under JIQ-KS begins to increase. $E[T]$ under A+JIQ falls between that under JIQ and JIQ-KS at all values of $\lambda$. By sorting idle-queues by server accomplishment, A+JIQ provides a significant improvement in $E[T]$ compared to JIQ, even in the low $\lambda$, high $r$ setting where JIQ performs poorly. This result demonstrates that using the accomplishment heuristic can shrink the $E[T]$ gap between JIQ and JIQ-KS, despite the lack of server speed information.

Figure 3 compares the servers’ accomplishment levels after $2 \times 10^6$ arrivals. Under all three variants of JIQ, the fast servers consistently have higher accomplishments than the slow servers. Notably, the fast servers’ accomplishments are similar under JIQ-KS and A+JIQ, and are higher under these two policies than under baseline JIQ. This illustrates that the low response time achieved by heterogeneity-aware policies is driven by the proportion of jobs dispatched to fast servers.

B. Accomplishment+Balanced Routing

We next consider Balanced Routing, a heterogeneity-aware “descendant” of JSQ($d$). Under both the baseline and the A+ versions of Balanced Routing, when a job arrives the dispatcher queries $d$ servers and sends the job to the queried server with the fewest jobs in its queue (including the job in service, if there is one). The difference between the baseline and A+ versions lies in the probabilities with which the $d$ servers are selected. Under Balanced Routing (BR), servers are queried with probabilities proportional to their speeds: server $i$ is queried with probability $\frac{\mu_i}{\sum_{j=1}^{d} \mu_j}$.

Intuitively, under A+BR the query probabilities are determined by asking “how accomplished is this server compared to the expected accomplishment under random dispatching?” We expect faster servers to be more accomplished by time $t$ under BR than under random dispatching, thus faster servers tend to have higher estimated speeds. As a result, fast servers are queried more often. This in turn increases their accomplishment, creating a feedback loop in which more accomplished servers become even more likely to be queried in the future.

Figure 4 shows $E[T]$ under JSQ($d$), BR, and A+BR. As under the JIQ-based policies, when $r$ is low all three policies perform similarly because the system is relatively homogeneous. At high $r$, JSQ($d$) does not perform as well as BR and A+BR because it does not take advantage of the larger difference between fast and slow server speeds.

A+BR performs somewhat similarly to JSQ($d$) when $\lambda$ is low, and increasingly close to BR when $\lambda$ is high. At low $\lambda$ most queues tend to be idle, so joining the shortest queue among $d$ queried servers often amounts to random routing among those $d$ servers. In this case, all servers tend to have similar accomplishment values, and so A+BR makes similar querying and dispatching decisions to JSQ($d$). As $\lambda$ increases, both fast and slow servers are busier more often; because fast servers complete their assigned jobs more quickly than slow

\footnote{Note that in practice an adjustment is required to avoid division by 0 and to ensure that all servers have a nonzero probability of being queried. In our implementation we address this by setting $a_i(0) = 1$ for all servers $i$.}
servers, they in turn are dispatched more jobs. Indeed, at high $\lambda$ the fast servers are more accomplished than slow servers by approximately a factor of $r$: in this regime the accomplishment-based approximation of server speeds is quite accurate, and so A+BR and BR converge.

That A+BR performs as well as it does is perhaps surprising: unlike A+JIQ, which relies on the accomplishment heuristic only to estimate the ordering of servers by their speeds, A+BR uses accomplishment to estimate the actual speed of each server. The BR policy, then, is likely to be much more sensitive to errors in estimating $\mu$. Nonetheless, our results demonstrate that A+BR typically performs very similarly to BR.

### C. Accomplishment+Shortest-Expected-Delay($d$)

We now turn to Shortest-Expected-Delay($d$), another heterogeneity-aware “descendant” of JSQ($d$). Under both the baseline and A+ versions of SED($d$), when a job arrives the dispatcher queries $d$ servers chosen uniformly at random and computes the expected delay at each queried server $i$, denoted $\bar{D}_i$. The job is dispatched to the queried server $i$ with the smallest $\bar{D}_i$. The versions of SED($d$) differ in how $\bar{D}_i$ is computed. Under Shortest-Expected-Delay($d$), we set $\bar{D}_i = \frac{N_i + 1}{\mu_i}$, where $N_i$ denotes the number of jobs at server $i$. In order to calculate expected delay when speeds are unknown, we must estimate server speeds. Consistent with the server speed estimation used for Balanced Routing, we set $\mu_i^{\text{est}}(t) \equiv a_i(t)/a_{\text{RAND}}(t)$ for each server $i$. Under Accomplishment+SED($d$) (A+SED($d$)), we set $\bar{D}_i = \frac{N_i + 1}{\mu_i^{\text{est}}}$.

Surprisingly, as defined, A+SED($d$) yields higher $E[T]$ than JSQ($d$) (see Figure 5). Figure 6 illustrates why: A+SED($d$) frequently misestimates server speeds by a considerable margin. To understand why this misclassification occurs, observe that the dispatching decisions for the earliest arrivals significantly impact the dispatching decisions made in the future. Because all servers start with zero accomplishment, a server that is not assigned any jobs early on will have significantly lower accomplishment than its expectation under random. Thus, its estimated speed will be much slower than its actual speed.

We resolve this challenge with an “easing policy” that sets $\mu_i^{\text{est}} = 1$ for all servers for the first 20,000 arrivals, effectively running JSQ($d$) for this period. This ensures that all servers receive some jobs while also allowing the fast servers to become more accomplished than slow servers. The easing policy increases the accuracy of the $\mu_i^{\text{est}}$ values (see...
D. Accomplishment+Join-the-Shortest-Queue($d_F, d_S$)

Finally, we consider the JSQ($d_F, d_S$) policy, a descendant of JSQ-$d$ designed specifically for heterogeneous systems.

Under JSQ($d_F, d_S$), when a job arrives the dispatcher queries $d_F$ fast servers and $d_S$ slow servers, where $d_F + d_S = d$, a constant. If any of the queried fast servers are idle, the job is dispatched to an idle fast server. If all queried fast servers are busy and any of the queried slow servers are idle, the job is dispatched to an idle slow server with probability $p_S$ and to the queried fast server with the shortest queue with probability $1 - p_S$. If all queried servers are busy, the job is dispatched to the queried fast server (respectively, slow server) with the shortest queue with probability $p_F$ (respectively, $1 - p_F$). The probabilities $p_F$ and $p_S$ are chosen optimally to minimize $E[T]$, given system parameters $\mu_F, \mu_S, q_F, q_S,$ and $\lambda$.

JSQ($d_F, d_S$) is highly sensitive to the detailed server speed information: $p_F$ and $p_S$ are optimized based on $\mu_F, \mu_S, q_F,$ and $q_S$. Absent accurate knowledge of these system parameters, it is infeasible to optimize the policy parameters. We propose the A+JSQ($d_F, d_S$) policy, which uses server accomplishment to capture the spirit of JSQ($d_F, d_S$) without requiring detailed server speed information.

Under A+JSQ($d_F, d_S$), when a job arrives the dispatcher queries $d_F$ servers from among the $k$s most accomplished servers and $d_S$ servers from among the $k(1 - s)$ least accomplished servers, where $s$ is a policy parameter capturing the dispatcher’s estimate of $q_F$. The job is then dispatched to the queried server with the shortest queue.

We note that the performance of A+JSQ($d_F, d_S$) is sensitive to the choice of the policy parameter $s$. In this section we set $s = q_F$, which yields the best performance, thereby demonstrating the potential of A+JSQ($d_F, d_S$). Throughout this section, unless otherwise specified, we set $d_F = d_S = 2$.

Figures 7(a) and (b) compare $E[T]$ under A+JSQ($d_F, d_S$) to that under JSQ($d_F$) and JSQ($d_F, d_S$) at moderate to high $r$ and low to moderate $q_F$. At low $\lambda$, A+JSQ($d_F, d_S$) performs similarly to JSQ($d_F$). As $\lambda$ increases, $E[T]$ under A+JSQ($d_F, d_S$) actually decreases, eventually matching the performance of its fully heterogeneity-aware counterpart at higher load. The idea behind A+JSQ($d_F, d_S$) is that it is beneficial to guarantee that some fast servers are always included in the query. The accomplishment heuristic can only help achieve this if fast servers do in fact receive more jobs than slow servers. When $\lambda$ is very low, this does not occur because most servers are idle and thus receive similar numbers of jobs. As $\lambda$ increases, queues build up at all servers; the fast servers complete their jobs more quickly than the slow servers, so they tend to have shorter queues, and, in turn, they become more accomplished. Thus, at high $\lambda$, the accomplishment heuristic has the desired effect of successfully sorting servers by their speeds.
When \( r \) is low and \( q_F \) is high (Figure 7(c)), we see the opposite results: JSQ(d) in fact outperforms \( \text{JSQ}(d_F, d_S) \) and \( \text{A+JSQ}(d_F, d_S) \) at high load. This performance reversal is a consequence of the specific parameter settings used for \( \text{A+JSQ}(d_F, d_S) \). In particular, we have \( q_F = 0.8 \), \( d = 4 \), and \( d_F = 2 \), meaning that only half of the queried servers are fast even though 80% of the servers are fast; fast servers are underrepresented in the query. Figure 10 shows results for the same setting, using \( d_F = 3 \) instead of \( d_F = 2 \), i.e., the fast servers are queried with probability roughly proportional to the fraction of servers that are fast. Here we see that \( \text{A+JSQ}(d_F, d_S) \) consistently outperforms JSQ(d).

IV. EXTENSIONS AND GENERALIZATIONS

In this section we evaluate the performance of the accomplishment heuristic in more realistic settings. In the interest of brevity, we show results only for \( \text{A+JSQ}(d_F, d_S) \). The lessons that we learn from studying \( \text{A+JSQ}(d_F, d_S) \) in more general settings apply to all of the policies we introduce in Section III.

A. General Service Times

Thus far, we have assumed that service times are exponentially distributed. We now turn to general service times and evaluate the sensitivity of our results to service time variability.

We consider four different service time distributions. Given \( q_F \) and \( r \), for all four distributions the mean service time on fast (slow) servers is \( \frac{1}{\mu_F} \left( \frac{1}{\mu_S} \right) \), where \( \frac{\mu_F}{\mu_S} = r \) and \( \mu_F q_F + \mu_S q_S = 1 \) (see Section II). The first two distributions are the exponential (see Section III) and the two-phase Erlang. The remaining two distributions are two-phase hyperexponentials; the service time on the slow servers is drawn from the distribution \( H_2(p_1; \mu_1, \mu_2) \), where \( \frac{p_1}{\mu_1} + \frac{1-p_1}{\mu_2} = \frac{1}{\mu_S} \) and \( p_1 = \frac{1-p_1}{\mu_2} \) (the service time distribution on the fast servers is defined analogously). We consider two hyperexponentials with squared coefficients of variation \( C^2 = 10 \) and \( C^2 = 50 \).

In Figures 9(a) and (b) we see that \( \text{A+JSQ}(d_F, d_S) \) outperforms JSQ(d) by a similar margin regardless of the service time variability, for the same three system configurations considered throughout Section III. The opposite trend is evident in Figure 9(c): as in Section III-D, the fast servers are underrepresented in the query. As before, when we set \( d_F = 3 \), \( \text{A+JSQ}(d_F, d_S) \) consistently outperforms JSQ(d) (see Figure 10). Unsurprisingly, mean response time increases with service time variability for all system parameter settings.

Our results indicate that the strong performance of \( \text{A+JSQ}(d_F, d_S) \) is insensitive to the service time distribution. This suggests that our A+ policies remain good candidates for achieving low mean response time when server speeds are unknown in settings with highly variable service times, as is more realistic in many practical systems.

B. More Than Two Server Classes

So far we have restricted our attention to settings with only two server speeds; this may be unrealistic in practice. We now extend our results to allow for more than two server speeds.

We consider three different systems with four classes of servers, each with a different set of server speeds and distribution of servers among the four classes. In all cases, the average service rate is 1; hence, we consider servers with rate \( \mu > 1 \) to be “fast” and servers with rate \( \mu < 1 \) to be “slow.”

There are many ways in which one could generalize the \( \text{A+JSQ}(d_F, d_S) \) policy for more than two server speeds; we study one relatively simple such generalization, defined as follows. Upon each job’s arrival the dispatcher queries \( d \) servers, of which \( d_F \) are chosen from among the \( sk \) most accomplished servers and \( d_S \) are chosen from among the \( (1-s)k \) least accomplished servers. That is, even though there are more than two server classes, the dispatcher continues to classify servers into only two groups. Choosing \( s \) is challenging in
this setting, and, as noted in Section III-D, the choice of \( s \) can substantially impact the system’s performance. Our results in this section are for \( s \in \{0.1, 0.2, 0.3, 0.4\} \); when \( s \) is low, there are relatively few slow servers that are classified as fast.

A+JSQ(\( d_F, d_S \)) reduces \( E[T] \) relative to JSQ(\( d \)) for most values of \( s \) that we consider (see Figure 11; we omit JSQ(\( d_F, d_S \)), which is not well defined in this setting). JSQ(\( d \)) only outperforms A+JSQ(\( d_F, d_S \)) for \( s = 0.1 \). Here, the fastest server class contains more than \( sk \) servers, so A+JSQ(\( d_F, d_S \)) queries the fastest servers less frequently than the uniform querying used by JSQ(\( d \)). When \( s \) is higher A+JSQ(\( d_F, d_S \)) outperforms JSQ(\( d \)) because A+JSQ(\( d_F, d_S \)) queries the fastest servers more often than JSQ(\( d \)).

As defined, A+JSQ(\( d_F, d_S \)) is not optimal in the many-speed setting. Opportunities for further improvement include classifying servers into more than two speed categories, using accomplishment data (see Figure 3) to learn the appropriate cutoffs between server classes, and choosing which server classes to query probabilistically, as in [6]. Nonetheless, our results demonstrate that the accomplishment heuristic continues to be a useful tool in this more realistic setting.

C. Changing Server Speeds

In some systems, server speeds are unlikely to stay fixed over time. For example, migration of a VM could cause the VM’s speed to change at certain points in time. When server speeds change, the accomplishment-based predictions of relative server speeds—upon which our A+ policies rely—could suddenly become inaccurate. Here we evaluate the robustness of the accomplishment heuristic to changes in server speeds.

Figure 12 shows, as a function of the total number of arrivals to the system, the fraction of servers that are actually \{fast,slow\} and that are classified as \{fast,slow\}. We consider two scenarios in which each server’s speed changes with probability 0.4 after (1) relatively few arrivals, and (2) many arrivals. In the former setting only a short amount of time is required for the classification to return to near-perfect accuracy, whereas in the latter setting the recovery is slow. After many arrivals, the (initially) fast servers have far higher accomplishments than the slow servers (see, e.g., Figure 3); after the speeds change, many arrivals are needed before the newly-fast servers’ accomplishments can catch up to those of their initially-fast counterparts. The consequence of this slow recovery time is that mean response time suffers, as many jobs are dispatched to servers that are mistakenly classified as fast.

Motivated by this observation, we modify the A+ policies so that the dispatcher stores only a partial history of servers’ accomplishments. Specifically, we track each server’s accomplishment over only the most recent 40 000 arrivals, grouped in “buckets” of 10 000 arrivals; after each 10 000 arrivals to the system, we discard the information in each server’s oldest “bucket.” Thus, out-of-date information collected prior to server speed changes will be discarded quickly.

We evaluated the performance of the partial-history accomplishment for a wide range of parameter settings; Figure 13 shows the results for a few representative cases. The classification accuracy depends on \( q_F, \lambda, r \), and the probability with which speeds change, with higher accuracy associated with higher values of both \( \lambda \) and \( r \), lower probabilities of changing speeds, and values of \( q_F \) close to 0.5. Importantly, regardless of the system parameters, the time to recover near-perfect server classification no longer depends on the number of arrivals before the server speeds changed, indicating that tracking only a short accomplishment history is a successful strategy for maintaining good performance in this setting.
Fig. 13. Fraction of servers that are actually fast and classified fast (light purple), actually slow and classified fast (dark purple), actually fast and classified slow (light green), and actually slow and classified slow (dark green) with partial-history accomplishment. In all cases, $r = 1.5$, $\lambda = 0.5$, and server speeds change with probability $0.4$. Top: $q_F = 0.2$; bottom: $q_F = 0.5$. The speed changes occur after 50,000 arrivals (left) or 300,000 arrivals (right).

V. Conclusion

This paper studies dispatching in large-scale, heterogeneous systems in which the dispatcher has limited information about server speeds. We propose a novel heuristic called server accomplishment that allows us to estimate server speeds in this setting. We show that the accomplishment heuristic can be incorporated into many state-of-the-art dispatching policies designed for systems in which server speeds are known, including Join-Idle-Queue, Shortest-Expected-Delay($d$), Balanced Routing, and Join-the-Shortest-Queue($d_F, d_S$). Our numerical results demonstrate that our “A+” accomplishment-based variants often are capable of closing most of the performance gap between heterogeneity-unaware policies and heterogeneity-aware policies with full speed information.

Tracking server accomplishment is one way to incorporate memory in a dispatching policy. Other work involving memory-based dispatching includes [1], [11], [15]. These papers focus on homogeneous systems, and memory is typically used to identify servers that have short queues rather than to identify fast servers. Consequently, the dispatching policies proposed in the above papers all represent candidates for use in conjunction with our accomplishment heuristic.

While most of our results focus on systems with two server speeds, exponential service times, and server speeds that are fixed over time, these assumptions are not always reasonable in practice. We show that, even when these assumptions are relaxed, the accomplishment heuristic provides a valuable tool for improving performance in the heterogeneity-unaware settings. There are ample opportunities for future work addressing practical concerns. For example, our results in Section IV-B are for just one, fairly simple, generalization of the A+JSQ($d_F, d_S$) policy when there are more than two server speeds; other approaches may yield even better performance.

Another consideration of practical importance is the communication budget available to the dispatcher. Each of our “A+” policy variants requires no more communication between the dispatcher and the servers than the baseline policy upon which it is based. This gives the A+ heuristic an advantage over, for example, policies that try to directly learn the server speeds, as such policies would require more frequent communication between the dispatcher and the servers. However, there has been considerable recent work focusing on settings in which communication is limited to far fewer than the $O(d)$ messages per job required by power-of-$d$ policies. Another direction for future work involves developing A+ versions of policies that have a severely restricted communication budget.

REFERENCES

[16] N. Vvedenskaya, R. Dobrushin, and F. Karpelevich. Queueing system in which communication is limited to far fewer than the $O(d)$ messages per job required by power-of-$d$ policies. Another direction for future work involves developing A+ versions of policies that have a severely restricted communication budget.

Another consideration of practical importance is the communication budget available to the dispatcher. Each of our “A+” policy variants requires no more communication between the dispatcher and the servers than the baseline policy upon which it is based. This gives the A+ heuristic an advantage over, for example, policies that try to directly learn the server speeds, as such policies would require more frequent communication between the dispatcher and the servers. However, there has been considerable recent work focusing on settings in which communication is limited to far fewer than the $O(d)$ messages per job required by power-of-$d$ policies. Another direction for future work involves developing A+ versions of policies that have a severely restricted communication budget.
OLTP In Real Life: A Large-scale Study of Database Behavior in Modern Online Retail

Nosayba El-Sayed  
Emory University  
nosayba.ae@emory.edu

Zhuoran Sun†  
Truist Securities  
zhuoran.sun@truist.com

Ke Sun†  
Carnegie Mellon University  
kes2@andrew.cmu.edu

Ricardo Mayerhofer  
Hopara Inc.  
rmayerhofer@hopara.io

Abstract—E-commerce is a multi-trillion US dollar industry and a major user of online transaction processing (OLTP) systems. Yet, we do not have a good understanding of the real-world properties that characterize hardware usage in database systems serving online retail. One key reason is the difficulty of obtaining monitoring logs from production retail datacenters, as large retailers typically consider such data to be highly sensitive. Additionally, resource-access patterns in OLTP systems and their correlations with the workload are generally challenging to model (not just in retail), due to high levels of transaction concurrency and complexity in OLTP applications. Relying on synthetic benchmarks alone to simulate and study these systems is not sufficient. Instead, this paper takes an empirical approach.

We present the first large-scale study of OLTP systems in real life, using traces from two large retailers. The first trace represents six months worth of logs from a large online retailer in Latin America, B2W Digital (B2W). The second trace represents 8 days of hardware logs from Alibaba, the largest online retailer in China. Our analysis reveals important characteristics of real systems serving retail. First, we identify the statistical distributions that best fit hardware-usage data and find that system behavior in modern retail workloads departs from certain conventional assumptions in the literature, particularly the assumption that resource utilization follows a ‘heavy-tailed’ distribution. Next, we examine how these resource-access patterns change with different retail seasons and correlate with workload demand and transaction type. Finally, we discuss practical implications for OLTP systems management, specifically in hardware provisioning and capacity planning.

Index Terms—OLTP, e-commerce, databases, system measurement, distribution fitting

I. INTRODUCTION AND RELATED WORK

Online shopping has been growing at unprecedented rates in recent years (and more recently due to the health pandemic declared in the year 2020 [1], [2]), making e-commerce a multi-trillion US dollar industry [2] and a major user of online transaction processing (OLTP) systems. Efficiently running OLTP systems in this growing sector relies critically on developing an in-depth understanding of how these systems are used "in the wild".

What characterizes resource-access patterns in database systems serving online retail? How do these characteristics change with different retail seasons? And how do different workload properties correlate with hardware-usage patterns? These are examples of questions that can improve the design and automation of resource management strategies in these platforms. However, investigating these questions for retail workloads faces two key challenges. First, characterizing resource-access patterns in OLTP applications in general (not just retail) is tricky as these workloads exhibit high levels of transaction concurrency and complexity [3]–[5]. Relying on benchmarks and hypothetical models alone to capture these complexities is not sufficient. Instead, analyzing real-world logs from production systems is crucial for accurately characterizing their behavior. This brings us to the second challenge, which is obtaining system logs from large retailers. We discuss each challenge next.

Accurate OLTP resource modeling: Characterizing the impact of workload changes on hardware-usage patterns is difficult in OLTP environments where many transactions are highly concurrent and tend to perform random I/O reads and writes [3]–[5]. For example, disk I/O in OLTP workloads grows non-linearly with system load and is challenging to predict [3], [4]. One common approach to modeling such complex OLTP interactions has been to execute benchmarks that simulate a target workload, then monitor the behavior of the underlying hardware. To make benchmarks realistic, we have seen more studies in recent years rely on real traces to drive their simulations [3], [5]–[7]. However, relying on experiments only to replay database transactions and subsequently characterize database behavior is not sufficient. Configuring benchmarks usually requires making simplistic assumptions that do not necessarily reflect the true complexity and messiness of real life. Additionally, benchmarks are typically used to simulate short time intervals (e.g., hours or days), and thus do not always capture long-term trends in real systems.

Another limitation in prior work on OLTP resource modeling is that it has not focused on studying an important class of workloads that represents a multi-trillion US dollar industry: modern online retail. TPC-W [8] is a widely known web benchmark designed to represent online retail operations, but it merely simulates a fictional online bookstore from the early 2000s. Similarly, the Dell DVDStore benchmark [9] simulates an online store that can be used for ordering DVDs. Neither benchmark represents modern, large-scale online retailers who sell various categories of items to millions of customers.

Access to e-commerce system logs: Understanding how OLTP systems are used in modern retail requires large-scale empirical studies using real logs collected over long intervals to capture different retail seasons. Additionally, these logs should include both application- and hardware-level data, to enable conducting a comprehensive analysis of

†Work done while student at Emory University.
the impact of workload properties on hardware utilization. Obtaining such logs from large retailers, however, is difficult as these companies are typically very protective of their data. Recently, in a promising move, the large e-commerce company Alibaba released two resource-usage traces from a production cluster that co-locates online services with offline batch jobs [10], [11]. However, Alibaba’s data includes hardware-level logs only and lacks application traces or transaction logs. Additionally, it spans short intervals: the 2017 trace spans 12 hours and the 2018 trace spans 8 days.

This study tackles the two challenges described above, by analyzing a large-scale, rich dataset collected over six months in the year 2016 from various subsystems in one of the top online retailers in Latin America who shared the data with us: B2W Digital (B2W) [12]. We use B2W’s data to characterize different short- and long-term patterns in system behavior in an online retail environment. To our best knowledge, this is the first large-scale empirical study of OLTP logs collected from a modern, large online retailer, spanning almost half a year and involving both hardware traces and transaction logs.

Furthermore, to better understand the generalizability of our findings from B2W’s data, we analyze Alibaba’s hardware trace as well and compare our observations on resource-usage patterns across these two large clusters serving online retail. Our contributions can be summarized as follows:

- We present a detailed analysis of hardware-usage patterns in two large online retail clusters. We identify which probability distributions are best fit to model various resources. Unlike findings from early studies [13]–[15], we find that the heavy-tailed Pareto distribution does not provide a good fit for all resource traces in our study, and that the gamma distribution is a good fit for modeling network traffic and disk I/O usage—both in B2W and Alibaba’s production clusters.

- We use B2W’s data to study long-term patterns in OLTP system usage and examine how they change in different retail seasons. We study correlations between hardware usage and workload properties (e.g., transaction type), and highlight patterns that distinguish real-world workloads from synthetic benchmarks. We further identify correlations between popular retail transaction types (e.g., shopping cart purchases) and disk I/O usage patterns.

- Based on our observations, we discuss practical implications for OLTP systems management. E.g., knowledge of resource-usage distributions can drive provisioning decisions; uncovering long-term trends in hardware patterns can improve capacity planning for different seasons; and understanding correlations between certain transaction types and resource patterns can lead to more precise workload classifiers.

II. OVERVIEW OF THE B2W DATA

B2W [12] is one of the top online retailers in Latin America, serving tens of millions of customers. Between July and December 2016, they logged and stored monitoring data from various parts of their database operations. They shared this dataset with a group of scholars with the goal of identifying opportunities for improving their resource-management optimization and automation strategies (the data is not currently publicly available and accessing it requires B2W approval). Their infrastructure is split between in-house and cloud resources. The collected dataset is large and rich, as it includes logs on their shopping transactions, various database dumps, and hardware monitoring data. We describe each trace next.

Transaction logs: These include records of every transaction executed on their shopping cart, checkout, and stock inventory databases. The timestamp and the type of each transaction are recorded. There are traces of 38 million unique shopping carts in the data.

Database dumps: These are dumps of database tables that contain important information on all the carts, checkouts, and stock items accessed during the observation period. This includes for example: the final status of each cart (e.g., PURCHASED, CANCELED, EXPIRED), the timestamps indicating the beginning and end of a checkout process, and more. This data has been completely anonymized by B2W to avoid exposing any sensitive information.

Hardware logs: This trace contains hardware monitoring data that was logged periodically (every 5 minutes), from the database servers. The data contains aggregate statistics computed over each monitoring period, including the average and sum values for each performance metric. The logged metrics include: CPU utilization, disk read/write bytes, disk queue length, and inbound/outbound network traffic.

III. CHARACTERIZING HARDWARE USAGE PATTERNS IN ONLINE RETAIL

We begin by examining how different hardware resources were used in B2W. More specifically, we study the questions: What characterizes the empirical distributions of hardware data in modern retail? And how does hardware usage vary over long periods that span different retail seasons?

A. What characterizes the empirical distributions of hardware-usage data?

Motivation: Our first goal is to identify the statistical distributions that best fit the hardware-usage data in B2W. Fitting empirical logs of resource data to known distributions (e.g., Weibull, Pareto, etc.), enables leveraging these distributions’ well-studied properties when managing hardware resources. For example, if network traffic in an Internet application is found to follow a heavy-tailed distribution (e.g., Pareto), this points to the presence of infrequent large requests that require more resources than the typical request. This observation can be used to guide hardware replication and provisioning strategies designed to meet QoS goals [16]–[19].

Additionally, identifying these best-fits helps system researchers understand which probability distributions should be used, or avoided, when generating hardware traffic data (e.g., network or disk I/O requests) to simulate the behavior of modern OLTP workloads. This is particularly important for guiding the design and parameterization of new benchmarks that represent modern retail environments accurately.
Methodology: We analyze the empirical cumulative distribution function (CDF) of CPU, network, and disk utilization data. In particular, we study how well each CDF is fit by six widely used probability distributions: the exponential distribution; Weibull distribution; gamma distribution; normal distribution; lognormal distribution; and the heavy-tailed Pareto distribution. We parametrize each distribution using maximum likelihood estimation.

To evaluate and compare the goodness of fit, we measure two metrics: Kolmogorov-Smirnov (KS), which calculates the maximum distance between the empirical and fitted CDFs [20], and Anderson-Darling (AD), which captures a weighted measure of the area between the empirical and fitted CDFs [21]. Note that we consider more than one metric to reduce the bias that some distributions may exhibit with a specific metric [21], [22].

Disk I/O results: We begin by examining these aggregate disk I/O metrics: write throughput, read throughput, and disk queue length. Figure 1 shows the results for fitting these metrics using the month of August’s data in B2W. Note that the absolute values have been normalized for read/write data (by the minimum recorded value in each metric), for B2W confidentiality reasons, but this does not have any impact on our curve-fitting outcomes. The title of each plot in Figure 1 indicates which distribution is the best fit. In all of the fitted CDFs in our dataset, both KS and AD metrics agreed on which distribution is the best fit; in very few cases, their selection of the second (or third) best fit had differed.

We observe from Figure 1 that for disk write throughput, the normal and Weibull distributions equally provide the best fitted curves to the data (followed by the gamma distribution). For disk read throughput, the gamma distribution is the best fit, followed by a normal distribution. The exponential and Pareto distributions performed significantly worse than others when fitting both disk read and write data, which can be easily observed by visual inspection as well in Figure 1. For disk queue length, we find that the Weibull distribution provides the best fit, followed by gamma, while the normal distribution provides the worst fit.

One key takeaway from these results is that disk I/O usage in B2W is not well-modeled by heavy-tailed distributions. Note that the heavy-tailed Pareto distribution, for instance, was identified in prior work as a good fit for hardware resource usage, e.g., in a storage cluster’s I/O bandwidth [23] and in online retail traffic (from the early 2000s) [13]–[15]. For B2W’s I/O data, we do not find evidence of heavy tails; even for metrics that are well-modeled by a Weibull distribution, we find that the fitted Weibull shape \( \beta \) parameter is 2.98 for disk writes and 1.1 for disk queue length (Weibull is considered heavy-tailed only for \( \beta < 1 \)).

The presence (or lack) of heavy tails in hardware-usage data is important to characterize since it can be used to drive hardware over-provisioning policies (we discuss more practical implications of these results in subsection III-D).

Network results: Network traffic is also critical to model; e.g., network bandwidth is increasingly becoming a potential bottleneck in cloud databases, where disk I/O is highly likely to go through the network [24], [25]. Figure 2 plots the fitted curves for both inbound and outbound traffic logged during August. We find that the lognormal and gamma distributions provide comparable best-fits to both inbound and outbound data, with lognormal being slightly better for...
outbound data. Similar to disk I/O data, the worst fits were found to be the Pareto and exponential distributions.

Note that the log-normality in node network traffic has been observed in prior empirical studies [18], [26], [27]. It is generally explained by the presence of self-similar processes whose logarithm is normally distributed and exhibits a lighter tail than that of a Pareto or a Gaussian distribution, for instance [26], [28], [29].

**CPU results:** We repeated the curve-fitting analysis for CPU utilization data (not plotted here), and identified the log-normal distribution as a best-fit for B2W’s CPU data, with Pareto and exponential distributions again being the worst fits.

**B. How does hardware usage vary over long periods of time?**

The statistical distributions identified so far to model hardware resources were fitted using B2W’s August logs. But what about the rest of the year? Can these models be used to represent the company’s hardware usage throughout the year, or will seasonal effects impact how representative the models are? Fortunately, the long observation period in B2W’s data enables investigating these questions empirically. We focus our analysis here on disk I/O operations since they are the most challenging to model in highly-concurrent OLTP databases; e.g., disk writes are caused by a combination of log writes, buffer pool misses, and dirty pages write-backs and thus grow non-linearly with the load, making deriving their performance models more complex than linear resources like CPU utilization or network traffic [3]–[5].

Figure 3 compares the empirical distributions of disk-usage data among the months available in the traces (we exclude July’s logs here since hardware monitoring in B2W was not very stable then). The first thing we observe by visual inspection is that the general form of the curves from different months look similar—the variations that exist seem more scale related. To formalize this observation, we repeated the curve-fitting analysis performed in the previous subsection (for August), for each month separately. We then examined if the best-fitted distribution had changed from month to month. We found that the answer is ‘no’ for the vast majority of the curves here, where the best-fitted distribution did not change over time. The one exception is disk queue length during November, where the gamma distribution provided a slightly better fit than the Weibull distribution (the best fit for all remaining months).

Although the best models remained stable over time, we did find that the values of their parameters changed from month to month. For example, and most notably, we find that the shape parameter $\beta$ of the Weibull distribution that best fits disk write throughput data varied from 2.98 in August to 3.68 in October, and finally to 11.7 in December. For disk queue length, the observed Weibull shape $\beta$ changes are weaker: from 1.08 in August to 1.06 in October, and then almost doubled to 2.11 in December. It is worth noting that since all of the observed $\beta$ values here are larger than 1, the general form of the Weibull density function remains the same $^1$, as well as the lack of heavy tails.

Figure 3 also shows that disk I/O rates increased significantly during November and December, compared to the rest of the months. This is intuitive for online retailers since these months include major shopping campaigns around Black Friday and Christmas (even in Latin America), where customers spend more time browsing items (i.e., more disk reads), and purchasing items (i.e., more disk writes). For example, disk writes increased in November and December by 24% and 30%, compared to August, and the median disk queue length grew by 1.96X and 2.48X (in the same months). We discuss practical implications of these results in subsection III-D.

**C. Examining the generalizability of observed hardware patterns in B2W: Alibaba’s trace analysis**

Can the observed hardware-usage distributions in B2W be generalized to other similar environments? To address this question, we looked for similar traces from large online retailers. To this date, the only hardware trace available from a large retailer is from a production cluster in Alibaba, the largest e-commerce company in China [11]. Alibaba’s

---

1. $\beta = 1$ reduces the Weibull distribution to an exponential distribution, and for $0 < \beta < 1$ the density function form becomes strictly decreasing.
trace contains hardware monitoring logs for their online e-commerce production jobs, which ran in containers, collected over a period of 8 days in 2018. An older version of the trace from 2017 spans 12 hours only [10]; we focus here on analyzing the longer, more representative 8-day trace.

The usage files in Alibaba’s trace include for each container 5-minute samples of its CPU utilization, normalized network traffic, and disk-space utilization (as a percentage). We repeat the curve-fitting analysis performed for B2W (section III) on Alibaba’s hardware data. Table I summarizes the best and worst fits identified for each hardware metric in Alibaba’s trace. We also show the quality of the Pareto fit, specifically, to compare it against findings from B2W’s data and from early studies on retail benchmarks. Detailed CDF plots are omitted for space purposes.

### TABLE I: The best and worst probability distribution fits to Alibaba’s hardware data, as well as the quality of the Pareto fit. Recall that KS refers to the Kolmogorov-Smirnov statistic which measures the quality of the fit (smaller is better).

<table>
<thead>
<tr>
<th>Resource</th>
<th>Best Fit (KS)</th>
<th>Worst Fit (KS)</th>
<th>Pareto Fit’s KS</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU Util.</td>
<td>Normal (0.065)</td>
<td>Lognormal (0.294)</td>
<td>0.233</td>
</tr>
<tr>
<td>Network-In</td>
<td>Gamma (0.048)</td>
<td>Normal (0.174)</td>
<td>0.083</td>
</tr>
<tr>
<td>Network-Out</td>
<td>Gamma, Weibull (tie: 0.047)</td>
<td>Normal (0.164)</td>
<td>0.082</td>
</tr>
<tr>
<td>Disk Util.</td>
<td>Gamma (0.083)</td>
<td>Pareto, Exp. (tie: 0.190)</td>
<td>0.190</td>
</tr>
</tbody>
</table>

Our first observation from Table I is that, similar to B2W, the heavy-tailed Pareto distribution was not identified as the best fit for modeling any resource in Alibaba’s cluster. In fact, Pareto was the worst and second-worst fit for disk and CPU utilization data, respectively, and more than 73% worse than the best fit’s performance (in terms of KS score) for inbound/outbound network traffic, in Alibaba’s cluster.

Additionally, where the Weibull distribution provided a good fit, i.e., for outbound network traffic, we found its shape parameter to be 1.1, which again indicates lack of heavy tails. Finally, we find that the gamma distribution provides a good fit for modeling all network traffic data in Alibaba—another similarity with B2W. We discuss some practical implications of these results next. (Note that Alibaba’s trace covers 8 days only, so we are not able to rerun the multi-month analysis performed for B2W in subsection III-B.)

### D. Hardware-usage distributions in online retail: Key observations and practical implications

We now present the key observations from our hardware-usage analysis so far for B2W and Alibaba’s clusters, and discuss some of their practical implications.

**Observation 1:** The heavy-tailed Pareto distribution is not found to be a good fit for modeling the usage of all hardware resources examined in B2W’s data (subsection III-A), as well as in Alibaba’s cluster (subsection III-C).

**Implications:** This observation contradicts characterization studies done in the early 2000s on an online bookstore workload [13]–[15], where a heavy-tailed Pareto distribution best fitted the data. These past studies explained the heavy tail property by the presence of ‘robot’ agents, like price comparison engines, which used e-commerce websites differently than regular customers and skewed resource utilization toward a heavy-tailed distribution. We do not observe similar trends in the two real-world datasets we examined. Even for resources that are well-modeled by a Weibull distribution (e.g., disk write throughput and queue length in B2W and outgoing network traffic in Alibaba), we find that the Weibull shape parameter was greater than 1, and hence does not reflect a heavy tail.

This suggests that the patterns observed in old retail workloads [13]–[15] do not necessarily hold in modern retail environments that B2W and Alibaba represent. The drastic changes in the world of online shopping and the infrastructure serving it, which happened mostly over the past two decades, is likely to have impacted resource utilization patterns in these environments significantly. This observation further warrants the need for new, realistic benchmarks that accurately represent modern retail workloads.

Additionally, the lack of heavy tails in resource-usage distributions (observed in both B2W and Alibaba) has important implications for capacity planning purposes, since the presence of heavy tails is commonly used to drive hardware over-provisioning and replication policies [16]–[19], [30], [31]. Our findings suggest that using fitted models based on real-time monitoring logs to drive provisioning decisions would be more effective than relying on classical provisioning strategies derived from possibly outdated assumptions on resource usage. Prior work on Internet network link provisioning, for instance, demonstrated the benefits of using fitted statistical models to mitigate under- or over-provisioning network bandwidth [26]; our results here support deploying such data-driven techniques in OLTP hardware management.

**Observation 2:** The gamma distribution is identified as either the best or second-best fitted model for network and disk I/O usage data, in both of B2W and Alibaba’s clusters.

**Implications:** Gamma is a flexible, two-parameter distribution used to model positively skewed variables. It is widely used and easy to interpret. When identified as a good fit for a resource, gamma’s shape parameter $\alpha$ can be used to directly assess the degree of skewness of this resource’s usage, where skew equals $2/\sqrt{\alpha}$. For example, Table II displays the calculated skew for network traffic in B2W and Alibaba, using the fitted gamma shapes. We find that Alibaba’s cluster exhibited higher network traffic skew than B2W (data with skew > 1 is considered ‘highly’ skewed).

### TABLE II: The fitted gamma’s shape $\alpha$ parameter and skew ($2/\sqrt{\alpha}$) for B2W and Alibaba’s network traffic data.

<table>
<thead>
<tr>
<th></th>
<th>B2W’s Data</th>
<th>Alibaba’s Data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Gamma Shape $\alpha$</td>
<td>Skew</td>
</tr>
<tr>
<td>Network-In</td>
<td>23.688</td>
<td>0.411</td>
</tr>
<tr>
<td>Network-Out</td>
<td>15.115</td>
<td>0.515</td>
</tr>
</tbody>
</table>

5
IV. UNDERSTANDING HARDWARE-USAGE CORRELATIONS WITH WORKLOAD PROPERTIES

So far we have been analyzing hardware-usage patterns in retail workloads. Now we turn our attention to studying how this hardware usage correlates with two specific workload factors: the rate of requests and the type of requests. Understanding how these workload properties impact hardware utilization is critical for building realistic performance models and in answering ‘what-if’ questions on database management and provisioning. Fortunately, B2W’s rich dataset allows us to explore these questions since it includes both hardware traces and database transaction logs.

a) Request rate vs. hardware utilization: In OLTP environments, CPU and network resources are believed to grow linearly with system load and are relatively easy to predict. Disk I/O, on the other hand, grows non-linearly with system load and is thus trickier to model. Prior work [3], [4] has identified two key reasons for this: i) the DBMS flushes dirty pages during low disk utilization periods, and ii) the I/O throughput grows sub-linearly with workload size due to update operations hitting the same page at higher rates.

To verify if these patterns hold in a production environment, we analyzed the relationship between the average system load and the measured hardware metrics during the same monitoring interval, in B2W. In Figure 4, we plot the results for disk I/O metrics and outbound network traffic (inbound network plot is very similar and is thus omitted), during August. Note that similar results are observed in the remaining months in the data.

Each scatter plot shows lines for linear, quadratic, and cubic functions fitted to the data, as well as the values for two computed correlation coefficients: Pearson’s coefficient captures the strength of the linear relationship between the variables, while Spearman’s coefficient is rank-based and makes no assumptions about the data distribution, thus is better at capturing the strength of highly nonlinear relationships.

The first thing we observe by visual inspection of Figure 4-(a) is that disk write throughput did increase non-linearly with the load. More formally, we find that the quadratic fit (adjusted R-squared=0.6) is significantly better than the linear fit (adjusted R-squared=0.4); the cubic fit offers further
improvement, explaining 8% more of the data compared to
the quadratic fit (adjusted R-squared=0.68). Disk writes also
reported the highest correlation coefficients with the load, with
a particularly high Spearman coefficient of 0.839. The second-
highest Spearman correlation was 0.768, between disk queue
length and the load. All correlations reported in this section
are statistically significant (p-value < 2.2e-16).

Disk reads, on the other hand, exhibit different and some-
what more linear trends in relation to the load, as seen in
Figure 4-(b). However, we find that there were intervals when
the load was not particularly high yet disk reads spiked.
The same pattern can be observed with disk queue length in
Figure 4-(c) and with network traffic in Figure 4-(d), although
less so in the latter. We asked B2W about these periods of high
disk I/O activities that do not correlate directly with the load,
and they said these are due to additional database optimization
tasks that were intentionally enabled during periods of low
traffic.

Finally, from Figure 4-(d) we can see that network traffic
activity grew linearly with the load. The adjusted R-squared
measure improved only from 0.19 in the linear fit to 0.24 in
both the quadratic and cubic fits; i.e., no further improvement
is gained by increasing the degree of the polynomial fit. A
similar trend was observed for CPU utilization (not displayed).

b) Type of request vs. hardware utilization: In OLTP
workloads, certain types of requests tend to occur frequently
and share similar characteristics. In an online retail envi-
ronment, users browse items, add items to shopping carts,
purchase items, and so on. These common operations can
be mapped to certain database transaction ‘profiles’ that have
similar properties and hardware-usage patterns. For example,
while “browsing” or “searching” for items may trigger disk
read-intensive operations, “purchasing” an item is likely to be
disk write-intensive, since it involves updating stock inventory
database tables.

In B2W’s transaction logs, information on the exact type of
request invoked is available. Instead of studying the relation-
ship between hardware usage and all types of requests (like we
did in Figure 4), we now ask the question of how certain types
of requests are correlated with hardware usage, and specifically
with disk I/O usage patterns. Table III studies this question for
some popular request types in B2W’s workload.

TABLE III: Correlation between disk I/O metrics and the
observed rate of certain types of retail transactions in B2W.
R indicates Pearson’s coefficient and S indicates Spearman’s
rank coefficient. All correlations were found to be statistically
significant with p-value < 2.2e-16.

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>Disk Write Throughput</th>
<th>Disk Read Throughput</th>
<th>Disk Queue Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET_CART</td>
<td>0.777 0.849</td>
<td>0.486 0.693</td>
<td>0.549 0.778</td>
</tr>
<tr>
<td>GET_CHECKOUT</td>
<td>0.713 0.840</td>
<td>0.444 0.683</td>
<td>0.510 0.763</td>
</tr>
<tr>
<td>PURCHASE_STK_TXN</td>
<td>0.802 0.849</td>
<td>0.514 0.694</td>
<td>0.572 0.780</td>
</tr>
<tr>
<td>CANCEL_STK_TXN</td>
<td>0.810 0.850</td>
<td>0.512 0.693</td>
<td>0.555 0.778</td>
</tr>
<tr>
<td>EXPIRE_STK_TXN</td>
<td>0.473 0.840</td>
<td>0.295 0.689</td>
<td>0.397 0.775</td>
</tr>
</tbody>
</table>

Operations GET_CART and GET_CHECKOUT retrieve
information on carts and checkouts, respectively, when needed; PURCHASE_STK_TXN refers to completing a checkout
by ‘purchasing’ the cart; CANCEL_STK_TXN refers to completing a checkout by ‘canceling’ the cart; and
EXPIRE_STK_TXN refers to the system expiring a checkout
process if the user is idle for a certain amount of time.

Table III shows that the strongest correlations observed
(according to both Pearson’s and Spearman’s coefficients)
are between the rate of purchase or cancel operations and
the aggregate disk write throughput. This is intuitive since
purchasing or canceling a shopping cart entails updating cart,
checkout, and stock inventory tables. The weakest correla-
tions observed are between expire operations and all disk
I/O metrics (in our preliminary analysis of this dataset we
found that cart expirations correlate with late night hours,
which may explain why the corresponding disk I/O rates are
low). We also find that the disk queue length exhibits the
strongest correlation with purchase operations, followed by
cancellations.

A. Hardware correlations with retail workload properties: Key
observations and practical implications

Observation 4: Disk writes in retail grow non-linearly with
the load and correlate strongly with certain transaction
types: purchases and cancellations. Patterns that distin-
guish real-world workloads from synthetic benchmarks
include increased disk I/O rates and network activity
during maintenance and cleanup events, which tend to
occur during low-demand periods.

Implications: The analysis of hardware data and application
transaction logs jointly from a large-scale production retail
environment has not been done before. The insights learned
here have two key implications for real-world systems.

First, we are able to verify, empirically, existing assumptions
about how different hardware components are utilized as a
function of system load in OLTP platforms. But we highlight
how real-world hardware does not exhibit as 'smooth' growth
rates as typically assumed in analytical models. In practice, it
is important to account for real-life activities (e.g., database
optimization or maintenance events) when attempting to model
hardware usage, for purposes like answering ‘what-if’ ques-
tions on provisioning these resources or when designing fore-
casting models to predict future resource demand.

Secondly, our observation on how different shopping trans-
action types correlate strongly with certain disk I/O metrics
can be directly utilized by transaction-type classifiers in OLTP
workloads. For example, if an on-going shopping cart is
predicted to end up with a ‘purchase’ action, knowing which
data stores are expected to be accessed and how (i.e., if the
action will trigger disk read- or write-intensive operations),
can enable more fine-grained decisions on database load balancing
and data migration [3], [5], [37], [38]. The design of next-
generation autonomous database systems will rely critically on
having such fine-grained forecasting and classification models.
V. Conclusion

This paper presents the first large-scale empirical study of OLTP system behavior in modern online retail. Our analysis of real traces from two large retail enterprises reveals which probability distributions are best fit to model different resources, and highlights the lack of heavy tails in resource-usage distributions. This critical observation departs from findings in past studies that relied on historic, synthetic benchmarks, further warranting the need for new benchmarks that accurately represent the world of modern e-commerce. We also examined long-term patterns in system usage and identified important correlations with workload demand and retail transaction type. We discussed practical implications for real OLTP systems, e.g., in hardware provisioning and capacity planning. We hope this study encourages more large retailers to share similar datasets with researchers or publicly, given the growing importance of this multi-trillion dollar industry.

References

Scaling Up The Performance of Distributed Key-Value Stores With In-Switch Coordination

Hebatalla Eldakiky, and David Hung-Chang Du
Department of Computer Science and Engineering
University of Minnesota–Twin Cities, USA
E-mail: {eldak002, du}@umn.edu

Abstract—The power and flexibility of software-defined networks lead to a programmable network infrastructure in which in-network computation can help accelerating the performance of applications. This can be achieved by offloading some computational tasks to the network. In this paper, we propose TurboKV, an efficient distributed key-value store architecture that utilizes programmable switches as: 1) partition management nodes to store the key-value store partitions and replicas information; and 2) monitoring stations to measure and balance the load among storage nodes. We also propose a key-based routing protocol to route the queries of clients based on the requested keys to targeted storage nodes. Our experimental results of an initial prototype show that our proposed architecture improves the throughput and reduces the latency of distributed key-value stores when compared to the existing architectures.

I. INTRODUCTION

Programmable switches in software-defined network promise flexibility and high throughput. Recently, the Programming Protocol-Independent Packet Processor (P4) [1] unleashes capabilities that give the freedom to create intelligent network nodes performing various functions. Thus, applications can boost their performance by offloading part of their computational tasks to be executed in the network. Nowadays, programmable networks get a bigger foot in the data center doors. Google started to use the programmable switches to build and control their smart networks [2]. Also, some Internet Service Providers (ISPs) [3] have already integrated programmable switches in their networks.

Recently, there has been an uptake in leveraging programmable switches to improve distributed systems, e.g., [4]–[11]. This uptake is due to the massive evolution on the capabilities of these network switches, e.g., Tofino and Tofino2 ASICs from Intel [12] with sub-microsecond per-packet processing delay and up to 12.8 Tbs bandwidth. These systems use the switches to provide orders of magnitude higher throughput than the traditional server-based solutions.

Meanwhile, through the massive use of mobile devices, data clouds, and the rise of IoT [13], enormous amount of data has been generated and analyzed for the benefit of society at a large scale. This data is often maintained in key-value storage [14]–[18], which is widely used due to its efficiency in handling data in key-value format, and flexibility to scale out without significant database redesign.

Such huge amount of data is partitioned across different storage instances in the data center. The data partitions and their storage node mapping (directory information) are either stored on a single coordinator node, e.g., the master coordinator in distributed Google file system [19], or replicated over all storage instances [14], [15], [20]. In the first approach, the master coordinator represents a single point of failure, and introduces a bottleneck in the path between clients and storage nodes. In the second approach, where the directory is replicated on all storage nodes, there are two coordination strategies that a client can use to select a node where the request will be sent: server-driven and client-driven.

In server-driven coordination, the client’s request is routed through a load balancer to a storage node. The selected node acts like the coordinator for the client request. It answers the query if it has the data partition or forwards the query to the right instance where the data partition resides. Unfortunately, this strategy has a higher latency because it introduces additional forwarding step when the request coordinator is different from the node holding the target data. In client-driven coordination, the client uses a partition-aware client library that routes requests directly to the appropriate storage node that holds the data. As shown in [14], this approach reduces the latencies by more than 50% for both 99.9th percentile and average cases, when compared with the server-driven one. This latency improvement is because the client-driven coordination eliminates the overhead of the load balancer and skips a potential forwarding step introduced in the server-driven coordination when a request is assigned to a random node. However, it introduces additional load on the client to periodically pull the updated directory information from one of the key-value storage nodes to perform the coordination locally on its side. It also requires the client to equip its application with some code specific to the key-value store used.

Another challenge in maintaining distributed key-value stores is handling dynamic workloads and coping with changes in data popularity [21], [22]. Frequent requests to hot data over cold data lead to load imbalance among storage nodes; some nodes are heavily congested while others become under-utilized. This results in a performance degradation of the whole system and a high tail latency. The most notable research in this area using the programmable switches includes NetCache [6] and Pegasus [9]. NetCache tackles the load balancing for high skewed read only workload via caching the most popular $O(n\log n)$ key-value pairs in the Top of Rack (ToR) switch. Pegasus solves the problem of load balancing...
through the selective replication approach. It maintains a coherent directory in the ToR switch for the most popular $O(n\log n)$ key-value pairs, and distributes the load between the storage nodes that have these replicated items. Although both of NetCache and Pegasus use the programmable switches to handle load balancing in distributed key-value stores, none of them handle the partition management problem in this environment. Also, they can only handle point queries and can not handle queries that ask for a specific key-range scan.

In this paper, we propose TurboKV: a distributed key-value store architecture that leverages the power and flexibility of the programmable switches to overcome the limitations of existing systems. TurboKV offloads the partitions management and query routing to be carried out in network switches. It uses a switch-driven coordination which utilizes the programmable switches as: 1) partition management nodes to store and manage the directory information of key-value store; and 2) monitoring stations to measure the load of storage nodes, where this monitoring information is used to balance the load among storage nodes.

TurboKV adapts a hierarchical indexing scheme to distribute the directory information records inside the data plane of the data center network switches. It uses a key-based routing protocol to map the requested key in the query packet from the client to its target storage node by injecting some information about the requested data in packet headers. The programmable switches use this information to decide where to send the packet to reach the target storage node directly. This in-switch coordination removes the load of routing the requests from the client in the client-driven coordination without introducing additional forwarding steps introduced by the coordination node in the server-driven one. Unlike existing systems [6], [9], TurboKV can coordinate the requests for point queries and range scans. It also handles two different partitioning techniques: hash partitioning and range partitioning.

To achieve both reliability and high availability, TurboKV replicates key-value pair partitions on different storage nodes. For each data partition, TurboKV maintains a list of nodes that are responsible for storing the data of this partition. It uses the chain replication [23] model to guarantee strong data consistency between all partition replicas. In case of having a failing node, requests will be served with other available nodes in the partition replica list. TurboKV also utilizes the architecture of software-defined network [24], [25]. In our architecture, a logically centralized controller, which has a global view of the whole system [26], makes decisions to migrate/replicate some of the popular data items to other under-utilized storage nodes using monitoring reports from the programmable switches. Then, it updates the switches’ data plane with the new indexing records.

We implemented a prototype of TurboKV using P4 on top of the simple software switch architecture BMV2 [27]. Our experimental results show that our proposed architecture improves the throughput and reduces the latency for all query types in the distributed key-value stores.

II. WHY IN-SWITCH COORDINATION?

We utilize the programmable switches to scale up the performance of distributed key-value stores with the in-switch coordination because of three reasons. First, programmable switches have become the backbone technology used in modern data centers or rack-scale clusters that allow developers to define their own functions for packet processing and provide flexibility to program the hardware.

Second, request latency is one of the most important factors that affect the performance of all key-value stores. This latency is introduced through the multiple hops that the request traverses before arriving to its final destination, in addition to the processing time to fetch the desired key-value pair(s) from that destination. When the key-value store is distributed among several storage nodes, the request latency increases because of the latency introduced to locate the desired key-value pair(s) before fetching them from that location. This process is referred to as partition management and request coordination, which can be organized by the storage nodes (server-driven) or by the client (client-driven).

Because client requests already pass through network switches to arrive at their target, offloading the partition management and query routing to be carried out in the network switches with the in-switch coordination approach will reduce the latency introduced by the server-driven coordination; the number of hops that the request will travel from the client to the target storage node will be reduced as shown in Figure 1. In-switch coordination also removes the load from the client in the client-driven coordination by making the programmable switch manage all the information for the request routing.

Third, the partition management and request coordination are communication-bounded rather than being computation-bounded. So, the massive evolution in the capabilities of the network switches, which provide orders of magnitude higher throughput than the highly optimized servers, makes them the best option to be used as partition management and request coordination nodes. For example, Tofino ASIC from Intel [12] provides few billions of packets processed per second with up to 12.8 Tbps bandwidth. Such performance is orders of magnitude higher than NetBricks [28] which processes millions of packets per second and has 10-100 Gbps bandwidth [7].

III. TurboKV ARCHITECTURE OVERVIEW

TurboKV is a new architecture of the future distributed key-value stores that uses an in-switch coordination model to maintain the partition management information of distributed key-value stores, and route clients’ queries based on their requested keys to the target storage nodes. Figure 2 shows the architecture of TurboKV which consists of the programmable switches, controller, storage nodes, and system clients. Programmable Switches. Programmable switches are the essential component in the proposed architecture. We augment the programmable switches with a key-based routing approach to deliver TurboKV query packets to the target key-value storage node. We leverage match-action tables and switch’s...
Fig. 1: Request Coordination Models

registrers to design the in-switch coordination where the partition management information will be stored on the path from the client to the storage node (Section IV-A). Other normal packets are processed and routed using the standard L2/L3 protocols which makes TurboKV compatible with other network functions and protocols (Section IV-B). Also, each programmable switch has a query statistics module to collect information about each partition’s popularity (Section V-A). This statistics are used to make data migration decisions to balance the load among storage nodes.

Controller. The controller is primarily responsible for system reconfigurations including (a) achieving load balancing between the distributed storage nodes based on the switches’ statistics reports (Section V-A), (b) handling failures in the storage nodes (Section V-B), and (c) updating each switch’s match-action tables with the new location of data through the control plane. TurboKV controller is an application controller that is different from the network controller in SDN, and it does not interfere with other network protocols or functions managed by the SDN controller. Our controller only manages the key-range based routing, data migrations and failures associated with them. Both controllers can be co-located on the same server, or on different servers.

Storage Nodes. They represent the location where the key-value pairs reside in the system. The key-value pairs are partitioned among storage nodes (Section IV-A1). Each storage node runs a simple shim that is responsible for re-forming TurboKV query packets to API calls for the key-value store, and handling TurboKV controller’s data migration requests between the storage nodes. This makes our design easy for integration with existing key-value stores without any modifications to the storage layer.

System Clients. TurboKV provides a client library which can be integrated with the client applications to send TurboKV packets through the network, and access the key-value store without any modifications to the application. Like other key-value stores such as LevelDB [17] and RocksDB [18], the library provides an interface for all key-value pair operations that is responsible for constructing the TurboKV packets and translates the reply back to the application.

IV. TurboKV DATA PLANE DESIGN

The data plane provides in-switch coordination model for the key-value stores, where all partition management information and query routing are managed by the switches. Figure 3 represents the whole pipeline inside the switch. In this section, we discuss how the switch data plane supports these functions.
replicas. The CR is simpler than the classical primary backup (PB) protocol [30]; write queries use fewer messages than the PB, (n+1) instead of (2n) where n is number of nodes.

3) Management Support in Switch Data Plane: In this section, we will discuss the design of the partition management tables related to the key-based routing protocol for the rack scale cluster shown in Figure 5(a). We will refer to the value we use for matching as the matching value, this value represents the key itself in case of range partitioning and the hashed value of the key in case of hash partitioning.

The partition management match-action table design is shown in Figure 5(b). Each record in the table consists of three parts: match, action and action data. The match represents the value that we match the matching value against, we refer to it as a sub-range. This sub-range represents the start and end keys of a sub-range from the whole key span in range partitioning, or the start and end hash values of a consecutive set of hash values in hash partitioning. The action represents the key-based routing that will be executed when a matching value falls within the sub-range. The action data consists of two parts: chain and length. Chain represents the forwarding information for nodes forming the chain of the sub-range. This information includes node’s IP address and the port from the switch to the storage node. Nodes’ information is sorted according to node’s position in the chain structure, and is used in updating packet during the action execution.

TurboKV uses two register arrays in the switch’s data plane to store the forwarding information: node IP array, and node port array. For each storage node, the forwarding information is stored at the same index in the two arrays as shown in Figure 5(c). For example, the information of storage node S1 is stored at index 1 in the two arrays. The index of the storage nodes in the register arrays is stored as action data in the match-action table records to form the chain as shown in Figure 5(b). The key-based routing uses these indexes to process the register arrays and fetch the forwarding information for the replica nodes.

B. Network Protocol Design

Packet Format. Figure 6(a) shows the format of TurboKV request packet sent from clients. The programmable switches use the Ethernet Type in the Ethernet header to identify TurboKV packets, and execute the key-based routing. Other switches in the network do not need to understand the format of TurboKV header, and treat all packets as normal IP packets. The ToS (Type of Service) in the IP header is used to distinguish between three types of TurboKV packets: range data packet, hash data packet, and TurboKV packet previously processed by the switch. The TurboKV header consists of three fields: OpCode, Key, and endKey/hashedKey. The OpCode stands for the code of the key-value operation (Get, Put, Del, and Range). Key stores the key of a key-value pair. In Range operation, Key and endKey/hashedKey are used to represent the start and end of the key-range, respectively. In case of hash partitioning, the endKey/hashedKey is set with the hashed value of the key to be used in the routing.

After processing packets by the programmable switch, the switch adds the chain header shown in Figure 6(c). This header is used by the storage nodes for the chain replication model. It includes two fields. The first field is the number of nodes which the packet passes by in the chain including the client IP (CLength). The second field has these nodes’ IP addresses ordered according to their position in the chain followed by the client IP at the end. The reply packet from storage node to client is a standard IP packet shown in Figure 6(b). The result is added to the packet payload.

Network Routing. TurboKV uses a key-based routing protocol to route packets from clients to storage nodes. The client sends the packet to the network. Once it reaches the programmable switch, the switch extracts the matching value from TurboKV header, and looks up the corresponding key-
C. Key-value Storage Operations

**PUT and DELETE Queries.** In CR, PUT and DELETE queries are processed by each node along the chain starting from the head and replied by the tail. On the switch side, after fetching the chain information from the register arrays based on a matching value, the query processing module sets the destination IP address in the IP header and egress port with the IP address and the forwarding port of the chain head node and changes the ToS value to mark the packet as previously processed. Then, the chain header is added to the packet with CLength equals to the length of the destination IP address, and the chain nodes’ IP addresses ordered according to their position in the chain followed by the client IP at the end as shown in Figure 7(a). Finally, the packet is forwarded to the next hop.

When the packet arrives at a storage node, the node updates its local copy of the data. Then, it reads the chain header, sets the destination address in the IP header with the IP of its successor and reduces the CLength by 1, then forwards the packet to its successor. Packets received by the tail node will have their chain header and TurboKV header removed, and the result is sent back to the client.

**GET Queries.** Following the CR, GET queries are handled by the tail of the chain. After fetching the chain information from the register arrays based on a matching value, the query processing module sets the destination IP address in the IP header and egress port with the IP address and the forwarding port of the chain tail node, and changes the ToS value to mark the packet as previously processed. Then, the chain header is added to the packet with CLength equals to 1 and one node IP which represents the client IP as shown in Figure 7(c). Finally, the packet is forwarded to the storage node. When the packet arrives at the storage node, the query result is added to the packet’s payload, and the client IP is used as the destination address in the IP header. The chain and TurboKV headers are removed from the packet and the result is sent back to the client.

**Algorithm 1 Range Query Handling**

```python
1: Input pkt: packet entering the egress pipeline
2: matched_subrange: the subrange where the start key of the requested range falls
3: Output pkt_out: packet to be forwarded to the next hop
4: Output pkt_out: packet to be circulated and sent to ingress pipeline as new packet
5: Begin:
6: pkt_out = pkt // clone the packet
7: if pkt.OpCode == range then
8: // check if range spans multiple nodes
9: if pkt.request.endKey > matched_subrange.endKey then
10: pkt_out.request.endKey = matched_subrange.endKey
11: pkt_out.request.Key = Next(matched_subrange.endKey)
12: if pkt.request.Key < matched_subrange.endKey then
13: end if
14: if pkt.request.Key < matched_subrange.endKey then
15: circulate(pkt_out) // send packet to ingress pipeline again
16: end if
17: end if
```

**Fig. 8: TurboKV Control Plane**

**Range Queries.** If the data is range partitioned, TurboKV can handle range queries. The requested range in the TurboKV header may span multiple storage nodes. Thus, the switch divides the range into several sub-ranges, each sub-range corresponds to a packet. Each of these packets contains the start and end keys of the corresponding sub-range. Each packet is handled by the switch like a separate read query and forwarded to the tail node of its partition’s chain. Unfortunately, the switch cannot construct new packets from scratch. Therefore, we placed the range operation check in the egress pipeline as shown in Figure 3. We use the clone and circulate operations, supported in the programmable switches and P4, to solve the packet construction problem. When a range operation is detected in the egress pipeline, the packet is processed as shown in Algorithm 1.

V. TurboKV CONTROL PLANE DESIGN

A. Query Statistics and Load Balancing

In TurboKV, the data plane has a query statistics module to provide query statistics reports to TurboKV controller. Thus, the controller can estimate each storage node’s load, and make decisions to migrate part of the popular data to one of the under-utilized storage nodes. As shown in Figure 8(a), the switch’s data plane maintains a per key-range counter for each key-range in the match-action table. Upon each hit for a key-range, its corresponding counter is incremented by one. The controller receives these statistics and resets the counters periodically. Based on the received statistics, If a storage node is over-utilized, the controller migrates a subset of the hot data in a sub-range to another storage node and reconfigures the chain of the updated sub-range. Then, the controller updates records in the match-action table of the switches with the new chain configurations. After data is migrated to other storage nodes, the old copy is removed from the over-utilized one.
Currently, the controller follows a greedy selection algorithm to select the least utilized node where data will be migrated. TurboKV uses data migration for load balancing as it adapts with all kinds of workloads compared to the caching approach, which performs well in highly skewed read-only workload.

B. Failures Handling

We assume that the controller process is a reliable process and is different from the SDN controller. We also assume that links in data centers are reliable and are not prone to failures.

Storage Node Failure. When the controller detects a storage node failure, it removes the failed storage node from its position in all chains, and updates their corresponding records in the key-based match-action table through the control plane. In each chain, the predecessor of the failed node will be followed by the successor of the failed node, reducing the chain length by $l$ as shown in Figure 8(b). If the failed node was the head of the chain, then the new head will be its successor. If the failed node was the tail of the chain, then the new tail will be its predecessor. Then, the controller restores the chain length by distributing the data of the failed node in sub-range units among other functional nodes, and adds these new nodes at the end of these sub-ranges’ chains.

Switch failure. The unreachable storage servers in the rack of the failed switch will be treated as failed storage nodes. The controller will distribute the load of these storage servers among other reachable servers as described before. Then, the failed switch needs to be rebooted or replaced. The new switch starts with an index table which contains all the key-ranges handled by its connected storage servers.

VI. SCALING UP TO MULTIPLE RACKS

In this section, we will discuss how to scale out the in-switch coordination in the data center network. In the network architecture of data centers. All the servers in the same rack are connected by a ToR switch. In the highest level, there are Aggregate switches (AGG) and Core switches (Core). To scale out distributed key-value stores with in-switch coordination, we develop a “hierarchical indexing” scheme. Each ToR switch has the directory information of all sub-ranges located on its connected storage nodes as described in Figure 5. In addition to the IPv4 routing table, each AGG switch has two match-action tables (range and hash), where each table consists of the sub-ranges in its connected ToR switches. The Core switches have the match-action tables of sub-ranges in its connected AGG switches. With each sub-range in either the AGG or Core switches, the action data represents the forwarding port towards the head or the tail of the sub-range’s chain. No chains are stored in these switches. When a packet is received by an AGG switch or core switch, this packet is processed by the key-based routing protocol without adding any chain header to the packet, and forwarded to one of the ports towards the head or tail of the sub-range’s chain based on the query type. When the packet arrives at ToR switch, the switch processes the packet as discussed in Section IV-C. Replicas of a specific sub-range may be located on different racks.

VII. PERFORMANCE EVALUATION

In this section, we show the effect of TurboKV on the key-value operations latency and system throughput.

Implementation. We implemented a prototype of TurboKV, including all switch data plane and control plane features, described in Section IV and Section V. We also implemented the client and server libraries that interface with applications and storage nodes. Due to lack of real hardware, the switch data plane is written in P4 and is compiled to the simple software switch BMV2 [27] running on Mininet. The key size of the key-value pair is 16 bytes. We used 4 register arrays for saving the actions data and query statistics. The switch data plane does not store any key-value pairs, which makes TurboKV consumes a small amount of the on-chip memory leaving enough space for processing other network operations. The controller is written in Python and communicate with the switches using the Thrift API generated by the P4 compiler. The client and server libraries are written in Python using Scapy [31] for packet manipulation. We used Plyvel [32], a Python interface for levelDB [17], as the storage agent. We used chain replication with chain length equals to 3.

Experimental Setup. Our experiments topology is shown in Figure 9. Each of the clients ($h_{17},...h_{20}$) runs the client library and generates the key-value queries. Each storage node ($h_{1},...h_{16}$) runs the server library and uses LevelDB as the storage agent. The whole topology runs on Mininet. The data is distributed over the storage nodes using the range partitioning described in Section IV-A1 with 128 records index table.

Comparison. We compared our in-switch coordination (TurboKV) with server-driven (S-Coord) and client-driven (C-Coord) coordinations described in Section I. In TurboKV, the key-based routing is used to route the query from client to target storage node, while both of C-Coord and S-Coord approaches route the query using the standard L2/L3 routing protocols. Note that, we compare with the ideal case of the C-Coord which can not be achieved in real life systems, where the client has the updated directory information and sends the query directly to the target storage node, ignoring
the latency introduced by pulling this information periodically from a random storage node. This ideal C-Coord represents the least latency that the client’s request can achieve because it represents the direct path from the client to the target storage.

Workloads. We use both uniform and skewed workloads to measure the performance of TurboKV under different workloads. The skewed workloads follow Zipf distribution with different skewness parameters (0.9, 0.95, 1.2). These workloads are generated using YCSB [22] with 16 byte key size and 128 byte value size. We generate different types of workloads: read-only workload, scan-only workload, write-only workload and mixed workload with multiple write ratios.

A. Effect on System Throughput

Impact of Read-only Workloads. Figure 10(a) shows the system throughput for read-only queries. We compare TurboKV vs the ideal C-Coord and S-Coord. In Figure 10(a), TurboKV performs nearly the same as the ideal C-Coord in the highly skewed workload (zipf-0.95 and zipf-1.2), and less than the ideal C-Coord by maximum of 5% in the uniform and zipf-0.9 workloads. This result is because TurboKV manages all directory information in the switch data plane and uses the key-based routing to deliver the requests to the storage nodes directly. Moreover, TurboKV eliminates the load of downloading the updated directory information periodically from storage nodes. In addition, TurboKV outperforms the S-Coord and improves the system throughput by 26-39%. This is because TurboKV eliminates the overhead of the load balancer and skips a potential forwarding step introduced in the S-Coord when a request is assigned randomly to a storage node.

Impact of Write Ratio. Figures 10(b) and 10(c) show the system throughput under uniform and skewed workload with varying the workload write ratio. As shown in Figures 10(b) and 10(c), the throughput decreases as the write ratio increases for the three approaches, because each write query has to update all the copies of the key-value pair following the chain replication approach before returning the reply to the client. TurboKV performs roughly the same as the ideal C-Coord in the workloads with low write ratio, but TurboKV outperforms the C-Coord as the write ratio increases. This behavior is because of the chain replication. In TurboKV, the switch inserts all the chain nodes in the packet that eliminates any further mapping on the storage nodes. But, in the C-Coord, the client sends the write query to the head of chain’s node. Then, each node in the chain has to look up its mapping table to know its successor. So, when the write ratio increases, this scenario is performed for larger portion of queries which affects the system throughput. Also, TurboKV outperforms the S-Coord by 26-44% in case of uniform workload, and by 37-47% in case of the skewed workload. This improvement is because of the elimination of the excess forwarding step and the elimination of further mapping look up to know the chain successor.

B. Effect on Key-value Operations Latency

Figure 11 shows the CDF of all key-value operations latencies under uniform and Zipf-1.2 workloads for TurboKV, ideal C-Coord and S-Coord. Figures 11(a), 11(b), 11(c), and 11(d) show that the read and write latencies in TurboKV are very close to the ideal C-Coord for uniform and skewed workload, because TurboKV skips potential forwarding step like the C-Coord by managing the directory information in the switch itself. Compared to S-Coord, TurboKV reduces the read latency by 16.3-19.2% for the uniform workload, and by 30-49% for the skewed workload. TurboKV also reduces the write latency by 11-12.3% for the uniform workload, and by 29-48% for the skewed workload. The reduction in skewed workload is larger than the reduction in uniform workload, because TurboKV reduces the tail latency by removing the load from the storage nodes from being the request coordinator. Figures 11(e), and 11(f) show that TurboKV reduces the scan latency by range of 13-23% for uniform workloads, and by 16-22% for skewed workloads when compared with the S-Coord. But, when compared with the ideal C-Coor, TurboKV increases the latency of the scan operation by 2-15% for uniform and skewed workloads, because of the latency introduced inside the switch from packet circulation and cloning to divide the requested range when it spans multiple storage nodes.

C. Comparison with Pegasus - one Rack

We compared the performance of TurboKV and Pegasus [9] on a rack scale topology, where one programmable switch connects 4 storage nodes and one client. Figure 12(a) shows the system throughput under different skewness values with mixed read/write queries (50%read, 50%write). We compare TurboKV, the S-Coord as the baseline (BL) and Pegasus. As shown in Figure 12(a), TurboKV performs nearly the same as Pegasus under different workload distributions. Both of
TurboKV and Pegasus outperform the BL by 62-93%. This result is because TurboKV uses the key-based routing with the data plane directory information to route the requests to the target storage nodes directly without any prior knowledge from the client about these storage nodes. TurboKV also can handle different types of partitioning (hash and range), and can scale out easily. On the other hand, Pegasus stores only the routing information along with the key-based-routing information to route the requests to the least loaded storage node. Other keys are mapped to a home server using a fixed algorithm. This home server will be responsible for request coordination (server-driven coordination). Figure 12(b) shows the CDF for scan latency for the skewed zipf-1.2 workload for an rack-scale configuration. Pegasus is not shown as it does not support range scan operation.

VIII. RELATED WORK

Distributed Key-value Stores. Key-value storage is widely used to support lots of large-scale applications [14]–[18], [20], [33], [34]. Distributing data over several key-value store instances has been widely studied. Some systems use hash functions to distribute the data among storage nodes [14], [15], [20], [34], while other key-value stores [17], [18] use key range partitioning where keys are stored on Sorted String Tables. All key-value stores use either client-driven coordination [14], [34], or server-driven coordination [14], [15], [20] to route client requests to target storage nodes. TurboKV supports hash and range partitioning, and uses in-switch coordination with the key-based routing protocol for request routing. Data Replication is used to achieve durability and high availability. [14], [15], [20], [34] replicate the data partitions and the mapping table on the storage nodes. TurboKV also replicates the data partitions on storage nodes and follows the CR model to guarantee strong consistency.

Hardware Acceleration. Lots of work used hardware to speed up the performance of the distributed systems. [6], [8] implement on-switch cache for load balancing. [7] uses the switches to implement in-network key-value store for coordination services. [10] also uses the programmable switches to scale up the performance of distributed shared memory systems. [35] uses the OpenFlow switches to save a forwarding rule for each cached key-value pair to route the request to the right caching node. [9] uses the switches along with the selective replication approach for load balancing. [11] uses the switches to improve the performance of the distributed reinforcement learning. TurboKV uses the programmable switches to solve the partition management problem; the switches act as the request coordination nodes that save the partition management information along with the key-based-routing protocol to route the request to the target storage nodes. In TurboKV, key-value pairs are saved on storage nodes which makes it not limited to applications of small data sizes.

IX. CONCLUSION

In this paper, we presented TurboKV; a distributed key-value store architecture that leverages the power and flexibility of the new programmable switches. TurboKV uses the in-switch coordination approach that utilizes the switches as partitions management nodes to store the key-value store partitions and replicas information, and uses a key-based routing protocol to route client packets to storage nodes. TurboKV decreases the query response time and improve system throughput. We believe that TurboKV can be deployed on the programmable switches currently integrated in the data center’s network.

X. ACKNOWLEDGMENT

This work was supported by NSF I/UCRC Center Research in Intelligent Storage and the NSF awards 1439662 and 1812537.

REFERENCES
Abstract—Shared micromobility is a rapidly growing transportation technology, with several companies establishing e-bike and e-scooter programs in cities all across the globe. In this paper, we use two years of empirical data on e-scooter usage from a pilot project in the City of Calgary to create a synthetic workload model of e-scooter traffic. Using this model, we develop a simulation environment to evaluate the impacts of different e-scooter management policies (e.g., fleet size, battery re-charging strategies, and urban parking infrastructure locations) on the efficacy of the e-scooter system. Our simulation results highlight the importance of proper site selection for parking areas and battery charging infrastructure.

Index Terms—Shared micromobility; Simulation modeling; System design and optimization

I. INTRODUCTION

In the past few years, bikes and scooters available for short-term rental have been deployed in cities all over the world as part of a paradigm shift in transportation towards shared mobility. As the global population becomes more urbanized, city planners are faced with the challenge of maintaining efficient and accessible transportation options in increasingly dense population centers. Shared vehicles play a key role in sustainable cities, and the market for these services has continued to grow rapidly since their introduction.

Shared mobility programs can offer many benefits in an urban setting. These include reducing reliance on personal vehicles, ameliorating traffic congestion, and making public transit more accessible by providing first-mile/last-mile access to major transit stations. Furthermore, the flexibility, novelty, and low cost of this transportation option make them appealing both for commute (e.g., work, school) and non-commute (e.g., leisure, sight-seeing, social outings) trips.

Nonetheless, implementing an urban shared micromobility program has its own set of challenges. From a user point of view, issues include consumer adoption, cost, ease of use, safety, and pandemic-related considerations. From the city’s perspective, challenges include infrastructure requirements (e.g., pathways, parking, re-charging stations), management policies (e.g., rider rules, sidewalk use, vendor selection, clutter from improperly parked scooters [15]), road safety (i.e., speed limits, interactions with cars, bikes and pedestrians), and sustainability. From a vendor’s point of view, challenges include operating costs and profitability (e.g., fleet size, pricing, battery re-charging strategies, revenue/cost sharing agreements).

These challenges are novel and involve multiple perspectives. Although ride-sharing and car-sharing programs have similar issues, and have existed around the world for quite some time, the current trend towards bike-sharing and scooter-sharing programs is quite recent. As such, there are relatively few studies on shared micromobility programs that might inform research and/or policy on these issues.

This paper aims to address some of these challenges, using the City of Calgary as a case study. In 2018, the City of Calgary announced a two-year pilot program for shared micromobility using e-bikes and e-scooters. Following the pilot study, the City released an open dataset to the public that included trip data for almost half a million e-bike and e-scooter trips for the years 2019 and 2020. The empirical data showed far greater usage for e-scooters than for e-bikes, and sustained growth in usage even during the pandemic. The City has since committed to a permanent year-round e-scooter program, and is in the process of selecting the vendors to provide and manage the scooters.

In this paper, we use simulation modeling to explore design issues and management strategies for a shared micromobility program based on e-scooters. Through analysis of the empirical 2019 and 2020 e-scooter data, we develop a detailed workload model including e-scooter trip characteristics, trip volume, and geospatial distribution within the downtown area. From this characterization, we construct a synthetic workload generator and a simulation model of Calgary’s e-scooter system. We then use this simulator to conduct experiments that examine the performance of the e-scooter service as impacted by factors such as e-scooter fleet size, parking infrastructure location and capacity, and battery re-charging infrastructure. Our simulation results offer insights into promising management strategies for an e-scooter system.

The research contributions outlined in this paper are:

- a workload characterization of e-scooter trips in downtown Calgary based on empirical trip data;
- a discrete-event simulation model constructed from that characterization; and
- results from simulation experiments regarding fleet size, parking locations, and battery charging infrastructure.

The remainder of this paper is organized as follows. Section II provides background information on shared micromobility, and reviews prior related research on shared mobility and traffic simulation. Section III provides a workload characterization of e-scooter traffic. Section IV describes the construction and validation of our simulation model. Section V presents simulation results, including the scenarios considered and our findings. Finally, Section VI concludes the paper.
II. BACKGROUND AND RELATED WORK

This section provides some background information on the field of shared micromobility and transportation simulation.

A. Shared Mobility

Shared mobility describes any transportation services that are shared by multiple users, including vehicle-sharing and ride-sharing services. The term ‘shared micromobility’ specifically encompasses bike- and scooter-sharing programs, such as those offered by companies like Bird, Lime, Roll, and Spin, who provide a fleet of vehicles for short-term rental use. These programs are rapidly gaining popularity around the world, with a reported market potential of $200-300B in the United States by 2030 [4]. At the time of writing, Lime has established shared mobility programs in more than 100 cities across the United States, Canada, and Europe [8], while Bird has more than 200 scooter-share programs worldwide [1].

There are many reasons why cities may want to pursue the development of shared micromobility services. Studies have shown that these programs create positive health impacts and reduce greenhouse gas emissions [14]. Moreover, increased availability of bike- and scooter-share options may also help to address transportation inequity by reducing reliance on individual vehicle ownership for transportation, and facilitating first-mile/last-mile connections to public transit [13].

In recent years, the focus of shared micromobility programs has begun to shift from bikes to scooters. In 2019, the National Association of City Transportation Officials reported that e-scooter usage had surpassed bike usage in shared-mobility trips, with 36.5 million bike-share trips and 38.5 million scooter-share trips in the United States the previous year [9].

In 2018, the City of Calgary announced a two-year pilot program on shared micromobility, with fleets of dockless bikes and e-scooters throughout the city. Anonymized trip data, amounting to approximately 450K trips, was collected in 2019 and made available to the public in 2020 [16].

B. Related Work

The prevalence of e-scooters in shared mobility systems is a relatively recent development, so there are comparatively few studies on shared e-scooter programs. Nonetheless, the body of research on this topic has been growing rapidly with the increasing public interest in shared e-scooters. As shared micromobility continues to develop and become more established within transportation ecosystems, understanding traffic patterns, costs and benefits, and necessary infrastructure will be crucial in maximizing the efficacy of these systems.

Prior studies of shared e-scooters have focused primarily on characterization and behavioural analysis of the users. In 2017, Sheehan et al. [13] discussed the relationship between shared mobility and transportation equity, and identified specific geographic, economic, social, and technological barriers to access. In 2020, Jiao and Bai [6] examined 1.7 million e-scooter trips in Austin, TX between April 2018 and February 2019, and confirmed that e-scooter usage tends to correlate with specific elements of the built environment, such as university campuses or the downtown. Reck et al. [11] investigated the influences of factors such as distance and time of day on the choice that users make between different shared-mobility vendors and different modes of transportation.

Another common approach to studying disruptive transportation technologies is to use simulation or case studies to examine the impact of policy or technology on the efficacy of these systems, or to consider possible approaches to addressing known challenges. For example, Clemente et al. [2] identified the key challenges of the ‘car sharing problem’ as: (1) optimal fleet size; (2) location of parking areas; (3) pricing policies; and (4) flexibility of use. They investigated strategies for maintaining appropriate geographical distribution of vehicles without unduly compromising flexibility, via real-time monitoring and pricing incentives. Subsequently, Pfrommer et al. [10] examined similar issues with regard to public bicycle sharing schemes. They compared the effectiveness of a tailored routing algorithm for collection and redistribution of bicycles, and proposed dynamic incentives to encourage users to adjust the destinations of their trips. Our work is similar to these prior works, but with a specific focus on an e-scooter system.

More generally, simulation modelling has been used to evaluate public transport accessibility [5], and to determine optimal placement of EV charging stations to maximise the effective range of electric cars [12]. In 2016, Dia and Javanshour [3] conducted simulation experiments to assess the feasibility of using autonomous shared mobility vehicles. More recently, Yan et al. [17] used simulation modeling to determine optimal locations for battery-swapping stations to facilitate shared e-scooter travel between tourist destinations.

III. DATA AND ANALYSIS

We used two empirical datasets for the construction and validation of our simulation model, as described next.

The first dataset is the collection of detailed trip-level data from 2019, which is available [16] on the City of Calgary’s open data portal. Table I shows the relevant headers from this dataset, which summarizes e-scooter trips on an hourly basis from May to December in 2019. Although these data include some geographic information, there is insufficient spatial resolution to derive any particular traffic patterns beyond an obvious concentration of scooter trips within the downtown area. For the purposes of our study, we filtered this dataset to focus solely on trips in the downtown area.

The second dataset consists of aggregate data from 2019 and 2020, including geographic information about scooter parking and scooter traffic volumes along individual streets. This dataset does not contain any trip-level information.

A. Daily Traffic Volume

Figure 1 shows the total daily count of e-scooter trips in 2019 and 2020. The periodic spikes in the graph reflect weekly cycles, for which scooter usage differs between weekdays and weekends. The total daily trip volume is highest in the summer months (June to September), which is consistent with the weather conditions necessary for safe scooter operation.
The scooters were introduced to Calgary’s streets later in 2019 (July) than in 2020 (May), resulting in different peaks in usage. The maximum fleet size also increased to 2,300 in 2020 from 1,500 in 2019. Despite the public health orders in response to the COVID-19 pandemic, which limited commuter traffic to and from the downtown, and reduced the permitted scooter fleet size in May and June, the overall trip volume is distinctly higher in 2020 compared to 2019. These trends suggest that demand for e-scooter shared mobility continues to grow.

### TABLE I
**Overview of City of Calgary Dataset (2019)**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Date</td>
<td>Date</td>
</tr>
<tr>
<td>Start Hour</td>
<td>int (0-23)</td>
</tr>
<tr>
<td>Start Day</td>
<td>String</td>
</tr>
<tr>
<td>Trip Distance</td>
<td>int (meters)</td>
</tr>
<tr>
<td>Trip Duration</td>
<td>int (seconds)</td>
</tr>
<tr>
<td>Start Point</td>
<td>Point(Lat, Long)</td>
</tr>
<tr>
<td>End Point</td>
<td>Point(Lat, Long)</td>
</tr>
</tbody>
</table>

### B. Hourly Trip Volume

An analysis of the average number of trips per hour reveals a clear diurnal pattern, as shown in Figure 2. Specifically, trip volumes peak around 4PM, and decline to their lowest values after midnight. Figure 2 also shows a notable difference between weekend and weekday traffic patterns. Weekday traffic follows a well-known tri-peaked behaviour [11], with distinct peaks at 8AM, 12PM, and 4PM, which correspond to morning rush hour, lunch hour, and evening rush hour, respectively. By contrast, weekend traffic increases later in the day, and has only a single peak in the early afternoon. All days have a slight ‘shoulder’ in the traffic patterns around 9PM, with this shoulder being most pronounced on Friday and Saturday evenings, and least evident on Sunday and Monday evenings.

### C. Geospatial Distribution

In both datasets, the majority of scooter trips originate from the downtown area. When broken down by neighborhood, the 2020 aggregate data shows that 68% of trips occur within the six neighborhoods at the heart of downtown.

Figure 3 is a heatmap showing the daily average trip count across each street in downtown Calgary. There is a high concentration of trips along the River Walk on the north (top) edge of downtown, with other noticeable concentrations along the 8th Avenue pedestrian mall (adjacent to the downtown Light Rail Transit line), 12th Avenue, and 17th Avenue. There are also many scooter trips along 4th Street and 5th Street, which are two of the main connectors across the railway tracks separating the north and south parts of downtown.

### D. Trip Characteristics

Figure 4 shows a statistical summary of the characteristics of e-scooter trips. From top to bottom, these graphs show the empirical distributions (histograms) for trip distance (in meters), trip duration (in seconds), and the average speed (in kilometers per hour, kph) during the e-scooter trip. The observed trip distances range from 100 m (i.e., slightly less than a typical city block) to more than 27 km. The mean trip distance is approximately 1.7 km. The shape of the distribution resembles an exponential distribution (CoV = 1.04), though the variance is higher, and the upper tail more pronounced.

Figure 5 shows that trip distance varies sharply by time of day. The predominant diurnal pattern shows a relatively steady
increase in average trip distance throughout the day, but with an abrupt drop in the wee hours of the morning. In addition, there is a pronounced difference between weekend trips and weekday trips. Weekends tend to have longer average trips with a peak earlier in the day, suggesting that these trips are recreational rather than for commuting.

Fig. 4. Empirical distributions for distance, duration, and speed (2019 data)

(a) Trip distance (m)
(b) Trip duration (s)
(c) Average trip speed (kph)

The trip durations follow a similar distribution to the trip distances, but deviate even more from an exponential distribution ($CoV = 1.08$). The shortest reported trip distance in the 2019 dataset is 30 seconds; the longest exceeds 2.5 hours.

The average trip speed, shown in Figure 4(c), does not resemble any immediately obvious distribution. These values are not reported in the 2019 dataset; rather, they are calculated (as a sanity check) from the distance and the duration. Interestingly, there is some correlation between speed and distance, as shown in Figure 6. While the maximum observed speed is near 30 kph (consistent with the reported top speed of many commercial e-scooters) regardless of trip distance, the minimum observed speed increases steadily with trip distance. This makes sense intuitively, since people going on long trips are unlikely to do so slowly. Furthermore, users may grow more comfortable with higher speeds once they have become familiar with riding e-scooters.

Fig. 5. Average trip distance (m) by hour and day of week (2019 data)

Fig. 6. Scatterplot of trip speed and trip distance (2019 data)

IV. SIMULATION MODEL

Based on the empirical trip data, we developed a workload model to generate 30 days of synthetic e-scooter trips within a Java simulation environment. Table II shows the parameters and settings used. To reduce computational load while processing a month’s worth of simulated trips on the geographic model of downtown Calgary, the simulation model focused on the events for: (1) beginning a trip; (2) ending a trip; and (3) the vendor collecting idle low-battery scooters to be recharged.

Several simplifying assumptions were made while constructing the simulation model, as follows:

- new e-scooter trips occur according to a Poisson arrival process, whose rate varies with time and day of week;
- users select e-scooters at random without preference for location or battery charge level, as long as the e-scooter has sufficient charge to complete the planned trip;
- no trips are abbreviated by low e-scooter battery level;
TABLE II
SIMULATION MODEL PARAMETERS

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value/Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Scooters</td>
<td>500</td>
</tr>
<tr>
<td>Inter-Trip Time</td>
<td>Exponential(hour, dayOfWeek)</td>
</tr>
<tr>
<td>Trip Distance</td>
<td>Exponential(hour, weekday/weekend)</td>
</tr>
<tr>
<td>Trip Speed</td>
<td>Empirical</td>
</tr>
<tr>
<td>Low Charge Threshold</td>
<td>25%</td>
</tr>
</tbody>
</table>

- all e-scooter trips maintain a constant speed between initial acceleration and final deceleration, with no stops;
- after recharging, e-scooters are returned to the fleet at the same location where they were picked up for recharging.

A. Trip Generation

Because neither of our empirical datasets included trip start times at a sufficiently fine granularity to determine the empirical inter-arrival time distribution, we assumed that the arrival process would follow a Poisson (random) arrival process, though with time-varying rates. For this purpose, we constructed a 24x7 array to represent the mean inter-arrival time on an hourly basis for each day of the week. We then generate simulated inter-trip times at random from an appropriate exponential distribution. These inter-arrival times can be scaled by a constant factor to reflect the increased trip volumes reported in the 2020 aggregate data.

The target trip distance is similarly selected from an exponential distribution, using a 24x2 array of average hourly trip distances observed for weekdays or weekends. This distance determines the minimum bound for the average trip speed, which is then selected from the empirical distribution determined from the 2019 trip data.

As part of the validation for our synthetic workload model, we produced Quantile-Quantile (QQ) plots comparing the simulated trips with the empirical data from 2019. Figure 7 shows that the synthetic trip generation using these methods closely approximates real-world trips. The fit for trip speed is excellent (as expected, since the empirical distribution is used), while the fit for trip distance is satisfactory. The main deficiency is in the tail of the distribution, for which the exponential model is lighter than the empirical distribution.

When a new trip event is generated, a random e-scooter is selected from the pool of available e-scooters (i.e., not currently occupied, and not being recharged). If the e-scooter does not have sufficient battery level remaining to travel the full distance of the planned trip, a new e-scooter is randomly selected. If, after five attempts, no suitable e-scooter has been selected, the trip is counted as unfulfilled. Because both the 2019 and 2020 datasets contain information about fulfilled demand, it is difficult to determine ground truth for unfulfilled demand. Furthermore, since neither dataset provides sufficient detail about the directionality of traffic, we use the location of this randomly selected e-scooter to determine the origin point of the trip, rather than a randomly selected origin determining the e-scooter to be used.

B. E-Scooter Battery Model

We constructed a detailed model for e-scooter batteries based on the electric vehicle battery usage equations given by Kurczveil et al. [7]. Their formulae use kinematics to determine the energy transferred between the vehicle battery and the vehicle motor. In the following Equations (1)-(7), \( v \) is the speed, and \( \Delta s \) is the distance travelled. \( E_{\text{batt}} \) refers to the chemical potential energy remaining in the battery, and \( E_{\text{veh}} \) refers to the kinetic energy residing in the vehicle. We ignore potential energy since the downtown core in Calgary is relatively flat. \( E_{\text{gain}} \) is the energy gained (or lost) by the battery between discrete time steps \( k \) and \( k + 1 \).
\[ E_{\text{batt}}[k+1] = E_{\text{batt}}[k] + \Delta E_{\text{gain}}[k+1] \cdot \eta_{\text{recup}} \] (1)
\[ E_{\text{batt}}[k+1] = E_{\text{batt}}[k] + \Delta E_{\text{gain}}[k+1] \cdot \eta_{\text{prop}}^{-1} \] (2)
\[ \Delta E_{\text{gain}}[k] = E_{\text{veh}}[k] - E_{\text{veh}}[k+1] - E_{\text{loss}}[k] \] (3)
\[ E_{\text{veh}} = \frac{1}{2} m v^2 \] (4)
\[ E_{\text{loss}}[k] = \Delta E_{\text{air}}[k] + \Delta E_{\text{roll}}[k] \] (5)
\[ \Delta E_{\text{air}}[k] = \frac{1}{2} \rho_{\text{air}} \cdot A_{\text{veh}} \cdot c_w \cdot v^2[k] \cdot |\Delta s[k]| \] (6)
\[ \Delta E_{\text{roll}}[k] = c_{\text{roll}} \cdot m \cdot g \cdot |\Delta s[k]| \] (7)

Using these equations, battery usage is estimated based on distance travelled and the average speed. We use Equation (1) when the energy gain from the battery is positive (i.e., acceleration), and Equation (2) when the energy gain is negative (i.e., deceleration, or constant speed). Our simulation assumes each e-scooter battery contains 1350 kJ at full charge. Table III shows the other values used in our battery model.

To simulate e-scooter recharging, a recurring event is scheduled for 10PM each evening to represent the vendor managing their fleet. This event creates a list of every e-scooter to be loaded onto the collection vehicle. The scooters are fully charged overnight, and the scooters for each e-scooter to be loaded onto the collection vehicle. However, as the e-scooter fleet size increases, the average number of scooters concurrently in use reaches a plateau around 65 scooters. This result is a manifestation of Little’s Law, a well-known conservation law from the field of queuing theory. For our baseline fleet size of 500 e-scooters, \( \lambda = 0.0664 \text{trips/s} \) and duration \( T = 981.68 \) s, so the average number of e-scooters in use at one time should be \( N = 65 \).

The results in Table IV also show that as the fleet size increases, the time and driving distance required to collect scooters increases, and the percentage of trips ending at a designated parking zone decreases. These undesirable trends reflect an excess supply of e-scooters, resulting in many unused and/or improperly parked e-scooters. These concerns must be balanced against the desire to minimize the number of unfulfilled trips. Based on our simulation results, at the current demand level, the optimal number of e-scooters to operate within the downtown area is between 400 and 500.

<table>
<thead>
<tr>
<th>Num. Scooters</th>
<th>100</th>
<th>200</th>
<th>400</th>
<th>800</th>
<th>1600</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max. in use</td>
<td>199</td>
<td>199</td>
<td>199</td>
<td>199</td>
<td>199</td>
</tr>
<tr>
<td>Avg. in use</td>
<td>65.3</td>
<td>65.3</td>
<td>65.3</td>
<td>65.3</td>
<td>65.3</td>
</tr>
<tr>
<td>Succ. trips/day</td>
<td>3656</td>
<td>5227</td>
<td>5753</td>
<td>5750</td>
<td>5732</td>
</tr>
<tr>
<td>Unsucc. trips/day</td>
<td>2086</td>
<td>557</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>% SNG-Parked</td>
<td>2.87%</td>
<td>2.99%</td>
<td>2.97%</td>
<td>2.95%</td>
<td>2.90%</td>
</tr>
<tr>
<td>Avg. Coll. Dist.</td>
<td>24.3 km</td>
<td>28.5 km</td>
<td>30.7 km</td>
<td>29.7 km</td>
<td>27.9 km</td>
</tr>
</tbody>
</table>

**B. Parking Areas**

The second set of simulation experiments focuses on e-scooter parking. The aggregate data from 2020 included a JSON file of 23 Share-and-Go (SNG) parking zone locations within the downtown. We use these locations as the baseline in the simulation model, assuming sufficient space for 6 e-scooters at each SNG zone. We then determine the number of trips ending at a parking zone with available space.

Our simulation experiments consider four strategies for placing additional scooter parking, increasing the baseline parking capacity by factors of 2, 4, 6, 8, and 10. The first strategy simply increases the size of existing SNG parking zones. The second strategy places additional SNG zone locations at random throughout the downtown area. The third strategy places additional SNG zones on the longest streets. The fourth strategy places additional SNG zones along streets with the highest e-scooter traffic volumes. Figure 8 provides a visual illustration of these latter three strategies.
In addition, these four sets of experiments were repeated with a modification to the simulation that allowed trips that ended on an edge with no available parking a 25% or 50% chance of extending to an adjacent edge with a parking space available. This variation tests the efficacy of nudging (e.g., discounts) to incentivize users to park their e-scooters properly.

Figure 9 presents the results from these simulation experiments. The vertical axis shows the percentage of e-scooter trips ending at an SNG zone with an available parking space (higher is better). The colored bars show the strategies.

The results in Figure 9 reveal three interesting trends. First, simply increasing the number of e-scooter parking spaces at the existing SNG zones is ineffective. Adding new parking zones is always better, even when the locations are selected at random. The improvements are magnified when SNG zones are placed according to edge length, and amplified further when assigned according to e-scooter traffic volume. These results suggest that the number and location of SNG parking zones is far more important than just the total number of parking spaces. Second, although the volume-based strategy for placing additional SNG zones yields the best results in terms of trips ended at an SNG zone, the length-based strategy is actually better when counting trips ending at or adjacent to an SNG zone. This is likely due to the proximal clustering of SNG zones induced by volume-weighted edge placement (see Figure 8c). Having greater geographic spread for the parking zones is better. Third, nudge strategies could be highly effective. That is, modifying the simulation to extend trips to adjacent edges with SNG zones resulted in significant improvement in all considered scenarios. However, achieving this behaviour in a real-world scooter system might require the implementation of incentives to improve proper parking.

C. Battery Charging Stations

Our final simulation experiment is designed to assess the impact of installing e-scooter battery charging stations within the downtown area. Doing so could reduce the number of unfulfilled trips, as well as the driving time and distance required by vendors to manage their scooters.

Figure 10 shows the candidate charging locations considered in the simulations. Each of the 23 existing SNG zones (A-W, lighter) were considered, as well as ten additional locations (a-j, darker), chosen based on e-scooter traffic volume.
Three rounds of experiments were conducted, allowing for one, two, or three charging stations, each capable of charging up to six e-scooters at a time. After running an initial test with only one charging location, and comparing the number of scooters charged for the 33 candidate locations, the top ten locations (L, E, R, K, W, B, C, a, V) were selected for consideration in the next round. After running the second set of tests, with pairs of charging locations chosen from the contenders, the seven best-performing locations (R, W, B, C, c, a, V) were used in the final tests with three charging stations.

Figure 11 shows the results from these experiments. As expected, the percentage of trips ending at a charging station with an available charging bay (purple; higher is better) increases when there are more charging locations. However, there are diminishing returns: the first and second charging location each have a pronounced effect, while adding the third one has only a marginal benefit. Among the seven best locations identified, four (R, B, C, V) are along 17th Avenue, and three (a, c, W) are along the River Walk. These locations make sense intuitively, since they are high-traffic areas for people, bikes, and scooters, especially on evenings and weekends.

Finally, Figure 11 shows the effects of the charging stations on collection time (pink line; lower is better) for the vendors when managing their fleet of scooters. From an average daily collection time of 2 hours with no charging stations, the average daily collection time decreases by 10% to 1 hour and 50 minutes for the best combination of three charging stations.

![Fig. 11. Effects of charging stations on availability and collection time](image)

### VI. CONCLUSIONS

In this paper, we have presented a workload characterization study and a simulation model of the e-scooter pilot project in downtown Calgary. Through experimentation with our simulation model, we have identified multiple important factors impacting e-scooter parking and collection costs.

The key findings of this paper may be summarized as follows. First, the appropriate number of e-scooters to be deployed in downtown Calgary is between 400 and 500 scooters. Second, increasing the number of parking zones for e-scooters is far more effective than simply increasing the number of spaces at existing parking zones, especially if users can be nudged to properly park their scooters. Third, installing one or more charging stations in the downtown area would help reduce operational costs for e-scooter vendors.

Future enhancements of our simulation model could include the implementation of e-scooter battery-swapping stations, or more refined models for trip origin/destination based on empirical data and knowledge of transit hubs.

### ACKNOWLEDGEMENTS

The authors thank the MASCOTS 2021 reviewers for their feedback and comments on our paper. The authors are grateful to Andrew Sedor at the City of Calgary for sharing his knowledge about the e-scooter system, and to Arda Ertürk at Roll for information about e-scooter batteries. Financial support for this research was provided by the NSERC CREATE program on Integrated Infrastructure for Sustainable Cities (IISC).

### REFERENCES


Performance Characterization of MPI_Allreduce in Cloud Data Center Networks

Malek Musleh∗, Allister Alemania†, Roberto Penaranda‡, Pedro Yebenes Segura§
Intel Corporation (@intel.com)
Email: *malek.musleh, †allister.alemania, ‡roberto.penaranda, §pedro.yebenes.segura

Abstract—Advancements in hardware architecture and system design have enabled the transformational pivot towards cloud-computing. The rising cost of on-premise, vertical scaling, and maintenance combined with the rise of workload heterogeneity have fueled this paradigm shift. Furthermore, increasingly demanding use-cases for storage and High-Performance Computing (HPC) as well as the emergence of new workloads such as machine learning and big-data have motivated research into network traffic analysis, resource disaggregation, job scheduling, and orchestration in a bid to reduce total cost while maintaining high performance. A prime focus of this research is the communication performance of collectives, which comprise a significant portion of the communication of many of the aforementioned workloads. In this paper, we characterize the performance of MPI_Allreduce, which is used extensively in HPC and Deep-Learning (DL) training workloads in cloud environments. We demonstrate several key insights including that the Ring algorithm performs better than the Rabenseifner algorithm under conditions of high congestion and packet drops. We further illustrate detailed performance analysis under different assumptions and provide recommendations on how to address issues that can manifest.

Index Terms—Data Center, Cloud Computing, Deep Learning, AI Training, HPC, Collectives, Congestion Control, AllReduce, Rabenseifner, Ring, Kary Tree, Recursive Doubling

I. INTRODUCTION

Cloud Service Providers (CSPs) have seen an accelerated influx of High Performance Computing (HPC) and emerging workloads such as big data, storage, and deep learning (DL) in recent years from both academia and industry [1]. These institutions are migrating to the cloud as a result of improved competitive performance and cost-structure relative to traditional on-premise deployment. As a consequence, HPC workloads—for which low latency, predictable performance, and highly performant networks are critical—are more often being executed in cloud environments long known for their high latencies, unpredictability, and resource sharing as a result of multi-tenancy.

A common middle-ware for HPC applications is the Message Passing Interface (MPI) [4] for which there are a number of popular implementations such as MPICH [5], OpenMPI [6] and MVAPICH [7]. MPI enables distributed/parallel applications to communicate using point-to-point messaging APIs as well as collectives APIs that provide a means to communicate efficiently among a logical subset of processes called a communicator. One of the most heavily used and performance critical MPI collectives APIs is MPI_Allreduce, which performs a mathematical reduction operation (e.g., min, max, sum) among the processes of a communicator and then distributes the result to all participants. Thakur et al. propose several optimizations of collective communication operations [8]. Previous works analyze the frequency of collectives usage and the portion of application time collectives consume in MPI applications [9]. Notably, many DL training models are heavily reliant on MPI_Allreduce as a result of extensive use of the collective by the software framework such as Horovod [10].

An important characteristic of MPI_Allreduce is that it is a blocking communication API that is dependent on the communication performance of all participating processes. Therefore, all processes participating in the MPI_Allreduce, $P_{Communicator}$, are slowed down to the speed of the poorest performing participant, $P_{Slow}$. HPC networks minimize this inefficiency by ensuring low latency and high bandwidth via a myriad of (often costly) techniques including tight flow control feedback loops, link over-provisioning, adaptive routing, and specialized network topologies such as Dragonfly [11], Dragonfly+ [12] (also known as Megafly [13]), and Slim-Fly [14]. Even with all this effort in HPC systems, network interference and congestion can remain a significant problem. Cloud environments are typically augmented over time, with components from various vendors, rather than from a single vendor building out a dedicated high-end performance fabric. Furthermore, ubiquitous use of Virtual Machines (VMs) and containers introduce dimensions of heterogeneity that are not experienced in HPC deployments—job and resource orchestration, diverse user-driven application requirements, heterogeneous congestion control algorithms, and so forth all come together to create a far more dynamic environment.

In light of the differences in architecture and design goals between HPC and cloud deployments, the significant number of works that analyze collectives in HPC environments might be less applicable to cloud deployments. Thus, in this work, we characterize network performance of MPI_Allreduce in a cloud Data Center Network (DCN) environment. In the context of a cloud DCN environment, we make the following contributions on performance analysis:

1) Algorithm vs. congestion control protocols
2) Placement strategies
3) Multi-tenancy variability and sensitivity

The rest of this paper is organized as follows. §II describes some background material of collective operations. §III discusses the methodology used in this work. We present our
findings in §IV. We discuss related work in §V. Finally, §VI summarizes our work and describes some future work.

II. BACKGROUND

The MPI_Allreduce operation mandates the high-level reduction operation (e.g., min, max) that must be performed but it does not dictate the algorithmic implementation. As such, there are a number of well-known implementations of MPI_Allreduce, all of which operate in an iterative manner. For example, if we consider P as the set of processes, in iteration 0, process 0, P_0, sends its data to P_1, P_1 sends its data to P_2, and so forth. We briefly describe several MPI_Allreduce implementations in the following paragraphs.

1) Ring: Ring algorithm was described by Barnett et al. [15]. The Ring algorithm organizes the processes, of a communicator (recall that a communicator contains all the processes that participate in a collective) into a logical ring structure. Each process’ message data (of size M) is typically partitioned into N segments of M/N size, where N is the number of processes. The algorithm executes two phases: reduce-scatter and all-gather (results sharing). The first phase performs N − 1 iterations where, in every iteration, P_i simultaneously receives a segment from its predecessor, P_{i-1}, and sends a different segment to its successor, P_{i+1}. After N − 1 iterations, each process has a completely reduced segment that is shared to all other processes in the all-gather phase, which takes N − 1 iterations.

2) Recursive Doubling: The Recursive Doubling algorithm [16] executes in \( \log_2(N) \) iterations. In iteration \( i \) \((i = [0, \log_2(N) − 1])\), processes that are distance \( 2^i \) apart communicate with each other. The nature of the communication pairings only works optimally when the number of processes is a power of two.

3) Rabenseifner: The Rabenseifner algorithm [17] is decomposed into two different steps: a reduction with results that are scattered among the different processes followed by an allgathering of these results. A variation of the Recursive Doubling (or Halving) algorithm is typically employed in each of the decomposed operations. This algorithm requires \( 2 \cdot \log(N) \) iterations and exchanges \( 2N \cdot \log(N) \) messages, remaining optimal in data transport.

III. METHODOLOGY

We use the Fabsim-X simulation framework [18] to conduct our analysis. The framework provides the capability to simulate large-scale DCNs with the ability to represent sophisticated workloads such as collectives, via our Scalable Workload Model (SWM) format [19].

In this work, we utilize the framework’s fine-grain Credit-Based Flow Control (CBFC) model to simulate HPC networks, and we use its TCP NewReno [20] and DCTCP [21] congestion control models to simulate cloud DCNs. Aside from this difference, we keep the remaining simulation parameters consistent. We use the Virtual Output-Queued (VOQ) [22] switch model depicted in Fig. 1 within the network topologies shown in Fig. 2. This switch marks packets with the Explicit Congestion Notification (ECN) flag when a specific percentage of buffer occupancy is reached, which is referred to hereafter as ECN threshold. We employ two process placement schemes:

1) Linear placement: processes are placed sequentially on nodes (e.g., process 0 is placed on node 0, process 1 is placed on node 1, and so forth)

2) Random placement: processes are placed on a random node (e.g., process 0 is placed on node 53, process 1 is placed on node 27, and so forth).

Placement is chosen once at initialization of the job. Key parameters of the topology are detailed in Table I. Unless otherwise specified, we execute five back-to-back iterations of MPI_Allreduce. Five iterations are executed to ensure we capture steady-state network behavior.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Congestion Control Protocol</td>
<td>TCP NewReno, DCTCP</td>
</tr>
<tr>
<td>Topology</td>
<td>2-Tier Fat-Tree</td>
</tr>
<tr>
<td>Topology Link Delay</td>
<td>100 ns</td>
</tr>
<tr>
<td>Routing Algorithm</td>
<td>Deterministic</td>
</tr>
<tr>
<td>Switch Model</td>
<td>Virtual Output-Queue (VOQ)</td>
</tr>
<tr>
<td>Switch Capacity</td>
<td>64 Ports</td>
</tr>
<tr>
<td>Intra Switch Delay</td>
<td>256 KB per port/virtual channel</td>
</tr>
<tr>
<td></td>
<td>200 ns</td>
</tr>
</tbody>
</table>

Table I: Simulation Parameters

A. MPI_Allreduce in HPC Networks

We first briefly characterize two popular MPI_Allreduce implementations, Ring and Rabenseifner, in an HPC network setting. In doing so, we provide a baseline performance expectation of MPI_Allreduce from the perspective of applications that, until recently, have been executed in HPC environments.

We execute five MPI_Allreduce operations back-to-back on the tapered topology depicted in Fig. 2 (1 process per node;
randomly placed) using a variety of message sizes and number of participating processes, \(N\), and we use completion time as the performance metric. Transmission is performed using lossless CBFC, or per-hop movement only when the subsequent destination hop has sufficient buffer availability, typical of HPC networks. For small message cases, we expect iteration/message count to dominate performance. Accordingly, we expect Ring performance to follow \(O(N)\) and Rabenseifner performance to follow \(O(\log_2(N))\). In contrast, when sending larger messages, we expect bandwidth to dominate, at which point performance becomes network-bandwidth bound.

Table II shows the results of our simulations following the expectations. With small messages, the performance delta between Ring and Rabenseifner increases with process count. As message size increases, the performance benefit of Rabenseifner’s scaling diminishes as bandwidth becomes the dominant factor in determining completion time. As the scale increases further, however, we observe a performance inflection (see 512 processes @ 32MB; 1024 processes @ 64MB). Analysis of job completion times show that Rabenseifner scales better than Ring at smaller message sizes (i.e. not bandwidth constrained).

<table>
<thead>
<tr>
<th>Message</th>
<th>Number of Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>16KB</td>
</tr>
<tr>
<td>16KB</td>
<td>11.75</td>
</tr>
<tr>
<td>32KB</td>
<td>2.31</td>
</tr>
<tr>
<td>64KB</td>
<td>4.44</td>
</tr>
<tr>
<td>128KB</td>
<td>6.29</td>
</tr>
<tr>
<td>256KB</td>
<td>12.06</td>
</tr>
<tr>
<td>512KB</td>
<td>22.48</td>
</tr>
<tr>
<td>1024</td>
<td>42.26</td>
</tr>
</tbody>
</table>

Table II: Relative performance of Rabenseifner and Ring MPI_Allreduce in an HPC network (>1.00 means Rabenseifner is better; 1.00 means same performance)

To start, we demonstrate the performance impact of a lossy network on MPI_Allreduce in direct comparison against a traditional lossless HPC network. We repeat the experiment described in §III-A except, instead of relying on lossless CBFC transmission, we employ two popular lossy TCP protocols, TCP NewReno and DCTCP. For brevity, results for 256, 512, and 1024 nodes are shown in Table III.

From the data, we make two major observations:

1) The performance drop off of Rabenseifner relative to Ring is more severe with Ring clearly outperforming Rabenseifner at a number of message sizes.
2) The performance inflection at large scales previously noted occurs earlier for both message and job size.

Deeper analysis of observation 1 reveals that this earlier performance transition is the result of packet drops. Both TCP algorithms handle Ring’s traffic (i.e., single communication partner) very well. In fact, no packet drops were observed when executing Ring MPI_Allreduce. In contrast, each process in a Rabenseifner MPI_Allreduce has many communication partners, which for larger message sizes, do result in packet drops, thus contributing to its earlier performance drop off.

Analysis of observation 2 reveals that Ring scales more poorly than Rabenseifner in the presence of congestion. Fig.

IV. MPI_Allreduce in Cloud DCNs

This section describes our analysis of MPI_Allreduce in cloud DCNs typically defined as fat-trees employing lossy congestion control protocols such as TCP NewReno.

A. Preview of Results

In this subsection, we briefly preview the key findings and results to be explained in the subsequent subsections.

1) HPC to DCN: Observations show that moving from an HPC to a cloud environment can incur significant performance penalties (up to 41%) as a result of lossy network behavior and suboptimal congestion control reaction for these synchronization algorithms (§IV-B).
2) Algorithmic Tradeoffs: Performance benefit of various algorithms and phase implementations (§IV-C).
3) Congestion Control Implications: We demonstrate the implications of NewReno vs. DCTCP. Specifically, when one outperforms the other, the trade-offs between proactive vs. reactive congestion mechanisms, the impact on performance for over-throttling injection, and the potential for performance unfairness (§IV-C).

B. Comparison with HPC Network Performance

<table>
<thead>
<tr>
<th>Message</th>
<th>Number of Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>TCP NewReno</td>
</tr>
<tr>
<td>16KB</td>
<td>256</td>
</tr>
<tr>
<td>32KB</td>
<td>14.74</td>
</tr>
<tr>
<td>64KB</td>
<td>8.90</td>
</tr>
<tr>
<td>128KB</td>
<td>5.01</td>
</tr>
<tr>
<td>256KB</td>
<td>2.78</td>
</tr>
<tr>
<td>512KB</td>
<td>1.69</td>
</tr>
<tr>
<td>1MB</td>
<td>1.08</td>
</tr>
<tr>
<td>2MB</td>
<td>0.82</td>
</tr>
<tr>
<td>4MB</td>
<td>0.75</td>
</tr>
<tr>
<td>8MB</td>
<td>0.74</td>
</tr>
<tr>
<td>16MB</td>
<td>0.67</td>
</tr>
<tr>
<td>32MB</td>
<td>0.64</td>
</tr>
<tr>
<td>64MB</td>
<td>0.80</td>
</tr>
<tr>
<td>128MB</td>
<td>0.92</td>
</tr>
</tbody>
</table>

Table III: Relative performance of Rabenseifner and Ring MPI_Allreduce in a cloud network (>1.00 means Rabenseifner is better; 1.00 means same performance)
3 shows the relative increase in job completion time as message size increases (e.g., the percentage increase in job completion time of Rabenseifner when moving from 16KB to 32KB messages). Ring experiences very little increase in run time at small messages because message count dominates. Note that Ring’s absolute run time is significantly worse than Rabenseifner, however. Once congestion occurs, we see approximately linear increase in run time as message size increases. In contrast, we see a steady increase in relative job completion time for Rabenseifner culminating in a sharp increase once congestion (as indicated by packet drops) occurs. Beyond this message size, the network is relatively bandwidth limited due to congestion, and we see Rabenseifner’s logarithmic message count scaling perform better than Ring’s linear scaling. These trends hold true for all cases.

Finally, we examine raw performance between HPC and Cloud performance of MPI_Allreduce. Fig. 4 shows the relative change in job run time when moving from an HPC environment to a cloud environment. In all cases, moving to a cloud environment increases job run time (i.e., decreases performance). Irrespective of the congestion control algorithm, Ring’s performance can be divided into two categories:

1) In situations where congestion does not occur, cloud Ring job run time is essentially identical to HPC Ring.
2) Up to 30% increase in job completion time, that slowly narrows as message size increases

Also, Rabenseifner’s traits can be divided into two categories:

1) In situations where congestion does not occur, cloud Rabenseifner job run time increases between 7% to 41% depending on message size and congestion protocol. Notably, TCP NewReno outperforms DCTCP.
2) A large, and somewhat erratic performance penalty occurs—due to the congestion build-up, drops and retransmissions. The congestion algorithm stabilizes performance when the message sizes become large enough to represent a steady, longer-lived flow.

These observations show that moving from an HPC environment to a cloud environment can incur significant performance penalties. Thus, it is important to be cognizant of these potential penalties when migrating these workloads to the cloud where environmental factors may be less controllable.

C. Congestion Control Implications

In this subsection, we present MPI_Allreduce performance when operating in a DCN environment with lossy fabric. Specifically, we look to evaluate the performance implications on job completion time when different congestion protocols are used in an untapered topology (as presented in Fig. 2). Focusing on an untapered network first enables us to better contrast the performance differences before adjusting...
for conditions (such as tapering) that would magnify them. We begin by extending the algorithm comparison between Rabenseifner and Ring variants for two well-established congestion protocols (TCP NewReno and DCTCP). DCTCP utilizes ECN to proactively react to congestion.

Fig. 5 illustrates the performance results for a 1024-node MPI_Allreduce using DCTCP and ECN threshold set to 70%. We evaluate different MPI_Allreduce algorithms for different message sizes—specifically medium to large message sizes as it is under these circumstances where packet drops occur and the specified congestion control activates to ensure reliability. The X-axis depicts the message size while the Y-axis represents the job completion time (normalized to the Ring algorithm). Each bar represents a different algorithm. The bars depicted by Ring/Rabenseifner and Rabenseifner/Ring represent slightly modified versions of the Rabenseifner algorithm where Ring is used for the reduce-scatter or all-gather phase instead of recursive doubling (default). The results indicate that as the message size increases (starting at 8 MB), the Ring algorithm actually outperforms (lower is better) all the other algorithms (because the others are above 1x relative to Ring). This is due to the fact that it does not experience any significant congestion as to incur packet drops and, as a result, does not experience any of the congestion throttling that would occur. For such large job size (1024 nodes), the smaller message sizes (1-4MB) experience less congestion (no packet drops) as the communication is more fine-grain, and so the extra algorithmic steps of Ring (and the inherent latency cost) cause it to perform more poorly than Rabenseifner. In contrast, the other collects algorithms experience more packet drops as a result of congestion due to their more varied communication. This, in turn, causes throttling by the DCTCP congestion control algorithm with the result being poorer performance. As message size increases further, Ring’s performance degrades due to poor scaling allowing Rabenseifner and the other algorithms to reach near parity in job completion time.

D. Placement Strategies

In this subsection, we evaluate collects performance against two job placement strategies: Linear vs. Randomized. The motivation stems from the fact that both HPC and Cloud environments are susceptible to job interference (among concurrently running jobs) and fragmentation over time (in terms of which servers get allocated for jobs). However, system heterogeneity is likely a more common occurrence in DCNs as they are augmented organically over time with different hardware and various vendors. In addition, Cloud system administrators lack direct control over specific protocols and configurations where such operating assumptions are controlled by the virtual machine (VM) of the executing applications. The former approach would be considered the more ideal scenario but also the least realistic, while the latter is the more likely given that a user isn’t provided such layout guarantees when running their workload in a DCN. The implications of placement on workload performance ultimately depend on the degree of locality of the communication, where co-location across the same rack/Top-of-Rack (ToR) switch is desirable in order to minimize the latency and delays due to network congestion. Furthermore, the underlying algorithm utilized ultimately dictates the degree and frequency of congestion occurrence based on the synchronization phases.

Fig. 6 illustrates the performance for several MPI_Allreduce algorithms with the aforementioned placement options. The colored bars show performance when the processes are scheduled randomly while the hatched overlay bars show performance when the processes are scheduled linearly. We make two major observations:

1) **Rabenseifner benefits from random placement.** When scheduled linearly, the processes are spread over a small number of switches. Because all processes in a Rabenseifner MPI_Allreduce communicate with each other, this creates significant congestion on the links connecting the ToR switches to the spine. In contrast, when processes are scheduled randomly, there is less concentration of traffic on any particular ToR-to-spine link, resulting in less congestion and better performance. The astute reader will note the lower concentration of traffic is due to fewer processes per ToR in the random placement case. This, however, does not invalidate the observation that spreading traffic across a greater number of switches reduces the hot spots that manifest congestion.

2) **Ring benefits from linear placement.** In contrast to Rabenseifner, each process in a Ring MPI_Allreduce only communicates with one other process. By linearly scheduling processes, there are at most only two processes per ToR that communicate with processes on another switch. For example, if processes 0-7 are scheduled on ToR switch 0 and processes 8-15 are scheduled on ToR switch 1, only processes 0, 7, 8, and 15 communicate with the other ToR switch. All remaining processes communicate within the switch to which it is connected.

E. Multi-Tenancy (AllReduce Only)

In this subsection, we demonstrate collects performance operating under conditions of multi-tenancy. Ensuring performance guarantees to different customers deploying and running workloads in the Cloud has been a critical challenge
for CSPs. CSPs are often driven to over-provision resources in an effort to mitigate performance interference and ensure quality guarantees to their customers. However, naive isolation techniques such as offering dedicated resources are increasingly cost-prohibitive and insufficient over time as network administrators deal with workload fragmentation. Therefore, it is critical to such implications on performance, and for our analysis we presume a completely randomized placement.

We construct 2 different scenarios for evaluation under a 1024 Node Cluster (See tapered topology depicted in Fig. 2):

1) 4x256 AllReduce with DCTCP (homogeneous workload with state-of-art used homogeneous protocol).
2) 4x256 AllReduce with 2 using NewReno and 2 DCTCP (homogeneous workload with heterogeneous protocol).

In both cases each job is distributed randomly across the different nodes in the cluster, and we look to evaluate the Job Completion Times across the 4 concurrently executing jobs within each case to evaluate interference and fairness implications. Fig. 7 illustrates the different multi-tenant scenarios enumerated, comparing Rabenseifner versus Ring AllReduce algorithms. Sub-figures 7a and 7b illustrate scenarios 1 and 2 respectively. The X-axis indicates the message size of the AllReduce, while the left Y-axis references the time (in microseconds) for the minimum, average, and maximum job completion times for the 4 concurrent AllReduce jobs. The right Y-axis depicts the calculated Range (MAX Job Completion Time - MIN Job Completion Time) / Average Job Completion Time. For example, a value of 0.25 indicates 25% performance variability between the slowest and fastest job. This job variability is exhibited on the right Y-axis.

The results also demonstrate the increase in Job Completion Time variability when using Rabenseifner with increasingly large message sizes. In contrast, the Ring algorithm exhibits minimal unfairness across Jobs despite increasing the message size. Furthermore, we observe that the homogeneous case (1), experiences reduced variability as compared to the hybrid case (2). As previously described, DCTCP utilizes ECN to proactively vary the injection-rate based on detected congestion/queue-build up in the network. This occurs even if the source sender is not a contributor towards the congestion (but receives an acknowledgment marked with ECN echoed on the reverse path). As a result, all senders will consequently throttle back based on the rate of ECN-marked acknowledgments received. In contrast, in the hybrid case, some jobs utilize the ECN feedback while the others do not (i.e. NewReno). In such instances, NewReno detects congestion in the network via packet timeouts and out of sequence received acknowledgments (indicating packet drops). Therefore, in cases where there is congestion build up, but not yet to the extent of experiencing packet drops, then the result behavior will be such that the DCTCP enabled senders will throttle back while the NewReno enabled flows will not. As a consequence, this increases the variability observed of Job Completion Times. For instance, using the metric described we observe 0.25 for case 1 as compared to 0.29 for case 2. This finding provides further indication that job unfairness due to co-existence of different congestion control protocols is a critical consideration that CSPs must contend with during large scale deployments, and complements some of the analysis by Cardwell et al. [23]. These results offer insights to recommended algorithms to use depending on the deployment and users’ tolerance to variability.

F. Multi-Tenancy (AllReduce w/ Background Traffic)

In the previous subsection's experiments, we were evaluating scenarios where more than one instance of MPI_Allreduce are being executed in the DCN. However, in a multi-tenancy environment, the type of applications that can be executed concurrently with an MPI_Allreduce application is likely to be diverse. Therefore, we also evaluate a scenario where MPI_Allreduce is executed concurrently with a range of generic background (BG) traffic. For this representation, we simulate an instance of MPI_Allreduce on 256 random nodes in the same tapered topology Fig. 2, while the remaining nodes are executing background (BG) traffic using NewReno that will interfere the MPI_Allreduce traffic. For this analysis, we focus on the performance of the collective as/how the BG traffic intensity and pattern varies with two different scenarios. The background traffic exhibited here is uniform random (UR), which represents well balanced BG traffic that is already in the network. In this workload, each corresponding sender randomly picks a new destination process to communicate a message of fixed size to. Fig. 8 shows the job completion time of MPI_Allreduce instances using different algorithms with varying intensity of BG traffic (20% and 80% of bisection bandwidth). There are three key takeaways from these results:

1) Ring outperforms Rabenseifner and DCTCP outperforms NewReno in the presence of significant congestion.
2) The larger number of active connections in Rabenseifner exposes it to increased susceptibility to flow collision (and hence increased variability between synchronizing exchanges). Ring only maintains 2 active connections (a process connection to send to and receive from).
3) Rabenseifner suffers from stale congestion information between its different phases as it transmits different size chunks at different rounds of exchanges between the same pair of processes—causing the actual injection rate to be misaligned with curen network conditions.

V. RELATED WORK

In this section, we describe some previous work, primarily highlight prior efforts on optimizing collective communication and their relation to DCNs and/or DeepLearning workloads. Nvidia has released a collectives communication library (NCCL) that implements both multi-node and multi-GPU collectives operations optimized to achieve high-bandwidth and low latency over PCIe and their proprietary NVLink high-speed interconnect. [24]. NCCL supports a variety of AllReduce algorithms including hierarchical ring-based and double binary-Tree-based that are effective at achieving high-bandwidth and low-latency across GPUs. Additional work
from Nvidia explore the performance benefit of an in-network architecture for accelerating collectives’ operations. Klenk et al. [25] propose specific switch designs to support performing reduction operations coupled with multicasting for DL training on shared-memory systems. While their work demonstrates significant performance opportunity, their proposal is targeted towards shared-memory architectures and not scale-out networks. For scale-out networks, Graham et al. describes Mellanox’s SHArP [26], a technology designed to offload collective operation processing, supported in some of their product lines. Specifically, the switch uses in-network trees to perform data reductions from a group of sources, and then distribute the final result. They demonstrate significant performance improvement (for small/medium operations).

Graham et al. also more recently describe an improved hardware based streaming aggregation [27] built on top of the SHArP technology to better enable in-network reductions for significantly larger message sizes as well as interoperability with DL-Poly and PyTorch (DL Training Frameworks). Key themes of combining in-network computations with efficient multicasting are applicable, and is an active research area. Alifatata et al. propose a generic API structure to improve collective operations in the Cloud [28]. Specifically, this framework attempts to exploit inherit locality across co-located MPI processes in the same node or rack and uses software-defined networking to minimize cross rack communication. This effort lends credibility to the problem needing to minimize communication overhead of collective operations in a cloud environment. Luo et al. similarly propose an optimized communication library (PLink) to accelerate Cloud-based distributed DL Training [29]. The library probes the physical network and then generates a fitted aggregation plan to leverage locality and dynamically updates based on changes in the network conditions. Both works are complementary to our analysis of evaluating the performance of different network congestion protocols and collective algorithm scaling. Our analysis can directly influence the chosen algorithm selected by the library based on probed locality and network conditions.

Job-Interference for HPC systems has also been an active research area. Despite all the innovations and customized system fabrics, performance variability due to job-interference remains a critical problem. While several works explore this in great detail for dragonfly systems [30], [31], extensive performance characterization and analysis for DCN remains a more elusive problem given the closed-nature of CSPs to disclose their workload patterns and network configurations. There are also several prior works that propose specific allreduce algorithmic optimizations specific to mitigating performance loss in specific latency sensitive use-cases or topologies. Canny et al. propose Butterfly Mixing AllReduce that interleaves the reduction operation with iterative stochastic gradient updates to avoid some synchronization delays, but the tradeoff is that may worsen time-to-convergence [32]. Mikami et al. propose a 2D Torus AllReduce [33] to minimize communication overhead of Ring operating on 2D Torus—by implementing partial reductions along both the horizontal and vertical dimensions before aggregating—an intuitive solution, but not directly usable in non-Torus networks.

VI. CONCLUSION

In this paper, we characterize the simulated performance of key collectives’ algorithms operating in a DCN environment. Specifically, we evaluate the 1) implications of different congestion control algorithms 2) highlight contrasting algorithmic scaling inefficiencies with respect to data-set size and topology size as well as 3) sensitivity to placement & multi-tenancy.
We focus on AllReduce collective with its various algorithmic implementations as it is the most pre-dominant ones used in HPC and Deep Learning Training workloads. This work demonstrates that previously defined algorithmic expectations established and validated with homogeneous, well-structured networks such as HPC, do not necessarily translate over to the Cloud where the structural assumptions are vastly different. Furthermore, next-gen implementations of HPC & AI indicate significantly larger data footprints, thus reinforcing the need for better scalability analysis and characterization. As a result, we demonstrate that previously established expectations and guiding recommendations need to be revisited, and find that algorithms such as Ring Allreduce once deemed suboptimal for increasing topology and workload scales are comparatively more efficient compared to Rabenseifner. Possible future work vectors include network optimizations such as switch-based hints for improved congestion reaction, and more efficient in-network compute for large networks.

REFERENCES


Enabling Extremely Fine-grained Parallelism via Scalable Concurrent Queues on Modern Many-core Architectures

1st Poornima Nookala  
Department of Computer Science  
Illinois Institute of Technology  
Chicago, Illinois, United States  
pnookala@hawk.iit.edu

2nd Peter Dinda  
Department of Computer Science  
Northwestern University  
Evanston, Illinois, United States  
pdinda@northwestern.edu

3rd Kyle C. Hale  
Department of Computer Science  
Illinois Institute of Technology  
Chicago, Illinois, United States  
khale@cs.iit.edu

4th Kyle Chard  
Department of Computer Science  
University of Chicago  
Chicago, Illinois, United States  
chard@uchicago.edu

5th Ioan Raicu  
Department of Computer Science  
Illinois Institute of Technology  
Chicago, Illinois, United States  
iraicu@cs.iit.edu

Abstract—Enabling efficient fine-grained task parallelism is a significant challenge for hardware platforms with increasingly many cores. Existing techniques do not scale to hundreds of threads due to the high cost of synchronization in concurrent data structures. To overcome these limitations we present XQueue, a novel lock-less concurrent queuing system with relaxed ordering semantics that is geared towards realizing scalability up to hundreds of concurrent threads. We demonstrate the scalability of XQueue using microbenchmarks and show that XQueue can deliver concurrent operations with latencies as low as 110 cycles at scales of up to 192 cores (up to 6900× improvement compared to traditional synchronization mechanisms) across our diverse hardware, including x86, ARM, and Power9. The reduced latency allows XQueue to provide orders of magnitude (3300×) better throughput that existing techniques. To evaluate the real-world benefits of XQueue, we integrated XQueue with LLVM OpenMP and evaluated five unmodified benchmarks from the Barcelona OpenMP Task Suite (BOTS) as well as a graph traversal benchmark from the GAP benchmark suite. We compared the XQueue-enabled LLVM OpenMP implementation with the native LLVM and GNU OpenMP versions. Using fine-grained task workloads, XQueue can deliver 4× to 6× speedup compared to native GNU OpenMP and LLVM OpenMP in many cases, with speedups as high as 116× in some cases.

Index Terms—concurrent data structures, fine-grained parallelism, lock-free, lock-less, queues, parallel runtime, tasks

I. INTRODUCTION

Task parallelism is an increasingly important class of parallelism in which computation is broken into a set of inter-dependent tasks which may be executed concurrently on various cores. The execution models of many parallel languages and libraries [1]–[6] rely on such task parallelism. For example, OpenMP [1] has evolved to a task-centric model to enable parallelization of applications where units of work are generated dynamically. When a task is created by some thread, it is conceptually queued for execution by a future available thread. Software dataflow languages [2], [3] similarly include a runtime that executes a dynamically unfolding task graph in which tasks are scheduled via concurrent queues. To achieve strong scaling and high levels of parallelism, today’s parallel languages and execution models are moving to tasks with finer granularity. One reason for this is that as core counts per node increase, applications need to support over-decomposition in order to improve performance, hide latency caused by blocking operations, and achieve maximum speedup. This and other drivers produce the same outcome: tasks and their dependencies need to be managed at sub-microsecond timescales.

Queues are an integral component of tasking runtime systems and as task granularity decreases, execution performance is increasingly dependent on queue performance. Of particular interest here are single producer, single consumer (SPSC) and multiple producer, multiple consumer (MPMC) concurrent queues. The queue itself contains tasks, typically in the form of pointers (to task objects). Threads running concurrently can interleave instructions in many ways and a shared data structure needs to be carefully protected to avoid races. Concurrent SPSC and MPMC queues are no exception and require that their state (e.g., head, tail and data) be protected with a synchronization mechanisms, such as mutual exclusion locks (mutexes), spinlocks, semaphores, or atomic primitives.

A second approach to concurrent queues is to avoid separate synchronization by incorporating race-avoidance directly into the data structure itself. This also has the benefit of avoiding common concurrency bugs (e.g., deadlocks) due to misuse...
of synchronization primitives. **Lock-free** data structures use atomic primitives, such as Compare-and-Swap (CAS) and Fetch-and-Add (FAA), to push the burden down to hardware and achieve synchronization at a finer granularity. Several libraries internally use lock-free techniques [7]–[9], but the literature has shown that it is difficult to write correct lock-free code [10]. Even more compelling are **lock-less** data structures [11], which not only avoid the use of locks, but also can avoid the need for atomic operations under certain conditions. Both lock-free and lock-less programming are challenging due to instruction and memory access reordering imposed by the compiler and the hardware, and the need to account for the memory consistency model supported by both.

In order to address the aforementioned issues, we design, implement, and evaluate XQueue, a novel lock-less concurrent queuing system with relaxed ordering semantics. XQueue is not a general-purpose MPMC queue, but rather a task queuing system aimed to improve the task management overhead in parallel runtime systems.

We make the following contributions:

1) We design and implement XQueue, a lock-less, relaxed-order MPMC queue that uses multiple queues for improved locality without using locks or atomic operations. We demonstrate the scalability of XQueue using microbenchmarks measuring latency as low as 110 cycles and throughput as high as 1 billion ops/sec across today’s largest shared-memory systems.

2) We integrate XQueue into LLVM OpenMP and evaluate the performance improvements on 5 unmodified applications (Fib, FFT, MultiSort, NQueens, and Health) from the Barcelona OpenMP Task Suite (BOTS) as well as the breadth first search (BFS) application from the GAP benchmark suite. We show that the combination of XQueue and LLVM OpenMP is capable of delivering better scalability for fine-grained task-parallel workloads with up to 6× speedup compared to native LLVM OpenMP and 1× to 4× speedup compared to GNU OpenMP in most cases, and up to 116× speedup in some cases.

### II. Motivation

Concurrent data structures have to deal with data synchronization and communication between threads. Synchronization mechanisms like mutexes, semaphores, and spinlocks are known to have significant overhead and can easily become the bottleneck to achieving high performance.

An SPSC array-based queue provides the lowest latency the microarchitecture can provide for enqueue and dequeue operations when both operations do not occur simultaneously since they do not require data synchronization or thread-to-thread communication and can benefit from data locality. A popular approach to implement parallelism in applications is to use concurrent queues for sharing work among various threads; MPMC queue is the most commonly used data structure in such cases. Thread contention on shared data, synchronization overheads, cache coherence effects, and cache misses are some of the many factors that can significantly impact the performance of MPMC queues and limit scalability. In order to show the scalability and performance of MPMC queues compared to SPSC queues, we selected five diverse systems (see Table I) from the Mystic Studb [12] that represent different architectures with large core counts. We conducted experiments on Ubuntu 18.04.3 and compiled using LLVM Clang version 11.0.0 with O3 optimization level and --march = native.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>skylake-192</td>
<td>Intel Xeon Gold 8160</td>
</tr>
<tr>
<td>skylake-48</td>
<td>Intel Xeon Gold 8160</td>
</tr>
<tr>
<td>skylla-32</td>
<td>Intel Xeon Gold 6130</td>
</tr>
<tr>
<td>skylla-16</td>
<td>Intel Xeon Silver 4110</td>
</tr>
<tr>
<td>phi-64</td>
<td>Intel Xeon Phi 7210</td>
</tr>
<tr>
<td>broadwell-16</td>
<td>Intel Xeon E5-2620 V4</td>
</tr>
<tr>
<td>haswell-12</td>
<td>Intel Xeon E5-2620 V3</td>
</tr>
<tr>
<td>haswell-8</td>
<td>AMD Naples 7501</td>
</tr>
<tr>
<td>haswell-8</td>
<td>AMD Naples 7501</td>
</tr>
<tr>
<td>ryzen-8</td>
<td>AMD Ryzen 7 1700</td>
</tr>
<tr>
<td>opteron-48</td>
<td>AMD Opteron 6168</td>
</tr>
<tr>
<td>power9-40</td>
<td>POWER9 EP73</td>
</tr>
<tr>
<td>thunderx-96</td>
<td>ThunderX 8XX ARM v8</td>
</tr>
</tbody>
</table>

**TABLE I: Testbed for evaluation from the Mystic System**

We measure the latency and throughput of a simple SPSC array-based circular queue to identify baseline performance on the latest many-core architectures. We run 1 billion enqueue operations followed by a sequence of dequeues. We measure the latency of each operation and calculate the average time per enqueue/dequeue pair. For throughput experiments, we measure the total time taken for 1 billion enqueue/dequeue operations and calculate the throughput. Results in Table II show the average latency and throughput of both enqueue and dequeue operations. We see that the latency of any operation on queues takes between 29 and 68 cycles depending on the architecture and clock frequency. Average throughput reaches 270 million ops/sec on skylake-192 machine. Although these results are significant, an SPSC queue is limited in parallel runtime systems because it cannot alone be used to implement parallelism and concurrency.

<table>
<thead>
<tr>
<th>Machine</th>
<th>Latency (cycles)</th>
<th>Throughput (ops/sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>skylake-192</td>
<td>41</td>
<td>270M</td>
</tr>
<tr>
<td>epyc-64</td>
<td>47</td>
<td>155M</td>
</tr>
<tr>
<td>phi-64</td>
<td>68</td>
<td>26M</td>
</tr>
<tr>
<td>thunderx-96</td>
<td>36</td>
<td>58M</td>
</tr>
<tr>
<td>power9-40</td>
<td>29</td>
<td>36M</td>
</tr>
</tbody>
</table>

**TABLE II: Average latency and throughput of enqueue/dequeue operations on SPSC queue**

We measure the latency and throughput of an MPMC queue by implementing the queue using a semaphore which tracks free spaces in the queue and uses pthread_mutex_lock to lock the queue during enqueue and dequeue operations. This experiment aims to quantify the poor scalability of MPMC queues using mutex locks. Each experiment enqueues and dequeues 1 billion items using an equal number of producer and consumer threads. In all experiments, we use a round-robin pinning of threads, with producer and consumer threads being on the same core (but distinct hyperthreads) which...
can result in better cache utilization, thereby reducing costly memory accesses.

(a) Latency
(b) Throughput

Fig. 1: Performance of MPMC queue operations

Figures 1a and 1b show the latency and throughput, respectively. Our results indicate that latency can reach up to millions of cycles under high contention, and throughput can drop down to as low as 311,329 operations per second (aggregate over all threads). For the Skylake-192 system, which had the best single core performance at 270 million operations/sec, the MPMC approach yielded only 810 operations per second at a 384-thread scale (a $333,333 \times$ loss of performance). The fastest MPMC queue throughput at any scale reached just 5 million operations/sec.

We were not surprised by these findings, as the fundamental problem stems from the cost of synchronization. In prior work, we studied the cost of synchronization mechanisms from low to high levels of concurrency, and found that none of these mechanisms offered scalable solutions beyond single digit concurrent threads [13]. Use of such concurrent data structures in modern parallel runtimes have significant overheads for managing extremely fine-grained tasks. For example, when computing the 44th Fibonacci number recursively using LLVM OpenMP, the runtime overhead dominates the overall execution time by consuming over 90% of CPU time for synchronization and scheduling. These findings motivated our investigation into methods to eliminate synchronization mechanisms in order to unleash the full performance of many-core architectures under high concurrency.

III. XQUEUE- SCALABLE CONCURRENT QUEUES ON MODERN MANY-CORE ARCHITECTURES

XQueue is motivated in large part by the significant latency gap observed with SPSC and MPMC models (Section II). A simple concurrent SPSC queue can enqueue and dequeue items in less than 100 cycles. Independent SPSC queues per core could, in theory, scale linearly with increasing core counts. Thus, we believe that an MPMC lock-less queue can be built using SPSC queues by manipulating the task/data flow carefully.

We introduce XQueue, a novel lock-less MPMC, out-of-order queuing mechanism that can scale up to hundreds of threads. XQueue uses B-queue [14] as a building block. B-queue is a concurrent SPSC lock-free queue designed for efficient core-to-core communication. It is implemented without using any locks, atomic operations, or barriers. The latency of queue operations in B-queue is as low as 20 cycles. B-queue uses batching where both producer and consumer detect a batch of available slots that are safe to use. Batching avoids shared memory access and therefore improves performance. Several fast SPSC queues have been proposed in recent years [15]–[17] and we aim to demonstrate that XQueue can be built with any fast and scalable SPSC queue.

Figure 2 shows the architectural of XQueue on a 4-core system. The key idea here is to have $N$ SPSC concurrent queues per worker if there are $N$ workers. There is one master queue and $N - 1$ auxiliary queues per worker, with $N$ (equal to number of workers) producers adding items into master queues. Every item is a void pointer that represents a task where a task could be a function pointer or data pointer. One worker exists for dequeueing tasks from the master queue as well as the auxiliary queues. A worker first tries to dequeue a task from the master queue. If a task is dequeued successfully, it is processed immediately. The item when processed can generate one or more items to be enqueued into the auxiliary queues of the other CPU cores. Every worker distributes work to auxiliary queues in a round-robin fashion as shown in Figure 2. A worker then tries to dequeue an item from its auxiliary queues and dequeued items are processed immediately.

A simplified version of pseudocode for worker logic is outlined in Algorithm 1. Since all queues in XQueue are concurrent SPSC queues, producer and consumer threads can act concurrently processing items in the queues. The strategy of distributing work across queues (as shown in Figure 2) ensures that there is a only a single producer and single consumer for every queue at any point in time. Due to this design, locks can be completely avoided thereby reducing the latencies of queue operations and improving overall performance.

A. Load balancing

In most parallel programming systems, it is a common scenario to use multiple queues, one per worker, with work produced and consumed locally by the workers/threads. Load balancing is commonly achieved by using techniques like work stealing [18], [19]. While XQueue also uses multiple queues, it balances load by the virtue of its design with $N$
Data: id ← coreId; next ← nextCoreId;
while 1 do
ret ← dequeueFromMaster(id, item);
if ret = SUCCESS then
retItem ← processItem(item);
enqueueToAuxiliary(next, retItem);
next ← (next + 1)%numCores;
if next == id then
next ← +1
/* do not enqueue to self */
end
end
ret ← dequeueFromAuxiliary(id, item);
if ret = SUCCESS then
retItem ← processItem(item);
enqueueToAuxiliary(next, retItem);
end
next ← (next + 1)%numCores;
if next == id then
next ← +1
/* do not enqueue to self */
end

Algorithm 1: Worker logic.

queues per core and consumer threads inserting items into the auxiliary queues of all the other cores. This architecture enables distribution of task graphs to multiple threads with minimal overhead due to the lock-less design as compared to the state-of-the-art work stealing techniques which primarily use locks or atomics to achieve synchronization.

In a task-parallel program, tasks can be modeled as a Directed Acyclic Graph (DAG) which can be traversed based on inter-dependencies between the tasks. Task graphs have a pool of ready tasks which can be processed by threads and subtasks can be generated. The master and auxiliary queues and the communication between them is modelled after the dynamic execution of a program where a task can generate subtasks. In the case of XQueue with $N$ workers and $N$ queues per worker, as shown in Figure 2, we employ a ring buffer topology for communicating between queues. Essentially, the consumer thread of every set of queues acts as a producer thread of $N-1$ auxiliary queues of all the other threads. This pattern of task distribution ensures optimal load balancing in terms of the number of tasks processed per worker. However, this may not be the best fit for every scenario for various reasons, such as data locality, task dependencies, and per task execution time. Optimal allocation of work among various threads is known to be NP-hard, but, in the case of XQueue, depending on the nature of work, the topology of connections between queues and task distribution strategy can be changed to achieve best performance.

The load balancing mechanism in XQueue can be considered as a push-based mechanism as opposed to pull-based work stealing approach. This primary difference impacts how initially imbalanced workloads are handled. For example, consider the case of Fibonacci. Execution starts with a single task which recursively unfolds the DAG as execution progresses. In the work stealing approach, idle workers randomly try to steal tasks from other workers. This results in several failed steals and coupled with the cost of locking for every steal, incurs significant overhead. On the other hand, the push-based approach of XQueue handles this efficiently with its round-robin distribution without the use of locks, thus incurring minimal overhead. We discuss the advantages and disadvantages of this approach in Section IV.

On modern many-core architectures, it is common to have multiple Non-uniform memory access (NUMA) zones which impact the latency of memory operations from various cores. In XQueue, every worker allocates queues in its respective NUMA zone. This ensures that any memory reads and writes from various threads have the lowest latency possible. However, when tasks propagate through auxiliary queues in the system, the latency of memory read/write is higher across NUMA zones. With XQueue’s ring buffer design across N cores with N queues, some latency is unavoidable due to the underlying architecture.

In summary, there is a lot of flexibility for defining the topology for task distribution statically and dynamically during program execution with XQueue. If the nature of the DAG and data access patterns are known, the task distribution can be tuned to achieve best performance as compared to state-of-the-art work stealing approaches.

B. XQueue Integration with the OpenMP Runtime

In order to extend our research to real systems, we integrated XQueue into OpenMP [20] to enable execution of unmodified OpenMP programs using XQueue. OpenMP’s tasking model provides a way to efficiently parallelize dynamic task graphs and recursive algorithms. Several implementations of OpenMP exist: GNU OpenMP (for GCC) [21], LLVM OpenMP [20], and Intel OpenMP. We chose to integrate XQueue into the LLVM OpenMP due to its open source code and its superior performance as compared to GNU OpenMP with fine-grained tasks [22].

Implementation: In the LLVM OpenMP tasking implementation, every thread owns a queue and the enqueue/dequeue operations are protected by locks implemented using Lamport’s bakery algorithm. We replaced the task queues in OpenMP with multiple SPSC queues per worker to model XQueue. OpenMP implements a work-stealing scheduler. Every thread first checks it’s own queue for tasks. If no tasks are found, a thread is randomly chosen to steal a single task. We replaced the work stealing scheduler with the scheduler for XQueue as shown in Algorithm 1. In our XQueue-based OpenMP implementation, every thread checks it’s own queue for tasks. If no tasks are found, the scheduler checks all auxiliary queues. This process of checking the master queue and auxiliary queues is repeated until a termination condition is satisfied.

Optimizations: We applied few optimizations to the XQueue system during integration with the OpenMP runtime. Since the core design of XQueue is to have multiple queues per worker, at higher thread counts (hundreds), the latency of checking all auxiliary queues can become significant and reduce the overall performance. To solve this issue, we implemented a hinting mechanism where every producer stores the ID of the last queue to which the task was pushed. This hint can possibly be over-written by multiple threads writing to various queues, however this simple mechanism reduces the latency of checking auxiliary queues many times.

IV. PERFORMANCE EVALUATION

We evaluate the performance of XQueue using synthetic and real workloads. For the purposes of evaluating XQueue inde-
pendedently, we developed a prototype parallel runtime system that can process a dynamic task graph with task dependencies using XQueue. We first evaluate XQueue individually using a series of micro-benchmarks. We deployed XQueue on 13 systems (Table I); we then picked the system with the highest number of cores, the skylake-192 with 192-cores and 8 NUMA zones to conduct deeper analysis.

A. Experiment Setup

We implemented three systems for the micro-benchmark evaluation:

1) XQueue (SPSC) uses a single SPSC queue per worker.
2) XQueue (MPMC) uses an MPMC queue with a master queue per worker.
3) XQueue (Cilk Deque) uses a Cilk deque [5] with a separate queue per worker.

Cilk deque is implemented as part of Cilk 5 multi-threaded language [5] and uses a shared-memory, mutual-exclusion protocol called the THE protocol [23] for implementing locks. This mechanism of locking is about 25% faster than hardware locking primitives.

For the macro-benchmarks, we use the XQueue-enabled LLVM OpenMP implementation with $N$ queues per worker and $N$ workers. We compare it with the native LLVM OpenMP and GNU OpenMP libraries.

B. Micro-benchmark Performance Results

In each experiment we perform 1 billion enqueues/dequeues concurrently by varying the number of threads. We consider a single operation to be the act of dequeing an item from the master queue and executing the function to which that item points to. The function performs a single NOP operation. The X-axis on all the figures represents the number of producers/consumers.

Figure 3a shows the latency of queue operations on XQueue using lock-less queue. Each queue operation takes around 110 to 400 CPU cycles on average on all architectures considered. ARM ThunderX shows the lowest latency and IBM Power9 shows the highest latency in these micro-benchmarks. Intel processors Skylake, Haswell, Broadwell and Xeon Phi show latencies in the range of 180 to 300 CPU cycles on average. The standard deviation is low across all architectures indicating that XQueue with lock-less queue can scale up to hundreds of threads with latencies as low as 110 to 400 cycles.

Figure 3b compares the latency of XQueue (SPSC) with Cilk Deque and MPMC queues on skylake-192. Here, Cilk Deque/MPMC is a single queue shared across all the workers. With 192 producers/consumers, latency of MPMC queue is $13 \times$ the latency of Cilk deque. Cilk deque’s Dijkstra-like locking mechanism achieves much lower latency than locks implemented using hardware locking primitives. However, the latency is much higher compared to XQueue which does not use any locks. It is noteworthy that XQueue has relatively constant latency as we increase the number of threads by two and half orders of magnitude, while Cilk deque and MPMC show significant latency increases over the same scale.

Figure 3c is a log-log plot showing the throughput of XQueue using lock-based and lock-less queues on the skylake-192 system. The throughput achieved on this system with XQueue with lock-less queue is 1 billion operations per second with all hyper threads being utilized. For XQueue using lock-based queue, the average throughput achieved is 200 million operations per second and 397 million for the Cilk deque. In the case of MPMC queue, each mutex lock is held for short intervals and contention is low, but acquiring the lock has a cost which explains the $5 \times$ gap in performance as compared to XQueue with lock-less queue. Cilk deque also incurs a cost for acquiring and releasing the lock (a $2.5 \times$ gap), although the cost is lower compared to mutex-based locks. As noted in Section II for MPMC queue, with high contention on the mutex lock with more than 8 threads, throughput drops to about 300K operations per second on skylake-192 with 384 threads. In case of Cilk deque, the throughput drops to 4 million operations per second. This clearly shows a $3300X$ gap in throughput between XQueue with lock-less queue and single lock-based queue with hundreds of threads.

The results obtained from micro-benchmarks using XQueue with lock-less queue and lock-based queue are significant and show that this architecture can scale to at least hundreds of threads with any scalable concurrent SPSC queue implementation. It can be noted that these micro-benchmarks do not take into consideration the cache effects of task distribution to other cores in XQueue since there are no auxiliary queues. Hence, this benchmark shows the lowest latency and highest throughput that can be achieved, providing a baseline.

C. Macro-benchmark Performance Results

To quantify the improvements in real application workloads, we evaluate the speedup achieved using XQueue-enabled LLVM OpenMP as compared to the native LLVM OpenMP and GNU OpenMP libraries. We evaluate five out of nine applications from the BOTS benchmark suite [24]: Fibonacci, FFT (Fast Fourier Transform), Multisort, NQueens and Health. Results are shown in Figure 4. We also evaluate the breadth first search application from the GAP benchmark suite [25] with real-world social network graphs such as those from Friendster and Twitter. Results are shown in Figure 6. The application workloads are summarized in Table III.

| Application | Inputs(S|M|L|XL) | Highest Task Count |
|-------------|-------------|-------------------|
| Fibonacci   | 44, 46, 48, 50 | 40.7B             |
| FFT         | 134M, 268M, 536M, 1B | 128M             |
| Multisort   | 134M, 268M, 536M, 1B | 14M              |
| NQueens     | 14, 15, 16     | 1.1B              |
| Health      | small, medium, large | 126M             |
| BFS         | friendster     | 79M               |
| BFS         | twitter        | 40M               |

Fibonacci (Fib) computes the Nth Fibonacci number using recursive parallelism. While Fib is hardly a critical parallel application, it does have extremely fine-grained tasks (e.g., addition of two numbers) with extremely large number of tasks, and thus exposes the limits of a tasking runtime in
terms of granularity. Figure 4 shows the results obtained on skylake-192. OpenMP with XQueue achieves $3 \times$ speedup as compared to the native LLVM and GNU versions for Fib(50). The performance gap increases with problem size due to the increase in overhead of locking operations in the native OpenMP versions with more fine-grained tasks. Further analysis using Intel Vtune Profiler showed that about 50% of the execution time is spent in these operations which includes waits and atomics, where as this overhead is negligible in the XQueue version due to the lack of locks or atomics. The overall runtime overhead for managing fine-grained tasks of this application reduced from over 90% to 29% of the CPU time when using XQueue.

Multisort sorts 32-bit randomly generated numbers using a fast parallel sorting variation of mergesort. It uses a recursive algorithm with a base condition of 2048 numbers and they are sorted using serial quicksort and insertion sort is used for arrays with less than 20 elements. The application scales well up to 96 threads for all the runtimes and XQueue is faster for all problem sizes with $1.97 \times$ speedup for the largest problem size. However, the performance drops by 50% at 192 threads. As shown in Figure 4, XQueue achieves similar performance compared to LLVM and GNU versions using 192 threads. LLVM and GNU versions of OpenMP exhibit high CPI (cycles per instruction) rate (0.5 for XQueue vs 24 for both LLVM and GNU for the largest problem size) which is the result of waits, atomics, and locks in the GNU/LLVM versions. However, since this application is heavily memory-bound, the benefits of avoiding locks and lower CPI in XQueue are outweighed by the data movement across cores, thereby resulting in no performance benefit.

Health simulates the Columbian Health Care System [26]. A list of potential patients in a village with one hospital are simulated with several possibilities of getting sick, needing treatment or reallocating to an upper level hospital. Every village being simulated is run as a task. The different probabilities at each step cause indeterminism and load imbalance. On skylake-192, the performance of this application is heavily impacted due to remote memory accesses for moving the large data across NUMA zones. Despite some load imbalance, XQueue achieves $6 \times$ speedup compared to LLVM variant and $4 \times$ speedup compared to GNU variant using the large input data.

Fast Fourier Transform (FFT) computes the 1D FFT of a vector with $N$ complex values using the Cooley-Tukey Algorithm. This algorithm recursively divides the FFT into several smaller Discrete Fourier Transforms (DFTs) creating multiple tasks at each step. Although the XQueue version has the advantage of reduced overhead due to lock-less queues, the task distribution suffers due to the static round-robin placement of tasks resulting in similar overall execution time as compared to other versions of OpenMP. Figure 5 shows the timeline view of the OpenMP parallel region for the largest problem size, where green represents effective work and black represents the spin/wait/overhead time introduced by load imbalance. It is noteworthy that OpenMP with XQueue with worse load balancing can still achieve slightly improved performance (between $0.9 \times$ to $1.2 \times$) due to the smaller overheads incurred by avoiding locks.
on an $N \times N$ chess board such that no queens can attack each other. The algorithm prunes certain branches of the tree that cannot reach the solution which creates load imbalance. Figure 4 shows that the XQueue OpenMP achieves 4X speedup compared to the GNU version. The performance loss in XQueue as compared to standard LLVM is due to the significant load imbalance. On the other hand, GNU OpenMP incurs huge synchronization overheads for managing fine-grained tasks (about 60% on skylake-192) and the performance is significantly lower for GNU OpenMP compared to OpenMP with XQueue.

**Breadth First Search (BFS)** is a fundamental building block of many graph algorithms: it checks the connectivity of the graph from given source vertices, visiting one layer at a time. In order to demonstrate the applicability of XQueue using real-world datasets, we evaluate the BFS application from the GAP Benchmark Suite [25] using social network graphs such as Twitter and Friendster. The original implementation of BFS in the GAP benchmark leverages loop parallelism (LP) to parallelize every level of the tree. We modified the code to use recursive task-based (TP) parallelism with a base condition of 1024 nodes to evaluate XQueue. We also evaluate the extreme case with a base condition of 1 node, which creates several extremely fine-grained tasks. Each data point is the average speedup obtained by running BFS 64 times from pseudo-randomly selected non-zero degree source vertices. The Twitter graph has 61 million nodes and 1.47 trillion directed edges for a degree of 23 where degree is the maximum number of edges connecting a vertex. The Friendster graph has 65 million nodes and 3.61 trillion directed edges for a degree of 55.

Figure 6 shows the speedup achieved for both the test graphs on the skylake-192 using 192 threads. For the Friendster graph with a base case of 1024 nodes, GNU OpenMP scales well up to 24 threads and performance degrades at higher concurrency levels. XQueue performs reasonably well at full scale of 192 threads as compared to GNU and LLVM. XQueue achieves a speedup of $1.4 \times$ for Friendster and $3 \times$ for Twitter graphs over GNU with base case of 1024 nodes. Execution times for LLVM and XQueue are similar for Friendster and for Twitter, XQueue achieves $2.4 \times$ speedup. For the base case of 1 node, while there is no significant performance difference between LLVM and XQueue, GNU’s performance suffers significantly (up to 116× slower) due to the overhead of managing fine-grained tasks. Since real social network graphs are very unbalanced, they result in highly irregular memory accesses and load imbalance. Compared to the original GAP BFS using loop parallelism, XQueue achieves $1.9 \times$ speedup using Friendster and $1.6 \times$ speedup using Twitter with 192 threads, showing promise that the task-based parallel approach can be beneficial for these types of workloads.

**V. RELATED WORK**

XQueue is most closely related to work in concurrent queues and parallel runtime systems.

**Concurrent queues:** Several researchers have proposed concurrent queue implementations. Scogland et al. [27] presented the characterization of various concurrent queues on many-core architectures and proposed a high-throughput queue specifically engineered for many-core architectures. Schweizer et al. [28] performed detailed analysis of x86 atomic instructions on various architectures and discovered that atomics prevent instruction level parallelism and that latency depends on architectural properties such as the coherence state of the accessed cache lines. Scott et al. [29] proposed a lock-free queue algorithm for machines that provide atomic primitives. Cache-friendly concurrent lock-free queue (CFCLF) [7] is a lock-free queue that employs a matrix for the queue structure, reducing core-to-core communication overhead and making it cache efficient. BQ [30] is a lock-free queue that exploits batching to gain better performance. Morrison et al. [31] proposed a concurrent nonblocking linearizable FIFO queue using atomic FAA that outperforms CAS based implementations by up to $2 \times$.

**Parallel runtime systems:** Most parallel runtime systems and execution models, such as OpenMP [20], Charm++ [32], and Swift/T [3], use concurrent queues for sharing data between threads or processes. OpenMP’s task construct [33] enables task-based parallelism. Charm++ demonstrates about 10-20% improvement in performance by using optimization techniques like lock-free queues, CPU affinity, and memory
management [34]. Recently, Cpp-taskflow [6] emerged as an alternative to OpenMP task parallelism for C++.

To the best of our knowledge, we are the first to explore lock-less strategies in concurrent programming where data can be carefully manipulated to avoid the use of locks. Furthermore, existing runtime systems have not focused on the efficient support of fine-grained tasks, resulting in sub-optimal application execution, a problem that will only get worse with larger many-core architectures.

VI. CONCLUSION AND FUTURE WORK

XQueue is an extremely scalable lock-less MPMC out of order queueing system which can be used in tasking runtimes to overcome the performance limitations due to overhead of synchronization. Evaluation results show that XQueue is scalable up to hundreds of threads of execution with up to 6900× lower latencies and 3300× higher throughput when compared to naive implementations. We integrated XQueue with LLVM OpenMP and were able to achieve up to 6× speedup compared to native LLVM OpenMP and 1× to 4× speedup compared to GNU OpenMP in most cases with up to 116× speedup in some cases on applications from the BOTS benchmark suite and BFS application from the GAP benchmark suite.

In our previous work, we explored various lock-based work stealing approaches [35]. In the future we will investigate lock-less work stealing [36] as a scalable mechanism for dynamic load balancing with the aim to improve the current deterministic load balancing, broaden the applicability of XQueue, and achieve better performance on modern machines with hundreds of cores. We also plan to explore integration with GNU OpenMP [21], the Swift/T workflow system [3], as well as the Parsl parallel programming library [2] in order to further broaden the applications that could take advantage of the proposed techniques.

REFERENCES

Mechanisms for Transition from Monolithic to Distributed Architecture in Software Development Process

1st Aneta Poniszewska-Marańda
Institute of Information Technology
Lodz University of Technology
Lodz, Poland
aneta.poniszewska-maranda@p.lodz.pl, 0000-0001-7596-0813

2nd Joanna Macioch
Institute of Information Technology
Lodz University of Technology
Lodz, Poland
joasia.macioch@gmail.com

3rd Bożena Borowska
Institute of Information Technology
Lodz University of Technology
Lodz, Poland
bozena.borowska@p.lodz.pl, 0000-0001-7640-5782

4th Witold Marańda
Dept. Microelectronics & Computer Science
Lodz University of Technology
Lodz, Poland
witold.maranda@p.lodz.pl, 0000-0003-0153-7099

Abstract—Almost every enterprise functions thanks to IT systems in various ways – for example, in the context of process automation, cost reduction or increasing work efficiency. Appropriate construction of the architecture and adapting it to the conditions in which the system will operate is crucial for achieving the desired quality factors. This work focuses on the problem of transforming the existing systems operating in monolithic architecture into the architecture of distributed systems. The key element is the developed set of steps through which the transformation of the architecture in a given enterprise will increase the chances of the success of such an undertaking and help make important design and organisational decisions. The proposed plan was also tested on the example of an existing IT company.

Index Terms—Information systems architecture, microservices, monolith, distributed systems, software development

I. INTRODUCTION

Currently, architectural styles for IT systems oriented to the speed of software development are in demand. Emphasis is placed on the flexibility of the solution due to the desire to respond to customer needs in the shortest possible time. It applies to introducing new functions and the appropriate scaling of services to adapt the infrastructure to the increasing load on servers. Therefore, it is natural to look for new solutions that will better adapt the system to constantly changing functional requirements. Distributed architecture is one alternative in the available architectural information system patterns that should be considered. Many of the systems in operation today have a monolithic architecture. At some point, it turns out that this type of architecture slowly ceases to meet the needs of a given organisation. It is inflexible, which is not conducive to introducing changes frequently, and improving it through automation and the use of continuous integration techniques has its limits. Each of the existing information system architectures has its advantages and disadvantages. It is essential to understand the requirements and adapt the architecture to the conditions in which a given system operates.

When choosing an IT system architecture, keep in mind that it is often not related to the functional requirements. Usually, they can be met by systems in any available architecture. On the other hand, non-functional requirements have a more significant influence on its selection because they define the features that the system should have. These include requirements for availability, resilience, reliability, efficiency, and scalability. The currently existing implementations, business rules, and the desired technological stack are also crucial in architecture choice. A properly selected architecture allows for stable and sustainable development of the system.

The problem discussed in this paper is analysis and identification of the mechanisms of transition from a monolithic architecture to microservices architecture in the software development process. This requires examining the differences between given architectures and identifying the leading technical and organisational challenges related to architecture transformation. The work aims to develop a plan that companies can use to successfully change an IT system’s architecture from a monolith to microservices. The multi-step plan logically organises the process of architectural transformation and breaks it down into smaller parts. Thanks to this, it is easier to estimate the time needed to complete the following points of the architecture change plan. The transformation of architecture may take up to several years and depends on many factors, such as the specificity of a given IT system or the degree of enterprise development.

The remaining parts of this paper are organised as follows.
Section 2 presents the analysis and comparison of monolithic architecture and distributed systems architecture while Section 3 deals with the proof of concept of preparing a plan for the transformation of the monolith into microservices in order to increase the probability of success of such a transformation. The proof was carried out on the example of an existing IT company.

II. MONOLITHIC ARCHITECTURE AND DISTRIBUTED SYSTEMS ARCHITECTURE

The term monolithic architecture refers to an architecture where all system components run in one application and teams run on the same codebase and typically on a single database schema. In addition, a single deployment unit is produced for the implementation of the application. It accepts tasks from users and processes them in accordance with the implemented business logic. It connects to a single database to read or write the required data.

To add more structure to the code, the monolith is often implemented as an n-layer or modular. The two types differ in the way the code is organised. In the case of the n-tier model, the division is strictly technical – each layer has a different role (separation of concerns) to play in the system. For example, the presentation layer will be responsible for exposing interfaces to the world, and the persistence layer will be responsible for interacting with the database. The disadvantage of a layered architecture is that business domains are found in all layers and can become tight-coupling between components in a given layer. It is also easy to fall into the anti-bump, called the architecture sinkhole [1]. It consists of some layers that do not perform any business logic while processing specific tasks, thus creating many unnecessary objects in vain. This results in unnecessarily wasted memory and degraded performance. It may seem harmless at first, but as requests increase, the problem grows rapidly. However, when it comes to a modular monolith, the division is based on domains – this technique is inspired by Domain-Driven Design (DDD). This method focuses on establishing Bounded Contexts. They make it possible to maintain the uniqueness of terms occurring within one context and separate independent business concepts from each other [2].

One of the examples of distributed systems architecture are microservices, also known as microservices. There is no formal definition of microservices. Some argue that microservice architecture is not architecture but a variety of service-oriented architecture (SOA). There is some truth to this, as microservices are certainly derived from APS. They can be defined as small, cooperating, autonomous services organised around business capabilities and having their life cycles [3]. Users send jobs to the AP’s runtime gateway or submit them via a website. Then, each of the tasks is directed to the appropriate service responsible for implementing the given business logic. Each website has its database. The services communicate with each other and cooperate in carrying out all the necessary operations and return the response to the user.

As a rule, new systems should not be implemented in the microservices architecture [4], [6]. Initially, we cannot accurately predict how individual components will be related or how they will work together. Therefore, it is impossible to define the scopes of responsibility of individual services precisely and to be sure that as a result, the interfaces will be well defined and the services will not be closely related to each other, which should be avoided in both architectures. Additionally, the monolithic system is the easiest and fastest to implement. When a company wants to go to market for the first time, it is a wise choice, especially when funding is limited – maintaining infrastructure for distributed architecture requires more resources.

As a consequence of the wrong division of service responsibility, instead of microservices, one can obtain a distributed monolith in which all services must be implemented simultaneously with each change. The standard approach is to implement the system in a monolithic architecture but keep modularity in the design. It means that there is loose coupling between the components, and the whole is decomposed into modules.

The authors of [10] present a case study of an enterprise application that was developed and deployed in the cloud in both approaches – using a monolithic approach and a microservice architecture in the Play web framework. The results of performance tests executed on both applications are presented. Next, the authors describe the benefits and challenges that enterprises can get when implementing microservices in their applications.

The paper [11] describes the use of a conceptual decision framework that unifies modularity and distributed architecture. It is made on a five-stage systems architecture spectrum. The authors propose to add an extensive computational layer to the framework. Moreover, they explain how it can enhance decision making about the level of modularity of the proposed architecture.

The authors of [12] present the report of a real-world case study from the banking domain. It was made to demonstrate how scalability is affected by reimplementing a monolithic architecture into microservices. The described case study was based on the FX Core system.

The work of [13] presents the proposition of a new approach to transform the existing applications into microservices with the use of code repositories. The authors use the evolutionary and static code coupling information to automatically extract the microservices from monoliths.

Finally, the paper [14] presents state of the art in the field of software architecture and microservices, together with open problems and future challenges. The given survey addresses newcomers to the discipline and practical issues of potential solutions.

The transformation of architecture should be holistic and deal not only with the phases of software development but also the organisation of processes and teams in the enterprise:

1) Team Building – the first and often overlooked aspect when it comes to the transformation process itself is
preparing teams for it. Architecture is related to the software development process and project implementation methodology.

2) **Extraction and implementation of services** – the most challenging task when transforming a monolith into microservices is the proper isolation of functions according to the Domain-Driven Design approach. The service cannot perform only CRUD operations; it should perform some part of the business logic.

3) **Service Discovery** – to ensure successful communication between all services, you need a service discovery mechanism. Services run on containers or in virtualised environments, and their number and location can dynamically change.

4) **Communication between services** – communication between services can occur in two ways – through the client included in the service or using an additional infrastructure layer, called the services network.

5) **Testing** – as the number of microservices grows, system testing becomes more and more complex and time-consuming. In order to be sure that the new changes in the code did not negatively affect the rest of the functions, it is necessary not only to run tests but also to ensure an appropriate environment for their execution.

6) **Implementation** – the emphasis is placed on high automation of the entire process due to the fact that the goal of the microservices infrastructure is a highly developed ecosystem, consisting of many services that can be implemented even several hundred times a day by ensuring continuity of supplies.

7) **Scaling** – usually, the scaling of the application is done in accordance with the three-axis scaling model described in [6].

8) **Support and Maintenance** – many factors contribute to the continued support and maintenance of services – in the case of infrastructure, it is certainly essential to implement proper system monitoring.

From the site of the modular monolith, there are two paths for further architectural development. First paths – after a long time, the architecture will degenerate into a monolith, in which the boundaries between individual functions will be blurred, and the code will start to resemble a big ball of mud. This term was first used in 1997 by Brian Foote and Joseph Yoder [7], and it is used in the case of systems whose code is incomprehensible, and the structure is not uniform. Any changes to such systems are risky because it is very difficult to predict the full effects of the changes. Such a regression of the system takes place because, in general, even disciplined teams are unable to minimise the disadvantages of monolithic architecture that favour such degeneration.

Second paths – when the modular monolith architecture is no longer sufficient, it can be transformed into microservices. Often, however, organisations only begin transforming their architecture when their monolith is no longer modular, and the codebase has grown to a size and complexity that makes system development very difficult. There are many approaches to the problem of how to start breaking down the business logic of the system and separating it. It is presented in the next section.

III. PLAN FOR TRANSFORMATION OF A MONOLITH INTO MICROSERVICES

The proof of concept initially defined a holistic strategy to transform the system architecture from monolith to microservices. It consists of 7 steps that must be carried out to increase the probability of success of such a transformation. The plan was created on the basis of aggregated knowledge of how microservice architecture should look like, what it should characterise, and how to prepare organisations for such a significant change.

The system on which the proof of concept was carried out belongs to an international company that deals mainly with employee benefits and mobile payments. It offers a platform where employers can top up their employees’ virtual accounts with amounts for various types of benefits in accordance with the regulations in force in a given country. Employees, in turn, can use them to pay in stores for services and products belonging to the range of benefits they have received. The system has also been integrated with leading Chinese mobile payment providers. Chinese tourists can pay using a dedicated mobile application with funds deposited in their local accounts. Currently, the system, which works in a monolithic architecture, serves one million customers and can significantly increase this number in a short time.

Along with the increase in development teams and the popularity of the application, communication, technical and organisational problems began to emerge. It was decided to transform the architecture. After numerous talks, it was decided to change the monolithic architecture, in which the current system works on the architecture of microservices, in the belief that it will solve most of the problems faced by engineers.

A. Step 1 – System and Infrastructure Health State Assessment

As a first step in the plan, the condition of the infrastructure and the system itself should be assessed in terms of the problems both of these parts pose in everyday work. It is also worth gathering opinions on what programmers would like to improve in the existing system, what to get rid of, and the qualitative factors of the current architecture. By identifying the problems, it will be easier to understand the priorities when transforming the architecture. Before any decisions were made, it was necessary to take a critical look at the server part of the system and identify the problems that developers encounter in their daily work. After organising a meeting attended by team members maintaining the infrastructure and the server section, the following list of issues was established:

1) **Technology** – the system is written in an outdated version of the language, the support of which ended in 2014. It means that programmers often cannot use new libraries to improve their work and deal with bugs that have
been fixed in subsequent versions of the language. It is also difficult to integrate additional external tools when needed.

2) Implementation – the system is a monolithic application that cannot be called modular. The only breakdown of the code that can be seen is the technical breakdown into layers. The code is confusing and full of anomalies, from the ubiquitous tight bindings to bloated classes and hard-coded values. Due to the quality of implementation, understanding the code is difficult and time-consuming. Some of the lesser known parts of the code are reluctantly changed because it is not sure how any modifications will affect the existing code. Due to the fact that the market confirmed the product, the teams began to grow – working on one code base was slowly becoming troublesome. Besides, the entry threshold for new team members was high.

3) Deployment – deployments require a shutdown period, so they take place mainly at night when there is the slightest chance that a customer will want to use the system during this time. The system works on one server and does not have any backup, so in the event of a server failure, it is not possible to quickly redirect traffic to another operating instance. Moreover, the deployed delta is high.

4) Tests – the system has poor test coverage (using an external tool, the test coverage was 45%). Therefore, when implementing new functions, it is not 100% certain that the recent changes will not affect the older processes. Despite the low percentage of test coverage, it takes a long time to run a complete set of tests – it takes about 30 minutes. Their execution time is disproportionate to their number and the size of the codebase. Part of the blame lies with the old version of the language and technology platform. Development environments do not exist, and the test environment is misused as development environments. Due to the fact that there is only one test environment, teams often block when testing systems. Moreover, the production and test environments differ significantly, which has sometimes resulted in the implementation of bugs in the production environment.

5) Database – the database was poorly designed. There is data redundancy, and normalisation would be useful in several places. One of the worst problems was the type of ENUM used on a set that changes frequently. It is located in a transaction data table that contains many millions of records – as a result, each time the set is changed, the entire table needs to be updated, which is very time-consuming.

6) Monitoring – logs are reviewed in the raw state when a problem occurs. They reside on the production server, so each developer must have SSH login privileges to the server to be able to analyse them. Moreover, log rotation takes place every few hours, so you have to look for information in several different files in one day. There is no centralised system where you can track application bugs – usually, the bugs come from customer service. Neither are any tools used to track server and system health. The Nagios program is used in the basic scope, but there is no visualisation to aggregate and illustrate the collected information.

7) Availability and Performance – many times the system has stopped working during rush hour traffic on it due to too heavy a load. The fact that the production system runs on an obsolete disk, despite the fact that the organisation has access to newer hardware, does not help either. Additionally, the base turned out to be a real bottleneck when it comes to effectiveness. It is usually unable to process the number of queries sent to it at peak times in a satisfactory time.

8) Flexibility – no modularity and implementation method, old technology – system development is prolonged. Each change was time-consuming and required manual testing. It was impossible to deliver functions at the pace the business wanted.

9) Scalability – asymmetric scaling is impossible, which would be very useful for this system because one part of it is heavier than the other.

Having the result of the first step, we can move on to the second step – make a similar assessment, but this time on the state of the team itself.

B. Step 2 – Assessment of the State of Team Organization

The second step of the plan is mainly to look at how the team is working. Such an assessment should be an aggregate of two sources. The first one is an outside observer who has spent at least a few weeks observing what work methodologies the team uses, how it deals with them, and what works and what doesn’t. The second source should be the band itself and their opinions on the subject.

The server team consisted of 14 engineers, a technical support team of two, and the operations team. The team developing the mobile application consisted of 6 people. They all worked in the Kanban methodology.

Efforts were made to coordinate the tasks so that each team member worked on an entirely different topic. Still, due to the high integration of the monolith code, there were often situations where the changes introduced by two programmers overlapped each other. At that time, the problem of poor communication in the organisation was exposed because it sometimes happened that potential conflicts in the code were discovered only when trying to merge the code into the development branch, due to the lack of teams and common goals. There was also a problem with determining the scope of responsibility for the code in case of errors. Each person worked a bit on all functions, so the responsibility and knowledge about the overall operation of the process were very blurred. On the occasion of projects in which several people were involved to develop the code together, one could notice a lack of teamwork.
Having collected all the data regarding the imperfections of the system, infrastructure, and the organisation of teams, it is possible to go to the third step and answer one of the most critical questions.

C. Step 3 – What Does Microservices Help Us with?

The third step is one of the most important because its goal is to answer the question "How do microservices help the organisation?" The previous two steps made it possible to identify most of the problems that an organisation faces during the software development process. In the third step, it is necessary to objectively consider which of them will be solved by transforming the architecture into microservices and adopting agile methodologies for working with teams. It may turn out that none of these problems or only a small percentage of them will be eliminated in the long term – it means that there is probably no point in transforming architecture. It is worth considering whether the cost of transformation outweighs its benefits. Perhaps the real key to solving the problems faced by the organisation lies elsewhere. The answer to this question will determine whether the transition to step four will be necessary, or the adventure with the transformation of architecture will end at step number three.

For a company where proof of concept was conducted, the answer to this question indicated that the next step in the plan would be inevitable. First of all, it will be much easier to work with an ever-growing team in microservices architecture due to the lack of only one codebase. It will also be easier to control the technical debt associated with outdated technology and with highly coupled code from which functions will be incrementally cut. On the occasion of reimplementing, it will be possible to cover the code with tests and gradually divide the expanded database. The team would also like to scale the relevant services to properly distribute the traffic and avoid future problems with an unavailable system. These are just some of the examples the organisation gave when it agreed to begin the transformation. Therefore, she could go to step four, which is to prepare an action plan.

D. Step 4 – Preparation of Task Plan for Iteration

The fourth step is to prepare as constrained as possible a plan for the introduction of one or more services. In the beginning, it is best to focus on selecting the functions to be distinguished and defining the scope of responsibility of the new service and then choosing a migration strategy – according to the guidelines presented in the previous chapter. Later, it would help if you also thought about how the environment in which the service will operate should change and what additional tools it will need. This is where infrastructure issues should arise, such as introducing other servers, service discovery, log aggregation, and metrics collection. The result of this analysis is to be a plan of tasks to be carried out to re-implement the function as a separate service and implement it successfully. Each of these tasks should be prioritised to ensure that all the necessary requirements are met by going to the next point of the plan.

The plan is also to be limited, so encourage people to think only about necessary changes. With the first microservice, you don’t have to worry about introducing service meshes or about automatic service discovery – it’s best to choose simple and known methods and gradually automate processes as the number of services increases. In fact, the degree of complexity that will be imposed in the plan depends on the team and their skills.

Returning to the company that considered the transformation of architecture – it developed the following plan of change:

1) reorganisation of teams,
2) introduction of a centralised log aggregation system,
3) containing the application,
4) preparing additional servers,
5) implementation of selected microservices,
6) including microservices into the continuous integration pipeline,
7) implementation,
8) scaling.

The presented plan is a relatively high-level outline of tasks that must be undertaken sequentially to implement the system’s first services. In the sixth step, i.e. the transformation itself, the effectiveness of this plan will be checked, and each of its points will be described in detail. Before the transformation itself took place, however, there was one more step ahead of the organisation.

E. Step 5 – Are We Ready For a Change

In the fifth step, we can use the knowledge acquired from the previous step. Having a plan of changes that need to be carried out, it is possible to answer the question of whether the organisation is ready for the change. Is it able to accept the costs of such a change – not only in the context of the working time that needs to be devoted but also the costs of, for example, additional servers?

Secondly, it is necessary to determine what is the knowledge in the team about the architecture of microservices. The consequences of ill-defined confined contexts that could end up in a diffuse monolith should be assessed. Suppose your team is not very knowledgeable about this architecture or lacks expertise in the specific technologies required, or would help maintain microservices. In that case, it may be worth considering investing in training for the team.

In this organisation’s case, 70% of the team was already working in a microservice architecture, and 40% even participated in the transformation of monolithic architecture to a microservice architecture, which made things much more manageable. The programmers already had some knowledge and experiences that they could use. The cost of maintaining the infrastructure was also not a problem. There is nothing else to do but go to step six.

F. Step 6 – Transformation

The sixth step is carrying out the entire process of architecture transformation and starting the execution of the prepared
plan from the fourth step to implement the first service in the production environment.

At the very beginning, it was decided to change the team’s organisation and the methodology of its work. It was too early to create functional teams, and it was decided to leave the division into the operational, application server and mobile teams at the beginning. The server part was divided into two teams, whose members are to work on familiar topics, and the goals of the teams are domain-oriented. The work methodology has been changed from Kanban to agile. Two-week sprints have been introduced, and teams have been desynchronised to prevent overlapping recurring sprint planning, review and retrospective meetings. The introduction of the iterative software development method and dividing people into teams was received positively by them – the transparency of the entire process was increased. Finally, it was easy to find out what topics are being addressed by the teams and at what stage of implementation. Teams had tools to provide feedback on the course of their work, and it was easier for them to adapt to possible changes to the work plan, product adjustments or the way they work.

After two months of work in the new methodology, the teams got together so well that it was possible to start thinking about starting the monolith decomposition. The choice of the service or services that would have to be distinguished from the monolith was crucial. Due to the extensive experience of teams with microservices, it was decided to choose a bolder choice than recommended, as the selected functions were of high risk. Three services were separated – the first one allows external systems to make payments for their services using the user’s wallets in the application. The second is a QR code payment operator between stores and Chinese payment providers. The third is an API key management service that uses external systems to use specific functions. The third service is new, that is, it has not been separated from the monolith, but its creation was dictated by the need for a centralised system that would store access data and easily modify them for individual companies when needed.

Both services were chosen because of the potential for asymmetric scaling in the future (usually, payment services are the most heavily loaded) and because there is enough knowledge in teams to distinguish them from the end-to-end monolith. In this case, both services will still communicate with the monolith to book the transaction to the base. Its decomposition was not possible – the code is too tightly connected, and the change of models would result in huge problems related to their display invoicing companies and settling transactions. That is why a compromise was decided at the beginning. Responsibility for individual services was allocated according to the team that developed it. Due to the desire to propagate knowledge from separate domains to other teams so that in the event of crisis situations, they can also deal with the repair of faults, it was decided to allow changes in the code to members of teams who are not responsible for the service, provided that at least one member approves the code review of the team responsible for the service.

After selecting the functions, determining the migration strategy and responsibilities, it was left to deal with changes in the architecture, such as log aggregation, application containerisation and the introduction of additional monitoring in the first place.

It started with the introduction of a centralised log processing and management platform. They are used for this popular ELK stack, whose acronym consists of the first letters of tools such as Elasticsearch, Logstash and Kibana.

As shown in figure 1, the entire ELK stack resides on one server. The basic stack has been modified to include the Filebeat utility, which has been installed on every server running any log-creating applications. Filebeat is a tool that reads files from disk and passes them to Logstash. Logstash is a tool to which logs are sent in order to process them and save them to the database. Theoretically, the logs could be sent to Logstash immediately, but the team decided to add Filebeat due to one key advantage – in case the application server or the ELK itself temporarily did not respond for some reason, Filebeat will wait to send logs to ELK, remembering where it stopped send logs and what has already been sent. In the case of logging directly to Logstash in the event of a network failure, Logstash would start responding with errors and logs that he did not correctly receive will be lost as for later visualisations. Filebeat has the ability to restore their logs, and even if it is turned off for some reason for a week when it is turned on again, it will throw all logs not sent since the last time it was run to ELK. In short, with Filebeat, the impact of logging on the performance of the entire application is reduced. After the logs are processed, they are saved to the Elasticsearch database used by Kibana – data visualisation tool.

In figure 2, there is a screen from Kibana that teams can use to monitor the system quickly, track tasks and catch bugs in the repair application. Logging applications on four levels – INFO, WARN, ERROR and FATAL. Logs are also collected from development environments (created and configured by the operations team in the meantime, decomposing services from the monolith by the server team), the test and the production environment.

For each job, we can also see a trace.id – that is, a unique ID given to messages so that it is easy to trace how they flow.
through services in distributed systems. The NGINX server is responsible for assigning this ID. Additionally, log rotation has been changed from a few hours to a daily one to make it easier to view the raw logs on the server, if necessary. The introduced log management system solves the problem of collecting logs for future microservices, collecting them from several different servers and tracking related tasks.

The next step that had to be taken was the containerisation of services. In the context of microservices, it makes no sense to run the service directly on the hardware. Each service requires appropriate configuration, libraries and resources in an environment where hundreds of such services may be possible to dynamically run, move applications, or scale quickly. The whole process should be as automated as possible.

The first step was virtualisation, which consists of running the operating system on the host and hypervisor, controlling the work of virtual machines with separate operating systems and services running on them. While the advantage was the isolation of virtual machines and the possibility of running many of them on one host, the decrease in application performance was significant due to introducing an additional layer – the guest operating system and the working hypervisor. Equipment resource consumption has increased significantly. Containerisation is the next step after virtualisation, in which additional layers were removed.

The team decided to containerise all services, i.e. put them together with configurations and dependencies in separate containers. A container is a self-contained, standalone instance of the runtime environment. The tool the team used for containerisation is Docker. Docker images, unlike virtual machine images, run in the kernel of the host operating system. Thanks to this, resource consumption is reduced, and the application start-up time is very short. Docker is responsible for the complete life cycle of a container, including creation, management and destruction. Containerisation is an introduction to orchestration for the company. When overtime, the number of containers will be counted in tens, it will be necessary to introduce a platform for their orchestration, i.e. for automation, management and scaling of containers. These platforms also offer load balancing tools, and much more. One of the most popular orchestration platforms is Kubernetes. Containerisation will also allow easy transfer of services to the cloud, although it is still a distant future for the organisation. For both the new services and the monolith, three Docker configuration files with the YAML extension were created – one for continuous integration, application launch, and database setup.

Having log aggregation and a template to containerise services, it was possible to start implementing pre-planned services and slowly start to decompose the monolith. The implementation itself did not cause any problems – we just had to remember to add the project to the company’s continuous integration tool, i.e. Jenkins. Additional servers have been specially prepared by the operations team for the implementation and testing of new services. However, as the number of servers increases, so does the number of potential problems. Thus, it was decided to extend the monitoring tools. Zabbix tools were used to monitor network parameters and server operation. Of course, the data can be much more – we can also monitor the response times of individual services, the consumption of CPU and memory resources [8]. It is up to the team to select the critical data to monitor.

On the occasion of the introduction of services, concerns were raised about the escalation of the problem of inefficient database and the lack of a replica of the monolith, which still partially depends on new services. In the event of a monolith failure, the services will always respond but will not be able to process the transaction successfully. To prevent this, it was decided to scale the application and replicate the database. In order to distribute traffic to more than one server, a highly available load balancing tool had to be provided. The choice fell on HAProxy with the Keepalived mechanism. To avoid a single point of failure, three HAProxy instances were introduced, and the Keepalived mechanism will allow switching between nodes depending on their health.

If the slave server discovers that the master is not responding, it assigns a virtual IP to its server, taking over all traffic. The system does not yet use service discovery mechanisms because there are too few to add extra infrastructure elements. First, all service locations are stored in the configurations of all three HAProxy instances. It is not the optimal solution but the simplest. Traffic is only routed to running application instances thanks to the cyclic push mechanism configured in the NGINX tool on each server. The traffic between the running instances is assigned according to the lowest number of connections. In the case of equal numbers, the distribution on the servers is carried out by the carousel algorithm.

The Galera replication protocol was used for its replication instead of the standard master-slave model for the database. All nodes accept queries in the cluster. Moreover, cluster data is updated synchronously, and each operation returns a result only after all nodes have performed it. The database is also highly available, and it is easier to scale. According to the rule, there are three nodes defined in the cluster that should always be an odd number to ensure quorum and avoid the split-brain problem. This problem occurs when nodes, due to a network failure, cannot communicate with each other, and as a result, the cluster is split up, and each of them believes
that it is the only active cluster. In such a situation, clusters may try to modify the same data simultaneously, leading to their corruption.

Performance tests were carried out using the Gatling tool to check the cluster’s impact on application performance. Performance tests of the three most popular operations were carried out – searching for available payment objects, making one of the payment transaction types and calculating the balances and credits of the user’s available wallets.

Figure 3 shows the results of first performance test. Thirty queries were sent every second. The minimum response time for the Galery cluster was 53% longer than for the local database and was 772 milliseconds. The maximum response time for the local base was 3983 milliseconds and was about 25% faster than the cluster. Half of the query response times gave the most significant difference of up to 50% for the local database and cluster. In the overall picture, 99% of the Galery cluster queries returned after 4625 milliseconds and for the local database after 3559 milliseconds. The difference between the response time of the bases for 99% of inquiries is 23%.

G. Step 7 – Repeating the Iteration

The seventh step of the plan should take place after the service is deployed. It encourages you to repeat the first three steps to evaluate changes in your system. If you want to introduce more services, it is also recommended to repeat the entire cycle. Of course, some steps will no longer be needed in subsequent iterations – such as steps three and five. Indeed, step 4 will contain a much more limited plan than when implementing the first microservices. This is because some of the infrastructure supporting microservices will already be present. Nevertheless, there are always other processes worth automating.

The organisation resolved the problems identified in step number three. Services meet essential design criteria such as organising services around business capability, autonomy, implementation independence, fault isolation. The facilitation of controlling the level of technical debt turned out to be of particular value.

IV. CONCLUSIONS

Monolithic architecture faces many problems as the codebase, and the number of teams working on it grows. The software development cycle is getting longer and more expensive. As a result, enterprises are inadequate to adapt to and respond to customer needs. The architecture of microservices addresses some of the problems related to software development in monoliths but also has several technical challenges that must be solved. The proof of concept showed that the proper preparation of the enterprise and teams for the transformation of architecture contributed to the structuring of this change’s entire process. No unexpected or previously unforeseen problems arose.

Decomposing a tightly coupled database turned out to be too difficult at the moment, but the key to success is to gradually unravel the contexts constrained in the code and carry out the migration responsibly according to the knowledge of the teams. The work should not only focus on the continuous isolation of the monolith but also the parallel restoration of modularity in the existing monolith. The transformation of the architecture can take several years, depending on the size and complexity of the systems. The experience and size of the teams also have an impact, as is the available time given by the company to work around the architectural change. Nevertheless, it is worth gaining the appropriate experience and ahead in automating processes to ensure a continuous delivery model and increase the company’s competitiveness in the market.

REFERENCES

Precomputed Ionospheric Propagation for HF Wireless Sensor Transmission Scheduling

Terry Koziniec, David Murray, Michael Dixon

Abstract—Global communications without reliance on an engineered communications network make the ionosphere an attractive medium for wireless sensors in remote deployments. However, ionospheric circuits' temporary availability is a challenge in scheduling transmissions for a sensor with limited power, communications and computational capacity, particularly where cost and antenna constraints limit operation to a single frequency. We describe a technique for scheduling transmissions based on precomputed propagation models. The models predict the time-varying Signal to Noise Ratio (SNR) at the receiver. We describe methods to determine threshold SNR values, using the Weak Signal Propagation Reporter (WSPR) database to determine if a time slot is suitable for transmission.

Two techniques are investigated to quantify the failed receptions: the Inverse Square Law method uses a statistical approach and a sampling measurement technique called Goldilocks. The two approaches yielded threshold SNR values of -21 dB and -19 dB, respectively, for a time slot with a 90% successful reception goal. Applying these thresholds to the modelled SNR, we generate a precomputed hourly transmission schedule. With the schedule determined monthly, a 12-month plan requires 36 bytes of wireless sensor storage. A six-day experiment, using a 1677 km path, found that the schedule resulted in an 83% reception rate when used with a power level of 200 mW.

Index Terms—HF, ionosphere, IoT, propagation, wireless sensor

I. INTRODUCTION

High Frequency (HF) radio waves are unique in their ability to offer infrastructure-free, non-line-of-sight communications by taking advantage of the ionosphere to reflect signals enabling over-the-horizon signal paths. A single hop allows communications to around 4,000 km, while subsequent reflections from the ground and back to the ionosphere can enable global communications under appropriate conditions. The challenge of using HF is that the ionosphere is a dynamic environment consisting of electrons and parent ions driven by solar electromagnetic events. HF wireless circuits are transient, and their availability is dictated by time of day, season, latitude, long and short term solar events.

Historically, Internet-based end-to-end data communication has required either a reliable physical layer or a data-link layer that presents a seemingly reliable frame delivery service. Examples of such mechanisms include the Media Access Control (MAC) of 802.3 (Ethernet) and the acknowledgement based re-transmissions of 802.11 (WiFi). To a large degree, this is driven by the technical requirements of Transmission Control Protocol (TCP) and its poor performance in lossy environments [1].

This work was supported by an Australian Government Research Training Program Scholarship.

Fig. 1. MQTT Framework enables IoT devices to connect to the Internet without the need for a TCP/IP stack implementation over the wireless channel.

While HF is routinely employed for data communications, particularly by the military [2], such systems are bulky, power-hungry and often require human intervention to ensure the selection of ionospheric circuits necessary to support Bit Error Rates (BER) in line with the requirements of TCP.

The ubiquity of IP, and the reliance on TCP, is being challenged with longed-lived battery-powered wireless Internet of Things (IoT) sensors. The small payloads associated with many IoT sensing applications and the large addressing and header overheads of TCP/IP are inefficient in terms of energy usage and transmission times. Wireless IoT devices frequently break from the traditional IP-to-the-sensor model to isolate themselves from the processing and airtime overheads of a full TCP/IP stack on an energy-constrained wireless sensor. This has led to the adoption of application-based communication protocols such as ISO standard ISO/IEC 20922:2016 Message Queuing Telemetry Transport (MQTT) 1.

Frameworks such as MQTT (Fig. 1) provide the benefits of Application Program Interface (API) standards while also allowing flexibility in the implementation of the wireless link layers. MQTT achieves this flexibility through the use of a wireless gateway that bridges the lightweight wireless components to a TCP/IP end-point. MQTT handles the movement of sensor data to the Internet but places no constraints or assumptions on the wireless link. This is particularly helpful for HF wireless links that behave very differently to a point-to-point or LAN based wireless technology. When considering the use of HF as a transmission medium, independence from the constraints of TCP can considerably simplify the design of the sensing node and allows for the acceptance of any limitations within the wireless portion the system. We argue that these frameworks provide broad standards-based approaches that can enable the development and integration of High Frequency (HF) wireless for niche sensing applications.

This paper considers the potential for maximising transmis-
sion success, despite the intermittent nature of ionospheric circuits, through transmission scheduling based on precomputed ionospheric propagation predictions. Implementation involves loading an HF-based sensor with a transmission schedule based on ionospheric predictions for the sensor’s lifetime. We generate ionospheric predictions using two leading systems used in the Northern and Southern hemispheres. Namely, the Voice of America Coverage Analysis Program (VOACAP) [3], and the Advanced Stand Alone Prediction System (ASAPs) [4]. We derive this relationship between SNR and transmission success from an extensive database of HF propagation data using two alternate methods to estimate the number of unobserved lost frames. Finally, the effectiveness of the precomputed transmission schedule is evaluated against preliminary experimental observations. The authors are unaware of previous work proposing a lightweight transmission scheduling scheme for wireless sensors using the HF band. This paper establishes a starting point for the development of such schemes. We assume that the sensor has no mechanism for measuring or updating its knowledge of ionospheric conditions after deployment. The results of our research are both a specific evaluation of the estimation techniques and a more general validation of the viability of precomputation as a mechanism for transmission scheduling.

II. BACKGROUND AND RELATED WORK

A. High Frequency Band Benefits

A key factor affecting the range of wireless devices is whether the sender and receiver have a low-loss line-of-site to one another. Obstacles, undulating terrain and the earth’s curvature are limiting factors. At frequencies higher than HF, even with optimal elevated sites, it is rare to achieve wireless connections beyond 100km, and at this range, the link would be “engineered” with directional and aligned antennas. For poorly situated sensors in a valley or separated by mountains, even short-distance communications may be impossible.

HF propagation occurs via a reflection off the ionosphere, with single skip distances of up to 4,000 km and global coverage being possible through complex propagation modes via layers of the ionosphere and ground reflections. HF has a range potential not afforded by other point-to-point technologies. Compared with higher frequency technologies, HF solutions are less likely to feature strongly directional antennas due to their cost and impractical dimensions. However, a lack of directionality allows multiple receiving stations to record sensor data directly from the source. Overhearing of transmissions may suit the communications architecture of the sensor network or improve its reliability through redundancy [5].

For truly remote locations or those with challenging line of sight issues, geosynchronous and Low Earth Orbit (LEO) solutions are proven and evolving solutions. However, their disadvantages include more costly radio hardware and the ongoing costs of maintaining a satellite service plan with a provider. While satellite services are generally reliable, there are applications where dependence on a third party or intermediate infrastructure is undesirable.

Existing research that attempts to map SNR measurements to digital HF transmission success [6] [7] [8] have used MIL-Std-188 [2] or similar radios with little in common with the IoT sensing devices our research targets. The relationship between SNR and BER is a function of the coding and protocol employed. Consequently, SNR thresholds do not translate between systems or protocols. Our research specifically examines this relationship under using an appropriate coding scheme.

B. High Frequency Band Challenges

While there are compelling reasons to choose the HF band, there are also significant obstacles to widespread adoption. The first is the limited bandwidth. The entire HF band comprises frequencies in the 3-30 MHz range, a width not much wider than a typical WiFi channel. The long wavelengths associated with the HF band make electrically efficient, resonant antennas unwieldy and add to deployment cost and complexity. This challenge increases where there is a requirement to operate across significantly different frequencies.

Although the HF band is generally a reliable medium for communications, it requires an experienced operator, in conjunction with ionospheric predictions and charts [4] [9], or an automated system to select an appropriate frequency for a particular radio circuit. An example of such a system is the Automatic Link Establishment system defined by MIL-STD-188-141B [2]. Such mechanisms either require an out-of-band medium for the delivery of ionospheric predictions, or they make use of synchronised exchanges between nodes to “probe” the ionosphere. Neither of these approaches is compatible with the notion of a low powered, autonomous HF-based wireless sensor node.

These limitations encourage using a single band HF radio and optimising the sensor platform around a fixed frequency. The hardware is simpler, cheaper and likely to see more battery power represented as radio frequency energy.

C. Assumptions Regarding HF Wireless Sensor Nodes and Applications.

Given the constraints discussed and the wide range of applications associated with wireless sensors, it is essential to understand the authors’ perspective regarding deploying and designing this niche technology. Typical applications and deployment scenarios are envisaged as having the following characteristics:

1) Remote locations without existing commercial communications infrastructure.
2) Deployment density is too sparse for a cell or mesh.
3) Best-effort data transmission is acceptable.
4) Sensor is treated as “disposable”.

Although this study assumes the unidirectional and unacknowledged transmission of data from the wireless sensor to one or more aggregating receivers, we intend that this work serves to develop a robust physical and data-link (media-access) layer approach that will underpin the development of a bidirectional acknowledged protocol for applications that require more certainty.
III. MODELLING AND PREDICTION OF THE IONOSPHERE

The non-trivial compromise that results from a decision to use a fixed frequency HF-based sensor is that we can no longer change frequencies and select from available ionospheric circuits. Rather we must transmit when one becomes available. With prior knowledge of where the node is deployed and the locations of one or more receiving stations, we can precompute propagation tables. Precomputation trades off a small amount of storage on the node for considerable CPU cycle and power savings in the field.

The ionosphere is a well-studied feature of our planet, and while it involves the interactions of complex systems, many aspects of its behaviour are cyclical, well understood and predictable. By far, the most significant source of energy to strip electrons from their parent atoms is our Sun. Day vs night, latitude and the seasons are all determinants of the ionisation level at a given point in space and time. The Sun itself rotates on a 27-day cycle, and from time to time, sunspots or solar flares develop, emitting high energy X-rays that quickly (≈8 minutes) enhance the ionisation level. On occasion, Coronal Mass Ejections (CME) result in particles directed at the Earth arriving, on average, 3.5 days later leading to periods of poor HF signal propagation [10]. Although these solar phenomena follow an approximate 11-year cycle of maxima and minima, individual event timing can not be predicted far into the future. Over shorter time-frames of days or weeks, astronomers can optically observe the Sun for evidence of slowly developing sunspots and these are used to estimate ongoing X-ray emissions. These estimates are then used to modify the predicted behaviour of the ionosphere in the near term. The longer-term cyclical trends are modelled using one of several commonly used ionospheric predictive systems. Many of these share standard physics engines, which have developed in parallel with advances in scientific knowledge surrounding the ionosphere.

This study utilised two modelling packages to predict propagation: Voice of America Coverage Analysis Program (VOACAP) and Advanced Stand Alone Prediction System (ASAPS) [4].

A. VOACAP

VOACAP traces its roots back to US Army research published in 1948 [11] and was embodied in FORTRAN code in 1968 as the Ionospheric Communications Analysis and Prediction Program (IONCAP). It is based on an electron density model [12]. Although the code is now over 40 years old, IONCAP is still the computational basis for many modern ionospheric propagation and modelling packages. Today VOACAP is maintained as an open-source project [3] and continues to be one of the most frequently used HF propagation prediction and modelling tools.

VOACAP uses the Smoothed Sunspot Number (SSN) as the metric for solar activity. The SSN is determined by considering six-monthly past observations and six months of predictions using the Lincoln-McNish [13] least-squares approach to fit a curve. These curves are then extrapolated for longer term predictions.

<table>
<thead>
<tr>
<th>TABLE I</th>
<th>MODELLING SYSTEM PARAMETERS (VOACAP)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Man-made noise level at 3 MHz:</td>
<td>145 (residential)</td>
</tr>
<tr>
<td>(dBW/Hz in 1 Hz bandwidth)</td>
<td></td>
</tr>
<tr>
<td>Minimum takeoff angle of main lobe (deg):</td>
<td>3</td>
</tr>
<tr>
<td>Required circuit reliability:</td>
<td>90%</td>
</tr>
<tr>
<td>Required S/N ratio (dB):</td>
<td>3</td>
</tr>
<tr>
<td>Multipath Power tolerance (dB):</td>
<td>3</td>
</tr>
<tr>
<td>Maximum tolerable time delay (ms):</td>
<td>0.1</td>
</tr>
<tr>
<td>Absorption model:</td>
<td>Normal</td>
</tr>
<tr>
<td>Transmission Power (W):</td>
<td>0.2</td>
</tr>
<tr>
<td>Antenna model (TX/RX):</td>
<td>13 (f4m2020a.13)</td>
</tr>
</tbody>
</table>

B. ASAPS

ASAPS is a commercial application developed by the Space Weather Services of the Australian Bureau of Meteorology. Initially developed as a DOS-based application in 1988, it has been continually developed and is in active service with military, commercial aviation and emergency services users.

ASAPS uses T indices [14] as the metric for solar activity. The T indice is derived from 30 years of measurements taken of the ionosphere at various locations during high (T = 100) and low (T = 0) periods of solar activity. These measurements are mapped on an hourly and monthly basis for 12 months, along with the observed SSN. This yields 576 ionospheric maps representing the hourly ionospheric conditions for a given year [15]. These maps form a lookup table that determines the reflection height for a given set of conditions [12].

C. Ionospheric Modelling Parameters

Table I outlines the system parameters used in the modelling. Although the parameters refer to VOACAP, ASAPS uses equivalent settings. The modelled antenna was a type 13, an explicitly defined three dimensional radiation pattern, rather than a predefined VOACAP model. This provides consistency across both ASAPS and VOACAP. The particular model used was f4m2020a.13 [16], corresponding to an elevated dipole with four radials over average ground and is a reasonable approximation of the actual transmit and receive antenna used in our experiments.

Although the transmitter power is 200 mW, ASAPS has a minimum configurable power level of 1 Watt. Consequently, the ASAPS model was set to 1 Watt and the resultant SNR tables were adjusted by $-7 \text{dB}$.

1) $\text{REQ.SNR}$: $\text{REQ.SNR}$ is a measure of the minimum SNR that the protocol requires for a successful decode. This parameter can be derived from the published WSPR sensitivity. Taylor [17] suggests a value of -31 dB in a reference bandwidth of 2500 Hz. From this sensitivity value, a required SNR of 3 dB can be calculated.

$$ \text{REQ.SNR} = \text{SNR[dB]} + 10 \log(\text{BW})[\text{Hz}] $$

$\text{REQ.SNR} = -31 + 10 \log(2500)$

$\text{REQ.SNR} = 3 \text{dB}$

2) SSN and T-indices: For our application, we are concerned with ionospheric predictions embedded in the wireless sensor at the time of manufacture and their effectiveness
TABLE II

<table>
<thead>
<tr>
<th>WSPR PROTOCOL SPECIFICATIONS [19]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message length (bits)</td>
</tr>
<tr>
<td>Forward error correction</td>
</tr>
<tr>
<td>Channel symbols</td>
</tr>
<tr>
<td>Modulation</td>
</tr>
<tr>
<td>Keying rate</td>
</tr>
<tr>
<td>Transmission length</td>
</tr>
<tr>
<td>Occupied bandwidth</td>
</tr>
<tr>
<td>SNR sensitivity for decode</td>
</tr>
</tbody>
</table>

at some point after deployment. For this reason we have considered solar indices available 12 months prior to our 2017 experiments.

IV. FRAME CODING - WEAK SIGNAL PROPAGATION REPORTER (WSPR)

All transmission experiments and analysis use WSPR encoded frames. WSPR is an Amateur Radio protocol designed to test propagation paths using the HF and lower bands. Characteristically the protocol is low speed, requiring very little bandwidth and features strong forward error correction. Typically it is used at low power levels, with most transmissions being 5W or less, and many dedicated WSPR transmitters operate in the mW range. While the frame structure of WSPR is optimised for its application, the encoding and physical layer characteristics of the protocol would also be suitable for wireless sensor node telemetry.

WSPR encodes the sender’s call sign, a power level and approximate geographic location known as a maidenhead locator [18]. Receivers of the signal upload the data, known as a “spot”, to an online, public database referred to as WSPRnet (http://wsprrnet.org). WSPRnet stores the spot and metadata, including the receiver’s measured signal strength, frequency, time, and maidenhead location. The data is stored indefinitely, and both WSPRnet and third parties provide visualisation tools to enable real-time assessments of propagation conditions. Table II outlines the technical specifications of the protocol.

Our basis for selecting transmission times for an HF-based wireless sensor is adequate SNR. However, the relationship between BER (Bit Error Rate) and SNR is moderated by multipath fading and frequency spreading from the Doppler effects of the moving ionosphere. This is most pronounced at sunrise and sunset when ions form and subsequently recombine rapidly. Once multipath and Doppler spread exceed the capacity of the encoding scheme, even substantial improvement in SNR will not make a circuit available [7]. This paper does not explicitly consider multipath and Doppler spread. In part, these effects already influence the predictions made by the prediction software, but more importantly, WSPR is designed to mitigate these effects. The tone separation of 1.46 Hz is designed to accommodate typical HF Doppler shifts, and the lengthy symbol duration, interleaving and FEC provide high immunity to Doppler fading and short duration noise events.

A. Derivation of SNR-Based Probability of Reception Success

Using SNR tables generated from a modelling package such as VOACAP or ASAPS to determine suitable frame transmission times, makes it necessary to equate a given SNR with the probability that the receiver can successfully decode the frame. We derive SNR based probabilities using pre-existing observations in the WSPRnet database. However, the database is limited as it records only reception reports from receivers and misses frames transmitted but never received. The lack of sender-side logging is problematic as we need to divide the number of frames received by the number of frames sent. We describe two different approaches to address WSPRnet’s missing sent frame data.

1) Inverse Square Law Method to Quantify the Number of Unheard Frames:

The principle behind this approach is to consider that signals decrease in strength as they spread over a larger surface area. As a result, we expect to see proportionally more weak signals than strong signals. For a reception sphere area given by \( A = 4\pi r^2 \) we expect a histogram of random SNR observations to follow this curve. With decreasing SNR being observed more frequently as the radius from the source increases. To the extent that the empirical observations are less than predicted, we assume a frame is present, but incapable of being decoded. For any SNR, we divide the number of observations in the WSPRnet database by the number expected based on the inverse square law to determine a probability of success.

Using \( n = 500,000 \) samples of WSPR spots for the 14 MHz band as reported in August 2020, the histogram is illustrated in Fig. 2. As expected, the number of observations grows as the SNR (reflecting signal strength) decreases due to the inverse square law. After which, it progressively declines due to the higher proportions of undecodable signals.

We fitted a quadratic equation to the higher SNR portion of the curve, where we assume relatively few frames will be lost. The portion of the histogram used to fit the quadratic equation was -16 dB through to 0 dB. The equation of the curve is given by:

\[
38x^2 - 518x + 3275
\]

The curve in Fig. 2 is a close fit with an \( R^2 \) value of 0.9998 with no pattern to the residuals. This evidence supports the mathematical underpinnings of the observed histogram in this region. The curve was then extrapolated to estimate the total number of frames present. Finally, points on the curve are located where 90% of frames were received and where 70% are received. 70% was selected as a criteria as the probability of reception of at least one frame is received when two frames are sent with a probability of 0.7 is: \( 1 - (1 - 0.7)^2 \) or 91%. This provides a wireless sensor node with an option to send a frame at an optimal time with a 90% or better chance of success or achieve the same result with a double transmission at the 70% threshold. Using this technique, the thresholds for 90% and 70% were -25 dB and -21 dB, respectively.

A source of concern with the histogram is the apparent “slump” from -17 dB through to around -21 dB. One theory is that not all waves radiate in all directions forever. For waves
that radiate parallel to the ground, a receiver may receive this strong signal directly via line of sight over a relatively short distance, but weak signals never make it beyond the line of sight. Thus signals are over-represented above -17 dB and underrepresented below -17 dB. Another explanation for the “slump” could be that a certain range of the decreasing SNR levels are seen at altitude as the signal passes to the ionosphere and back. So while a particular SNR level does occur, if no observer is present to record it, these values will be understated.

2) Goldilocks Method: To address the limitations of the inverse square law technique, we propose a contrasting approach that focuses, not on the population as a whole, but on individual frame reception events. Considering transmission events in isolation removes dependence on the normality of the data. We refer to this method as the “Goldilocks” technique for determining the presence of undecoded frames. The name reflects the strict conditions necessary for a transmission to qualify for sampling.

For each frame arriving at the receiver, we record the SNR and record a “success” for a decoded frame and a “failure” if it is not. The probability of success at a particular SNR is the sum of the successes divided by the total frames seen by the receiver at the same SNR. The complicating factor is that, by definition, we do not know anything about frames that the receiver did not successfully decode. The WSPRnet database contains only spots (receipts). It does not include records of sender activities, so a frame not decoded may never have been sent.

Goldilocks starts with the problem of identifying the SNR for a frame that was not seen (Fig. 3). As we cannot measure an unseen frame’s SNR, Goldilocks searches the database for periods of stable SNR for a given receiver and transmitter pair. This is a period where a receiver reports an identical SNR for a frame received in period 1 (P1) and the same SNR for a frame received from the same sender in period 3 (P3). Goldilocks assumes that the SNR in the intervening period, P2, would have had the same SNR. This assumption is an acknowledged source of error. However, in practice, dramatic shifts in SNR are mainly limited to sunrise and sunset when the SNR is steadily increasing or decreasing, and these are not times that the Goldilocks criteria for stability will be met.

Two further conditions must be satisfied before Goldilocks can consider P2 as an actual measurement period. Firstly, there must be confirmation that the sender observed in P1 and P3 is also transmitted in P2. Goldilocks looks for third party
recipients of the frame in P2 to confirm that this is the case. If we cannot prove transmission in P2, we cannot distinguish between lost frames and those not sent. The second condition that must be satisfied is that the receiver could receive the frame in P2. If both tests are satisfied, then there is evidence the transmitter sent the frame, and the receiver was listening. In this case, the non-receipt of a frame is a deemed loss at that SNR value. Similarly, any recorded frame is a success for that SNR. If Goldilocks can not confirm a frame was sent or that the receiver was listening in P2, the test is rejected.

Using the combined 2016/17 WSPRNet 14 MHz database of 536 million records, 271,998 met the Goldilocks P2 conditions for sampling. The results are shown in Fig. 2. We sought thresholds representing probabilities of success of 90% and 70%. These thresholds were met at approximately -19 dB and -29 dB, respectively. The 90% thresholds for the two techniques are within 2 dB of each other in a region of the loss curve where minor variations in SNR tend not to lead to significant differences in loss rates. Such a similar outcome from vastly different techniques suggest that the two tools may be used to cross-validate threshold estimates. At the 70% threshold, while the 4 dB difference is not great, it is more significant, both in magnitude and due to the increased sensitivity to SNR as we approach WSPRS’ SNR sensitivity of -31 dB (in a 2500 Hz bandwidth).

B. Experimental Assessment of Precomputed Ionospheric Propagation

Data from an experiment conducted in January 2017 was used to simulate a sensor node’s deployment and compare predicted, scheduled frame transmission success against observations in the field. A Perth, Western Australia (-32.0664S, 115.8367E) based receiver was placed 1694 km from a transmitter placed near Ceduna, South Australia (-32.066458S, 115.8367E). The experimental configuration is outlined in Table III.

V. RESULTS

A. VOACAP and ASAPS predictions

Fig. 4 shows the daily SNR predicted by ASAPS and VOACAP for frames transmitted at Ceduna and received at Perth.

ASAPS is quick to categorise a marginal circuit as unavailable. The transition is so sudden that there is no difference in the parts of the curve captured by the -19 and -29 dB thresholds. We attribute ASAPS behaviour to its basis in historical observations of actual propagation. Practical limits in the resolving power of the ionosondes used to take the measurements mean that below a certain level, signals are simply undetectable.

In contrast, Fig. 4 shows VOACAP’s predictions to be more granular, with a period between 1230 and 1415 UTC where the predicted SNR is below -19 dB but at, or above, -29 dB. This marginal signal area allows consideration of the merits in defining a 70% threshold.

In practical terms, the predictions made by VOACAP and ASAPS are very similar. At the 90% threshold, both tools predict the band closure will be at 1230 UTC. ASAPS predicts the band opening at 2230 UTC, and VOACAP expects it will open at 2330 UTC. It is essential to consider that the thresholds are approximate only; the graph’s vertical placement is a function of the assumed noise levels at the receiver and the...
transmitter’s ground characteristics. Both of these values are specified only in broad terms. A mere 2 dB shift in signal or noise levels can bring VOACAP and ASAPS into perfect alignment or spread them apart.

B. Experimental transmissions

Of the 1485 transmissions that were generated, 768 were successfully decoded at the receiver. A success rate of approximately 53% forms the baseline for a node without knowledge of the ionospheric conditions. Non received frames were tagged with a -99 dB SNR in a local database.

VI. Analysis

Fig 5 shows the observed SNR plot against the ASAPS and VOACAP predicted values. Visually there is broad agreement between the experimental observations and the predictions. Successfully transmitted frames are generally at those times when ASAPS and VOACAP SNR exceed the -31 dB SNR decoding limit of WSPR frames. Non received frames can be seen banded at the bottom (-99 dB) of the graph clustered around those times that predictions suggest no circuit exits.

However, the trailing edge of the prediction is not as consistently or accurately reflected in the observations. On all but the first evening, the circuits progressively faded or ended earlier than predicted. Given that the predicted SSN and T-index was lower than observed during the experimental year, and there was some modest sunspot activity, it would be reasonable to have expected circuits to remain open longer than predicted rather than closing early. This observation, along with the already noted differences between the models, warrants further investigation.

Table IV summarises VOACAP and ASAPS’ success in predicting the observations at the various thresholds suggested by the inverse square law and Goldilocks curves. Success is the ratio of packets successfully decoded divided by the number of packets that would be sent at a given prediction threshold. Both the -19 dB and -21 dB thresholds (targeting 90%) yielded a success rate of 0.82 when applied using either the VOACAP or the ASAPS predictions. The statistic Agreement (overall agreement) also reflects how accurately non-transmission times are identified. A predictor with a higher Agreement is likely to yield more transmission periods and is a true measure of the tool’s discriminatory power. Using this measure, the -21 dB threshold of the inverse square law technique combined with the VOACAP prediction yields an overall agreement of 0.86, as does the ASAPS prediction.

At the thresholds targeting 70%, the important parameter is the success of frames sent when the predicted SNR lies between the 90% threshold and the 70% threshold. In the case of the inverse square law model, lowering the threshold to -25 dB (70%) from -21 dB (90%) sees 70 more frames transmitted. Of these, 15 are successfully received for an effective rate of 21%. Using Goldilocks’ thresholds, an additional 330 frames are transmitted, with 63 successfully received. A success rate of 23% is below the 70-90% we were expecting.

Delving deeper and using the Goldilocks 70% threshold (-29 dB), VOACAP considers the band to be open an hour earlier at 2230, allowing an additional 60 time-slots over the experiment’s six days. Of these, 76.7% proved viable, consistent with our 70% target. However, the threshold change also sees the band close 2 hours later, and of the additional 140 time-slots made available, just 20.7% proved viable.

VII. Conclusion

Overall our research has shown that a simple, lightweight transmission schedule can significantly improve the likelihood of transmission success for an HF-based wireless sensor node. By implementing a precomputed schedule the likelihood of success increased to between 82%, and 83%, using a transmission schedule predicted 12 months before deployment. This improvement was achieved without an excessively conservative approach to choosing viable time slots. Up to 58% of available time slots were considered viable, with the ionospheric models both able to correctly categorise viable and non-viable time-slots with up to 86% accuracy.

We derived SNR and loss rate relationships using an inverse square law and the novel Goldilocks techniques. At the 90% threshold, both techniques yielded results that were in line with expectations, with the Goldilocks thresholds leading to marginally better success and exhibiting greater discriminatory power. While both techniques are valuable and can be used to cross-validate, Goldilocks makes fewer assumptions, and the statistics suggest that predictions using the thresholds are more reliable.

At the 70% threshold, the experimental data was not in agreement with the predictions at band closure. However, the Goldilocks thresholds led to appropriate categorisation (76.7%) at the band opening. Further experiments are required to determine whether the observations at band closure relate to short term conditions or whether the notion of a two-tier transmission schedule is too unpredictable to be useful.

For an HF-based wireless sensor node, the cost of the improvement is 36 bytes of storage per year for the hourly transmission plan, computed monthly, and assuming a single threshold (90%). The small CPU and memory footprint makes it feasible for micro-controllers to incorporate an HF transmission schedule.

A. Limitations

Both VOACAP and ASAPS resolve time to whole hours, adding complexity when mapping predictions to observations.
The lack of resolution when ionospheric conditions are changing quickly at sunrise and sunset may have contributed to poor results at the 70% threshold at band closure.

B. Future Research

The propagation predictions are based on long-term trends and cannot account for short-term solar events. In future work, we intend to investigate the viability of using dual-frequency GPS to measure the ionosphere directly to account for short term solar events [21]. Although this increases cost, complexity and power consumption, we believe that growing consumer and self-driving vehicle demand will improve efficiency and drive down the cost of dual-frequency GPS.

C. Availability of data

The datasets and high resolution figures used to conduct the research and presented in this paper can be accessed via the DataVerse project [22].

REFERENCES

Efficient Brain-Inspired Hyperdimensional Learning with Spatiotemporal Structured Data

Jiseung Kim
DGIST
js980408@dgist.ac.kr

Hyunsei Lee
DGIST
wwhslee@dgist.ac.kr

Mohsen Imani
UC Irvine
m.imani@uci.edu

Yeseong Kim
DGIST
yeseongkim@dgist.ac.kr

Abstract—Brain-inspired hyperdimensional (HD) computing is a new computing paradigm based on theoretical neuroscience to enable efficient learning. In HD computing, the original data are encoded to points in a high-dimensional space to perform learning with lightweight algebra. In this paper, we propose STEMHD that elicits key features from spatiotemporal data along with a hardware design that empowers computation reuse. Our evaluation shows that STEMHD successfully interprets structural data at a low cost achieving higher accuracy than the state-of-the-art methods. Our evaluation shows that STEMHD improves performance and energy efficiency during the model training by 16.3% and 19.7%, respectively, with a negligible accuracy loss of less than 0.25%. For the model inference, we observe the inference speedup of 1.96x on average.

Index Terms—Hyperdimensional Computing, Alternative Computing, Data representation

I. INTRODUCTION

Brain-inspired hyperdimensional (HD) computing [1] is a new learning paradigm that draws inspiration from research in neuroscience. Interesting findings in the cognition mechanism show that input stimuli of sensory pathways project into massive streams sparsely encoded in the brain, thereby, allowing more neurons to benefit from the expansion [2]. For example, the mammal olfactory system projects sensor inputs to the brain cortex hosting millions of neurons, which is approximately three orders of magnitude more than the original number of the dense inputs [3].

HD computing emulates this by exploiting a high-dimensional space to represent data instead of using conventional numbers. It first encodes given learning data in the original space to vectors in the high-dimensional space, called hypervectors, and then performs the learning procedure with lightweight vector operations. Compared to other neural-inspired learning techniques such as spiking neural networks [4], HD computing offers several unique advantages induced by the high dimensionality, e.g., extremely parallel computations excellent for designing efficient accelerators and high robustness against various noise and hardware failure [5]–[7]. Prior research has shown that HD-based learning could offer significantly higher efficiency during training with multiple orders of magnitude speedup when accelerated on specific hardware designs such as in-memory computing [8] and ASIC [9]. The high efficiency makes HD computing an attractive solution for learning on less-powerful devices, e.g., embedded edge platforms and Internet of Things systems.

Several results of earlier HD learning research present relatively higher efficiency as compared to the computationally expensive deep learning [8]–[14]. However, the required dimension increases with the complexity of classification tasks. For example, most prior work utilizes 10,000 dimensions to achieve acceptable classification quality. An underlying reason of this large dimensionality is that the encoding methods in the existing HD learning are based on randomly-generated bipolar hypervectors to differentiate each feature in a data sample. Since the randomly-generated hypervectors are closer to be orthogonal with a higher dimensions, we need to use a large number of components for a hypervector if all the features are independent. However, the encoding approaches are sub-optimal if there exists information in feature positions. For example, for many sensory input data, the features quantized and extracted from the time and frequency domain should have an explicit correlation with each other. Image data are another example in that pixels close to each other would have a higher relationship than farther ones.

Unfortunately, to the best of our knowledge, all state-of-the-art encoding methods used in prior HD-based classification works lack the ability to interpret such spatiotemporal structure of feature positions. They necessitate rather high dimensionality to satisfy near orthogonality for many randomly-generated hypervectors, resulting in lower computational efficiency and missing the opportunity to achieve better prediction accuracies. In addition, the computed values of random base hypervectors are hard to predict. Thus, the large size of the vector elements should be computed every time, hindering potential data-dependent optimization approaches.

In this paper, we propose an accurate and efficient HD learning architecture, called STEMHD (SpatioTemporal Encoding Mechanism-based HyperDimensional Learning), which explicitly captures the spatiotemporal structure of the given data for higher efficiency. Our solution encompasses a new HD learning principle which utilizes hypervectors preserving feature vectors’ positional information during encoding. The proposed method offers several advantages. First, we observe that the STEMHD encoding principle provides much faster convergence with high accuracy in training and is more robust to dimension reductions. These properties enable various optimization techniques for the accelerator designs of HD computing. Second, the new base hypervectors created by the STEMHD encoding principle have many similar components to each other considering the spatiotemporal information. Based on this property, we propose a new hardware optimization technique to accelerate the proposed encoding strategy further. The proposed technique enables computational reuse
for the IID learning procedure, significantly reducing redundant computations that commonly happen in HD encoding. Third, our approach can be integrated with existing encoding methods seamlessly. It touches on the way to create the base encoding hypervector; but without any extra computational overhead to the state-of-the-art HD learning architectures. Thus, we can utilize the hardware accelerator designs developed so far with very minor modifications while improving prediction accuracy.

We have implemented STEMHD on Kintex-7 FPGA KC705 for various practical problems with structured datasets, including human activity recognition, emotion recognition, and lightweight image classification. Our evaluation results show that the proposed method outperforms the existing HD-based learning in terms of accuracy. The optimization technique based on the computational reuse achieves 16.3% faster runtime and 19.7% energy saving for model training with a very negligible accuracy loss of less than 0.25%. For model inference, it also improves the speed by 1.96x.

II. PRELIMINARIES

HD computing [1] is a brain-inspired complete computing paradigm based on sparse distributed memory (SDM) [15], a human memory model devised in neuroscience. HD computing uses high dimensional, randomly generated vectors, i.e., vectors of dimensions in the thousands, for representation. The high dimensional and random vectors, called hypervectors, resemble neurons in the brain as they are holistically represented. That is, unlike traditional computing where positions of elements define the meaning, in HD computing, information are distributed equally over all its components and the vector as a whole is seen as an entity. Hypervectors can be used to represent entities in hyperdimensional space, or hyperspace. Their high dimension and holistic representation from i.i.d. components make distinct data points orthogonal in hyperspace. Furthermore, they can be efficiently manipulated with lightweight arithmetic like bit-wise addition, called bundling, to combine hypervectors into a single, similar composite hypervector; and bit-wise multiplication or XOR, called binding, that results in a new hypervector that associates its factors.

When using hyperdimensional computing in learning models, hypervectors are used to represent data points in hyperspace such that the distance between similar or related vectors will be closer in distance to one another while unrelated vectors are further in distance. The similar hypervectors in proximity of each other can be bundled together to form a single hypervector that will represent the class of the hypervectors. Inference is then done by comparing the distance (i.e., hamming distance) or similarity (i.e., dot product or cosine similarity) of a new hypervector, called a query hypervector, with each class hypervector. The class hypervector with the closest distance or highest similarity determines the classification of the query hypervector.

The quality of the HD learning is heavily reliant on how to project each raw feature data \( \mathbf{F} \in \mathbb{R}^F \) to a hypervector \( \mathbf{H} \in \mathbb{R}^D \). Current encoding methods first generate a set of hypervectors, called base hypervectors, of very high dimensions (e.g., 10,000) with random values for each dimension.

![Fig. 1: Runtime Breakdown of HD Computing Inference](image-url)

They then use these base hypervectors to encode data into hypervectors. For example, work in [8] utilizes an encoding method, known as ID-Level encoding, which exploits two hypervectors for each feature value, (i) a randomly-generated ID vector, \( \mathbf{I} \in \{−1, 1\}^D \), to distinguish different feature positions, and (ii) a level hypervector, \( \mathbf{L} \in \{−1, 1\}^D \), which quantizes the signal range of the input feature value. By binding the two, it projects an i-th feature value to the hyperspace, and the final hypervector is encoded with the bundling, i.e., \( \mathbf{H} = \sum \mathbf{L_i} \times \mathbf{I_i} \). This encoding method is extremely hardware-friendly in that the computing can be done efficiently only with binary operations; however, it sometimes does not offer sufficient accuracy due to the information loss during the quantization. Prior researchers have proposed more sophisticated encoding methods, e.g., random projection [16] and non-linear encoding [7], which generally achieve higher accuracy than the ID-Level encoding but with extra computational complexity. It should be noted that all the existing encoding methods assume that each feature is independent of each other, assigning completely randomized hypervectors for each feature position.

It is widely observed that the encoding procedure is the main bottleneck of HD computing [17]. To better illustrate the encoding cost, we measure the runtime taken to perform the HD computing-based inference on Kintex-7 FPGA KC705. Figure 1 summarizes the results for five practical applications. The results show that the HD encoding on average takes 10.6× higher runtime than the rest of inference procedure. The high dimensionality of the hypervector exacerbates this inefficiency of the encoding procedure, e.g., including energy, performance, and memory footprints, which are not desired due to limited resources and application requirements.

In this paper, we discuss how to empower the HD encoding with spatiotemporal knowledge. The STEMHD encoding principle is compatible with state-of-the-art encoding methods and can inject the feature vector’s structural information with zero computational overhead at runtime. We accordingly show a hardware optimization technique to reduce the expensive cost of the encoding procedure, which can be implemented on various hardware platforms.

III. STEMHD Design

A. HD Learning Workflow and Background

Figure 2 shows an overview of the proposed STEMHD learning workflow. STEMHD workflow consists of two stages,
training and inference, following the standard ML procedure.

1. **Encoding:** The training and inference share a common encoding module to project the raw data into the hyperspace. As discussed earlier, the encoding method highly dictates the accuracy in the cost of the learning procedures. The key module of STEMHD is the spatiotemporal base hypervector generator, which creates a set of $F$ base hypervectors that defines the relationship between $F$ different features.

With the generated base hypervectors, we can encode each sample in the raw training data into a hypervector with a choice of the state-of-the-art encoders elaborated in Section II. We denote the sample hypervector as $H^k$ where $i$ is the sample index and $k$ is its class label. The hypervector can have either binary or non-binary elements depending on the chosen baseline encoder. We also denote the baseline encoding function with $\phi(\cdot)$, i.e., $\phi(F) = \bar{H}$ for a feature vector $F$ and an encoded sample hypervector $\bar{H}$.

Once the data is encoded we can train the class hypervector. The goal of the training stage is to train a $K \times D$ class hypervector, $C = [\cdots C_{k} \cdots]^T$, which models $K$ classes in the problem datasets with representative $D$-dimensional hypervectors. It encompasses two sub-stages, called initial training and retraining.

2. **Initial Training:** In HD computing, we can memorize different hypervectors using the bundling operation. Inspired by this property, the training procedure starts with learning the initial class hypervector by bundling all encoded hypervectors belonging to each class, i.e., $\hat{C}_k = \sum_i H^k_i$. Since the bundling operation (element-wise addition) for many hypervectors creates a hypervector which is similar to the combined hypervectors, $\hat{C}_k$ is close to sample hypervectors in the $k$-class, but different from the samples in the other classes as long as the encoding is performed adequately.

In STEMHD, the initial training is not mandatory if the following retraining procedure runs for enough iterations. Many prior works adopt it as a standard learning procedure since the unbelievably simple single-pass training can offer high enough accuracy with low computational costs. However, we also observe that STEMHD can also achieve a similar or much higher accuracy level with several epochs with the retraining. Thus, STEMHD provides two paths of the training depending on the user’s preference. For example, if it requires training a model in an extremely lightweight way, e.g., with a few pass learning, we can perform the initial training. Otherwise, we can skip the initial training procedure and start the following retraining substage with the class hypervector filled with zeros.

3. **Retraining:** During the retraining substage, STEMHD fine-tunes the class hypervector by examining each sample hypervector $\bar{H}$. Following the state-of-the-art retraining procedure, STEMHD first identifies which row of the class hypervector has the highest similarity to the sample hypervector:

$$\arg \max_{k \in K} \delta(\hat{C}_k, \bar{H})$$

where $\delta$ is a function which measures the similarity of two given hypervectors, e.g., cosine similarity for non-binary hypervectors, and the hamming distance for binary hypervectors. Once it finds a sample incorrectly classified $k'$ where the correct class is $k$, we refine the class hypervector by $\hat{C}_k = \hat{C}_k + \bar{H}$ and $\hat{C}_{k'} = \hat{C}_{k'} - \bar{H}$. The element-wise addition and subtraction populate the hyperdimensional pattern of $\bar{H}$ into the correct class $\hat{C}_k$ over $\hat{C}_{k'}$. We repeat the retraining procedure over multiple epochs to converge the model.

4. **Inference:** During the inference stage, we use the same encoding method as the training stage. For a given raw feature vector, STEMHD uses the same based hypervector generated and used in the encoding to encode it to a hypervector, called a query hypervector $\bar{Q}$. Then, we perform the similarity computation procedure shown in Equation 1 to predict the class of the sample.

### B. Spatiotemporal Base Hypervector Generation

Before elaborating our STEMHD encoding principle, we first discuss the HD operation property with random vectors and the existing encoding mechanisms. Let $\bar{A}$ and $\bar{B}$ be two hypervectors randomly sampled from $\mathbb{R}^D$. The two hypervectors are nearly-orthogonal, i.e., $\delta(\bar{A}, \bar{B}) \approx 0$. For any hypervector, say $\bar{C}$, binding with each of the two hypervectors creates another two nearly-orthogonal hypervectors, i.e., $\delta(\bar{C} \times \bar{A}, \bar{C} \times \bar{B}) \approx 0$.

Due to this property, the existing encoding cannot keep the spatiotemporal information. For example, let us consider a feature vector whose elements are consecutive measures in a time series, $\bar{F} = (f_0, \cdots, f_{F-1})$. Let us denote the feature vector for a shifted time series with a new measure $e'$ as $\bar{E} = (e_0, \cdots, e_{F-1})$ where $e_0 = e'$ and $e_t = f_{t-1}$ for $t > 0$. The two feature vectors $\bar{F}$ and $\bar{E}$ intrinsically have similar data; however, the existing encoding methods assign different random base hypervectors for $f_t$ and $f_{t+1}$, and thus projects them into orthogonal hyperspaces and $\delta(\phi(\bar{F}), \phi(\bar{E})) \approx 0$.

We propose a rigorous mathematical principle to generate the base hypervectors sustaining spatiotemporal data structure to address this issue.

#### Generation for 1D feature:

Supposed a feature vector $\bar{F} = (f_0, \cdots, f_{F-1})$. Our goal is to create a set of correlated base hypervectors, $\{\bar{B}_1, \cdots, \bar{B}_1, \cdots, \bar{B}_{F-1}\}$, so that hypervectors having closer feature indexes are more similar to each
Fig. 3: 1D Base Hypervector (HV) Generation

other. We define here a kernel size, denoted as $K$, which is a distance in the feature index considered as completely different features, i.e., $\delta(\vec{B}_f, \vec{B}_{f+K}) \approx 0$.

Figure 3 describes the base hypervector generation procedure of STEMHD. The first step is to create $\left[\frac{K}{2}\right]$ hypervectors, called partition hypervectors, by exploiting the random sampling method used in the baseline encoder. The created hypervectors correspond to each feature at $(N \times K)^{th}$ index for $N > 0$, i.e., differentiating all the distinct features defined with $K$. The encoding of the other features are then calculated using the HD interpolation with $\vec{B}_{N,K}$ and $\vec{B}_{N,K+K}$. The HD interpolation of $\vec{B}_{N,K+\lambda}$ is done by taking the left $\lfloor(1 - \lambda/K) \cdot D\rfloor$ elements from $\vec{B}_{N,K}$ and the rest of the elements from $\vec{B}_{N,K+K}$.

Please note that a base hypervector generated with our method has many duplicated components to the hypervectors for neighbor features. This property can enable new data-oriented optimization techniques. We discuss our optimization strategy in Section IV.

Generation for 2D feature: The base hypervector generation method for 2D feature encoding is a straightforward expansion of the one used for the 1D data. Figure 4 illustrates the procedure. We create the $\left[\frac{K}{2}\right]^2$ random partition hypervectors which divide the feature vectors into a 2D grid and regard each created hypervector as a $\sqrt{D} \times \sqrt{D}$ hypermatrix. Then, depending on the index for each 2D feature, it interpolates four neighbor hypervectors and determines the base hypervisor.

The proposed STEMHD principle generates the base hypervectors, which preserve the distance of the feature indices. For example, the base hypervector at the center is similar to its neighboring base hypervectors and orthogonal to the others whose index difference is larger than $K$ at any axis. Note that we exploit the linear HD interpolation in this paper due to its simplicity. It is also possible to use other nonlinear interpolation methods, e.g., 2D Gaussian function, with minor modification.

C. STEMHD Integration

We revise the state-of-the-art encoding methods to use the spatiotemporal base hypervectors. In this paper, we introduce how to upgrade the three widely-used encoding method.

1D-Level Encoding [8]: The encoding function is formulated with $\vec{H} = \sum_i \vec{I}_i \times \vec{I}_i$, as discussed in Section II. To consider the spatiotemporal data structure, we replace the ID hypervector $\vec{I}_f$ corresponding to each feature with the generated

Fig. 4: Generation and Interpolation of 2D Base Hypervector bases, $\vec{B}_f$. It thereby does not incur any extra computational overhead in the encoding and learning stages.

Random Projection Encoding [16]: Random projection encoding uses a set of random hypervectors drawn from $\{-1, 1\}$ to represent different features and multiplies them with the original feature values to obtain the encoded hypervector. It overcomes the information loss issue of the ID-Level encoding due to the quantization. In the integration with STEMHD, we simply substitute the random hypervectors with our base hypervectors, i.e., $\vec{H} = \vec{F} \times \vec{B}$ where $\vec{B} = \cdots \vec{B}_f, \cdots \vec{I}_t$.

Nonlinear Encoding [7]: Non-linear encoding often provides the highest accuracy level among the state-of-the-art encoders, comparable to fully-connected neural networks for many problems. This method samples elements of the 2D vector from Gaussian distribution, i.e., $\vec{I}_t \sim N(0, 1)$. The encoded hypervector is computed by $\vec{H} = \cos(\vec{F} \times \vec{I})$ where $\vec{I} = \cdots \vec{I}_t, \cdots \vec{I}_t$. The cosine function amplifies the non-linearity to $\vec{H}$. We can also binarize $\vec{H}$ by taking its sign bit. To integrate with this method, STEMHD creates the partition hypervectors with $N(0, 1)$ and interpolates them to create the base hypervectors. Then, we can use $\vec{B}$ instead of $\vec{I}_t$ during the encoding.

IV. STEMHD HARDWARE OPTIMIZATION

A. Computation Reuse of STEMHD

The encoding method achieving higher accuracy usually has a larger amount of computations. For example, non-linear encoding shows a great improvement in classification quality; however, it requires computing hypervectors whose elements are non-binary. In many hardware platforms, it may cost critical overheads, e.g., an FPGA has a limited number of DSPs, which are also power-consuming.

In this paper, we propose an optimization technique for the STEMHD encoding procedure. Our proposed optimization is based on the fact that the base hypervectors generated by STEMHD have many duplicated components, allowing the opportunity for computation reuse. Even though the encoding principle of STEMHD can be seamlessly integrated with the state-of-the-art encoding methods without any extra computation cost at runtime, the computation reuse can further reduce the runtime and power consumption of the accelerator designs.

Figure 5 illustrates the idea of the proposed optimization technique. As discussed in Section III-B, thanks to the nature of the interpolation, the base hypervectors next to each other,
Fig. 5: Computation Reuse of STEMHD (when $f_t - f_{t+1} < \tau$)

$B_t$ and $\bar{B}_{t+1}$, have many overlapped components (1). We first project a feature value $f_t$ to the hyperspace during the encoding, following the usual process of the original encoder. We store the computed results for each dimension while computing the bundling operation to $\bar{H}$ (2). Then, before processing the next feature value $f_{t+1}$, we check if there is a meaningful difference between $f_t$ and $f_{t+1}$, i.e., $|f_t - f_{t+1}| < \tau$ where $\tau$ is a configurable threshold value. If they are similar enough, we can assume that the computation outputs for the duplicated hypervector components between $B_t$ and $\bar{B}_{t+1}$ will also be similar. In this case, we can reuse the computation results and only need to compute the non-overlapping components (3). It should be noted that, for many datasets, the neighboring features often have similar values, e.g., consecutive time-series features and pixel values of images.

B. FPGA Implementation

We implement an FPGA-based hardware accelerator that tailors the proposed optimization technique. The HDC encoding for feature vectors involve large-scale vector-matrix multiplications. FPGA has the option of using either lookup tables (LUTs) or DSPs to support the encoding process. LUTs are bitwise logic of FPGA, and they are typically slow in implementing high-precision multiplication. On the other hand, FPGA has a limited number of DSPs, which are computationally costly, thus it cannot provide the parallelism required by the encoding module.

We implement an FPGA implementation that exploits computational reuse to eliminate a large portion of redundant computation in STEMHD. To maximize STEMHD throughput, the FPGA processes an application through the following steps: (i) it fetches an input data into FPGA block RAMs (BRAMs). (ii) The FPGA compares the similarity of a chunk of pixels with their neighbors in parallel. This comparison is performed using FPGA LUTs. (iii) For features with a close distance to their neighbors, FPGA reuses the pre-computed encoding. For other features, the FPGA uses DSPs to support multiplication and LUT resources to compute the multiplied vectors. This architecture minimizes the amount of repeated computation while enabling a data flow that maximizes FPGA resource utilization. For instance, our evaluation shows that exploiting LUTs for vector accumulation can improve FPGA throughput by about two times. This along with the computational reuse further improves FPGA efficiency compared to a naive implementation that relies on expensive and limited DSPs for computation.

V. EXPERIMENTAL RESULTS

A. Experimental Setup

We implement and synthesize STEMHD using Verilog on the Kintex-7 FPGA KC705 Evaluation Kit. We estimate the runtime and power consumption using Xilinx Vivado Design Suite. To compare the efficiency of our computation reuse technique, we also modified our source code so that the FPGA fully computes all the dimensions every time, and measured the performance/energy-related metrics. We compare the accuracy of HD-based learning with state-of-the-art DNN models created by AutoKeras [18] for automated hyperparameter tuning. It should be noted that the base hypervector generation of STEMHD is an one-time procedure, and the overhead is very negligible. In our experiment, it only takes less than 0.01% of the runtime for the entire training/inference procedures.

Dataset: We use six practical datasets for our benchmark, summarized in Table I. There are three datasets with 2D features; MNIST and Fashion have the same data structure, consisting of 28 x 28 bit images. The Face dataset has grey-scaled images of 32 x 32 bits. The rest of the three datasets have 1D features. In particular, the Emotion dataset has two groups of 1D Fourier-transformed features extracted from two types of EEG brainwave signals. Thus, we apply our STEMHD encoding mechanism for each group, respectively, and bundle them to create a single hypervector for each sample. HAR has nine groups of 1D features, where each group has 128 time-series features extracted from different sensors.\footnote{In the original dataset, it also includes extracted features; however, in our evaluation, we instead use raw time-series features to verify the impact of considering the temporal locality on the accuracy.} Similarly, we encode each group individually and combine them to generate the final encoded hypervector.

B. Accuracy Evaluation

Figure 6 first compares the accuracy of the proposed STEMHD mechanism over other learning techniques. For the DNN, we use the auto-tuned DNN models discussed in Section V-A. For the HD-based learning, we use three encoding methods, nonlinear encoding for all datasets, random projection encoding for 2D-feature datasets as the baseline (MNIST, Fashion, and Face) and ID-Level encoding for 1D-feature datasets as the baseline (Emotion, Heart, and HAR). The results show that the STEMHD encoding mechanism improves the accuracy of the datasets that have spatiotemporal information more than the state-of-the-art encoding method. For example, for the Heart dataset, the STEMHD encoding principle increases the accuracy by 5.1% as compared to the ID-Level encoding. Note that this improvement is obtained without any runtime costs as the proposed method
only changes the way to generate the base hypervectors. On average, STEMHD provides an improvement in accuracy of 1.3% for the nonlinear encoding and 1.8% for the ID-level hypervector encoding over the cases not using the spatiotemporal feature encoding. Compared to the DNN, STEMHD with non-linear encoding has only a 0.56% accuracy loss.

To better present how STEMHD obtains the accuracy improvement by changing the base hypervector, we visualize the information included in the hypervectors using the hypervector decoding procedure shown in [16], which extracts original features from hypervectors and maps them to the original space. Figure 7 compares how the sample and class hypervectors are encoded for the Fashion dataset using STEMHD as compared to the state-of-the-art encoding scheme, which assumes the independence of all features. The results show that, without using the STEMHD encoding principle, the encoded hypervectors have a non-negligible amount of noise in its high-dimensional patterns (the second column). As a result, the class hypervector initially generated by the single-pass training (the fifth column) exhibits undesired random fluctuations in the extracted features. In contrast, the STEMHD principle plays the role of an image filter, which removes the noise from the encoded hypervectors (third and fourth columns). The trained hypervectors at the last column, thereby more clearly represent the patterns we want to learn.

C. Evaluation of Computation Reuse Technique

As discussed in Section IV, STEMHD enables the computation reuse optimization for the expensive encoding procedure based on the fact that the neighboring hypervectors share common components. Figure 8 shows the accuracy loss when using different reuse threshold values, \( \tau \). The results show that the accuracy does not hugely change when using a small \( \tau \) value. For example, when \( \tau = 1\% \), the accuracy loss is very negligible, only 0.25% on average. We also observe that

D. Exploiting High Accuracy for Efficient Design

One of the attractive properties of HD computing is its high efficiency. For example, as compared to the DNN training, the HD-based learning achieves multiple orders of magnitude improvement in terms of the performance and energy efficiency [8], [25]. It is also widely known that HD-based learning can provide sufficient prediction quality with a small number of training epochs [16].

To better understand how STEMHD works in such cases, we collected the detailed traces of accuracy changes over different training epochs. Figure 11 shows the experimental results for two representative datasets, MNIST and Emotion. The results show that, even with a single epoch training, the STEM-based encoding method significantly outperforms the accuracy of the original non-linear encoding method. For example, for MNIST, only with a couple of epochs, it starts converging with the accuracy of 97%, which is highly close to the maximum
Fig. 9: Efficiency Improvement of Computation Reuse Technique During Training

Fig. 10: Energy Efficiency Improvement During Inference

we can accurately project different features into a single hypervector using a smaller dimension size. To verify this idea, we evaluated how the accuracy changes for different dimension sizes.

Figure 12 summarizes the results for the two representative datasets. The results show that STEMHD is much more robust to the dimensionality reduction. For example, when using $D = 3,600$, it only loses 0.74% accuracy for the MNIST dataset as compared to the case of $D = 10,000$. In contrast, the hypervectors encoded without using STEMHD shows a relatively high degree of the accuracy drops, e.g., 3.98% for Emotion when using $D = 3,600$, where the loss is 1.64% for the STEM-based encoding. This property enables various dimension-wise optimization techniques. To estimate how much efficiency improvement can be obtained, we collected the speedup and energy efficiency improvement for training when reducing the dimension size in our FPGA design. The results show 50% speedup and 2.45× energy efficiency improvements if using $D = 3,600$ as compared to $D = 10,000$.

VI. RELATED WORK

Work in [1] explains the concept of HD computing and discusses its mathematical foundations. Earlier research presented the practical value of HD computing for various learning tasks [9], [16], [26], [27]. Recently several industries have also focused on HD computing to enable lightweight artificial intelligence, e.g., at Google [28], Numenta [25], and Vicarious [29]. Prior work also utilizes the HD computing technique for light-weight learning including language recognition [8], [30], human activity recognition [11], multimodal sensor fusion [31], robotics [12], DNA pattern matching [32], [33], and security [34]. They show that the HD computing can achieve the accuracy level comparable to the deep neural networks. The highly parallelizable nature of HDC also enables various hardware designs including low-power GPU and ASIC [9]. Since HD computing performs learning tasks in a lightweight
Fig. 12: Accuracy/Efficiency vs. Dimension fashion, it has often been considered as a suitable alternative to deep learning in embedded devices having limited resources.

VII. CONCLUSION

In this paper, we propose STEMHD, which performs lightweight, efficient learning with high accuracy based on HD computing. Unlike the existing HD encoding method, which is agnostic to the spatiotemporal information of given data, STEMHD explicitly takes into account the structural data to project the features into the hyperspace. The proposed encoding principle can be implemented with zero runtime cost and further enable data-oriented optimization technique as it creates repeated computation workloads. Based on this finding, we accordingly propose a computation reuse technique which allows the effective efficiency and accuracy tradeoff. We implement an FPGA-based accelerator and evaluate the proposed STEMHD mechanism with real-world datasets. The results show that the proposed method outperforms the existing encoding method in terms of accuracy. Combined with computational reuse, it achieves 16.3% runtime and 19.7% energy saving for the training while guaranteeing very negligible accuracy loss. For the inference, we observe significant improvements of 1.96× average speedup. We also discuss that we can devise various optimization techniques, e.g., epochwise and dimension-wise, utilizing the improved accuracy.

ACKNOWLEDGMENT

This work was supported by the National Research Foundation (NRF) of Korea (NRF-2018R1A5A1060031). This work was also partially supported by National Science Foundation (NSF) #2127780, Semiconductor Research Corporation (SRC) Task No. 2988.001, Department of the Navy, Office of Naval Research, grant #N00014-21-1-2225, and generous gifts from Cisco.

REFERENCES

Sparse Matrix-Vector Multiplication Cache
Performance Evaluation and Design Exploration

1st Jianfeng Cui
School of Computer
National University of Defense Technology
Changsha 410073, Hunan, China
Email: cjf19@nudt.edu.cn

2nd Kai Lu
School of Computer
National University of Defense Technology
Changsha 410073, Hunan, China
Email: kailu@nudt.edu.cn

3rd Sheng Liu
Science and Technology on Parallel and Distributed Processing Laboratory
National University of Defense Technology
Changsha 410073, Hunan, China
liusheng83@nudt.edu.cn

Abstract—In this paper, we conducted a group of evaluations on the SpMV kernel with sequential implementation to investigate cache performance on single-core platforms. We verified a similar pattern inside a suite of sparse matrices covering various domains, which makes cache hit rate extraordinary inspiring in a sequential environment. This implicit regularity drove us to propose a cache space splitting approach, aiming at a better locality in dense vector accessing and utilization of large cache capacity in modern processors. Finally, we explored the design space of cache on Matrix 3000 GPDSP and proposed a group of cache parameters, based on our experimental results.

Index Terms—SpMV, cache, sparse, matrix, PIN, simulation

I. INTRODUCTION

Sparse matrix-vector multiplication (SpMV) is a fundamental computing kernel in various domains including machine learning, scientific computing, and signal processing. Compared to dense linear algebra computing, SpMV suffers from inefficient exploitation of current computing platforms, due mainly to the sparsity and irregularity of nonzeros in the matrix. To achieve better performance, a multitude of works during the past decades have focused on the storage format of sparse matrices [1]–[5] and the optimization of algorithms on specific hardware platforms [6]–[8]. However, little attention has been paid to cache performance, while a terrible cache hit rate exacerbates memory-cache traffic and consequently hinders overall performance. Large matrix scale and irregular distribution of nonzeros are the main reasons for poor cache performance. Moreover, parallel computing incurs complicated cache behavior on shared cache multi-core platforms.

Some previous studies are positive in alleviating cache misses. Williams, et al. proposed sparse cache blocking [9], an optimization to allocate a fixed fraction of cache lines for different data resources. Xie, et al. introduced an efficient compressed storage format for SIMD computing platforms [5], which is less sensitive to sparsity and irregularity, hence it makes better utilization of vectorization operations on modern processors. However, the SpMV kernel is still notorious for its high cache miss rate and poor exploitation of peak performance.

To better understand the underlying problem of cache performance on shared cache multi-core parallel platforms, we turned back to sequential SpMV computation to fully investigate cache behavior, aiming at inspiration for a better solution on multi-core platforms. We conducted our experiment with a software approach to simulate different hardware environments. Our experimental results show that a massive amount of sparse matrices that arise in real applications are not completely irregular. On the contrary, we identified some universal patterns in these matrices and obtained positive results in our evaluation. The results show that cache miss rate changes little when cache size shrinks from 16 Mbyte to 64 Kbyte, which indicates that sparse matrices with similar patterns as tested matrices are insensitive to cache capacity.

Furthermore, we explored the design space of the cache on our Matrix3000 GPDSP with overall consideration of future computing tasks. Additionally, we proposed an exclusive-write-inclusive-read cache splitting approach to decrease cache misses and efficiently utilize shared cache space. We refer to this approach as virtually individual cache space, or VICS. On shared cache multi-core platforms with VICS, each core is allowed to own an individual cache space and has the exclusive privilege to write into it. Moreover, each core can still read from outside the exclusive space, which means VICS can benefit from both large cache capacity and straightforward cache behavior in sequential processing. Details of experiment and analysis are discussed in the remainder of this paper.

This work was supported in part by Science and Technology on Parallel and Distributed Processing Laboratory (PDL) under grant WDZC20200500111.

* Sheng Liu is corresponding author.
<table>
<thead>
<tr>
<th>Group</th>
<th>Dataset</th>
<th>Domain</th>
<th>Spy Plot</th>
<th>Rows</th>
<th>Nonzeros per Row</th>
<th>Vector Size/Mbyte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Generated</td>
<td>Matrix 1</td>
<td>HPCG_SpMV</td>
<td>—</td>
<td>16,777,216</td>
<td>27</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>Matrix 2</td>
<td>HPCG_SpMV</td>
<td>—</td>
<td>16,777,216</td>
<td>27</td>
<td>128</td>
</tr>
<tr>
<td></td>
<td>asia_osm</td>
<td>Road networks</td>
<td></td>
<td>11,950,757</td>
<td>2.13</td>
<td>91</td>
</tr>
<tr>
<td></td>
<td>cago15</td>
<td>DNA electrophoresis</td>
<td></td>
<td>5,154,859</td>
<td>19.24</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>delaunay_n23</td>
<td>Computational geometry</td>
<td></td>
<td>8,388,608</td>
<td>6.00</td>
<td>64</td>
</tr>
<tr>
<td></td>
<td>GAP-road</td>
<td>Road networks</td>
<td></td>
<td>23,947,347</td>
<td>2.41</td>
<td>183</td>
</tr>
<tr>
<td></td>
<td>hugtrace-00020</td>
<td>Network</td>
<td></td>
<td>16,002,413</td>
<td>3.00</td>
<td>122</td>
</tr>
<tr>
<td></td>
<td>mawi_201512012345</td>
<td>Internet traffic</td>
<td></td>
<td>18,571,154</td>
<td>2.05</td>
<td>142</td>
</tr>
<tr>
<td></td>
<td>nlpklk200</td>
<td>Optimal problem</td>
<td></td>
<td>16,240,000</td>
<td>27.11</td>
<td>124</td>
</tr>
<tr>
<td></td>
<td>kmr_V2a</td>
<td>GenBank</td>
<td></td>
<td>55,042,369</td>
<td>2.13</td>
<td>420</td>
</tr>
<tr>
<td></td>
<td>stokes</td>
<td>Semiconductor process</td>
<td></td>
<td>11,449,533</td>
<td>30.51</td>
<td>87</td>
</tr>
<tr>
<td></td>
<td>venturiLevel3</td>
<td>Numerical simulation</td>
<td></td>
<td>4,026,819</td>
<td>4.00</td>
<td>31</td>
</tr>
<tr>
<td>2</td>
<td>F1</td>
<td>Structural problem</td>
<td></td>
<td>343,791</td>
<td>78.06</td>
<td>2.62</td>
</tr>
<tr>
<td></td>
<td>hood</td>
<td>Structural problem</td>
<td></td>
<td>220,542</td>
<td>44.87</td>
<td>1.68</td>
</tr>
<tr>
<td></td>
<td>ldoor</td>
<td>Structural problem</td>
<td></td>
<td>952,203</td>
<td>44.63</td>
<td>7.26</td>
</tr>
<tr>
<td></td>
<td>mac10848</td>
<td>Structural problem</td>
<td></td>
<td>10,848</td>
<td>113.36</td>
<td>0.08</td>
</tr>
<tr>
<td></td>
<td>G3_circuit</td>
<td>Circuit simulation</td>
<td></td>
<td>1,585,478</td>
<td>4.83</td>
<td>12.10</td>
</tr>
<tr>
<td></td>
<td>Na5</td>
<td>Quantum chemistry</td>
<td></td>
<td>5,832</td>
<td>52.41</td>
<td>0.04</td>
</tr>
<tr>
<td></td>
<td>thermal2</td>
<td>Thermal</td>
<td></td>
<td>1,228,045</td>
<td>6.99</td>
<td>9.37</td>
</tr>
</tbody>
</table>

Fig. 1. Overview of the evaluated matrices. Including 2 generated by HPCG_SpMV and 17 collected from the University of Florida Sparse Matrix Collection.
We conducted our experiment on Linux server which runs CentOS 7.2 operating system. The server has a 2Tbyte RAM size, which is capable to accommodate the entire data structure of our SpMV program. We used the basic computing kernel $y = A \cdot x$. The compiler version is GCC 6.3.0.

Two groups of matrices were investigated, namely Group 1 and Group 2. Group 1 contains 10 large matrices that cover various domains: road network, numerical simulation, internet traffic, etc. We conducted an evaluation on Group 2 as a contrast with the previous research of Tanabe, et al. [10], therefore, Group 2 consists of 7 identical matrices in their research. All the 17 matrices were collected from the University of Florida Sparse Matrix Collection [11], an overview of the evaluated matrices appears in Fig. 1. Additionally, Matrix 1 and Matrix 2 were generated by the HPCG_SpMV program to simulate the exactly regular and random situations, respectively. To acquire randomly distributed nonzeros, the function that generates multi-diagonal matrix was modified.

The storage format used in our experiment was compressed sparse row (CSR), which is one of the most common data structures used to store sparse matrices. The SpMV kernel in HPCG_SpMV was also modified to adapt the CSR format. The Pseudocode of our CSR-based SpMV kernel is illustrated in Fig. 2.

To flexibly gain results of different cache parameters, we performed a software approach to simulate real hardware platform. Intel Pin [12] is utilized to record memory access traces, by combining routine-level and instruction-level instrumentation. The version of Intel Pin we used is Pin 3.15. With a comprehensive memory access record of the SpMV kernel, we are able to reproduce cache behavior and output cache miss rate on CacheSim$^1$.

The main properties of cache that we considered are as follows:

**Cache size.** Normally, Cache size is the most significant factor of cache miss rate. If the cache is capable to accommodate all the data to be used, then only compulsory misses occur. The matrices we used were verified to occupy a memory space larger than cache size to avoid this situation. The dense vector corresponding to the smallest matrix venturiLevel3 in Group 1 occupies 31 Mbyte memory space, which is almost double the maximum cache size we simulated.

**Cache line size.** Compulsory misses arise when memory blocks are fetched into the cache. If this block is used repeatedly - which requires excellent locality – compulsory miss rate will be amortized by all the iterations. It is natural to believe that a large cache line size means few compulsory misses. However, when the cache size is fixed, a larger cache line size also means fewer cache lines, thus the conflict miss rate grows. Furthermore, latency increases when cache line size is larger, this makes it a classical trade-off in cache design. In this paper, we evaluated cache line size from 32 bytes to 256 bytes, which covers most of the modern processors.

**Replacement policy.** As is assumed above, the cache size is smaller than data, thus capacity misses and conflict misses are inevitable. The problem faced by designers is which cache line is to be replaced. In our experiment, we used Random and Least Recently Used (LRU) to evaluate the impact of replacement policy.

The mapping scheme is also an important factor of cache performance. However, when talking about it, the trade-off between latency and accuracy is what we consider, because cache hit rate is always better with more ways. In this paper, we are concentrating on cache hit rate rather than latency and we used 8-way set associative as mapping scheme of our cache, which is common in modern processors. Fig. 3

---

<table>
<thead>
<tr>
<th>Property</th>
<th>Option</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cache size</td>
<td>64 Kbyte ~ 16 Mbyte</td>
</tr>
<tr>
<td>Cache line size</td>
<td>32 byte ~ 256 byte</td>
</tr>
<tr>
<td>Replacement policy</td>
<td>LRU / Random</td>
</tr>
</tbody>
</table>

---

$^1$https://github.com/jiangxincode/CacheSim
III. RESULTS AND DISCUSSION

We initiated our study on Matrix 1. Fig. 5(a) shows the cache miss rate of the entire computing kernel, loading data from dense vector \( x \), and storing data to vector \( y \), respectively.

The miss rate of storing is exactly the compulsory miss rate of accessing vector \( y \), which is primarily based on cache line size because \( y \) is stored contiguously and accessed sequentially. Miss rate of accessing \( x \) is negligible, which is in violation of common sense. However, this can be explained by the nonzero data layout of Matrix 1.

Fig. 4 pictures the top left corner of the minimized version of Matrix 1. White blocks represent zeroes and green blocks are nonzeros. When rows of the matrix are accessed sequentially, an excellent temporal locality is identified in vector \( x \), because it is indexed by column indices of nonzeros.

We modified the HPCG_SpMV program to generate a randomly distributed matrix as a contrast, in which the column indices have a uniform distribution. As Fig. 5(b) shows, the result perfectly corresponds to the probability distribution. Collectively, when the SpMV kernel is sequentially executed, the matrix pattern determines the cache miss rate.

Then we conducted the evaluation with matrices in Group 1, and the results are presented in Fig. 6. These matrices from real applications are not as regular as Matrix 1, which is straightforward according to the spy plots. Apparently, the results are not favorable compared to the result of Matrix 1, but they are still much better than those conducted on parallel platforms [5], [9]–[11]. Given the circumstances, we tried to carry out detailed scrutiny to recognize the internal patterns of these matrices. We checked more than 200 random parts of the matrices and figured out that:

- Nonzeros in the same row are not totally independent, but partly contiguous.
- Nonzeros in adjacent rows also reside nearby.

We can intuitively understand it through a thumbnail of first 17 rows in asia_osm, as is depicted in Fig. 7.
size that smaller cache size may affect the cache miss rate less than we thought. The result pictured in Fig. 6 shows the cache miss rate increase when cache size shrinks from 16 Mbyte to 64 Kbyte. According to the absolute or relative increase, most matrices are insensitive to cache size, which is supportive to our assumption.

Fig. 8 depicts the result of evaluation on Group 2, as the contrast with previous work of Tanabe, et al. [10] We implemented our simulation using exactly the same cache properties in their work. Apparently, Cache miss rate in our sequential environment is much better than that obtained with CUDA.

As is discussed above, there exists a large fraction of contiguous memory access in our SpMV kernel, which makes compulsory misses critical. Thus, we examined the impact of cache line size, and the results are illustrated in Fig. 9. For most matrices, the cache miss rate decreases rapidly when the cache line size scales from 32 bytes to 64 bytes. However, it slows down severely when the cache line size reaches 128 bytes. Therefore, we inferred that cache line size larger than 128 byte is inefficient in either miss rate or latency.

Fig. 10 presents how much replacement policy affects cache miss rate. Most results using LRU replacement are slightly better than those using random strategy. Meanwhile, this effect grows when the cache size is smaller. It can be intuitively figured out that when the cache size is smaller, the probability of useful cache line being randomly replaced is higher, which consequently results in more cache misses.

Since it has proved that for matrices in Group 1, or to go a step further, for multi-diagonal-like matrices, large cache size brings a disproportionate fraction of performance improvement when writing into it. But when reading, large cache size shows greater significance when the nonzeros are more irregular. This prompted us to introduce an efficient strategy to utilize large cache capacity. We refer to this approach as Virtually Individual Cache Space, or VICS. Fig. 10 shows the hierarchy of shared cache multi-core processor with VICS. VICS is not a physical component in the shared cache, it is a partition of cache space and each part is assigned to a core, dynamically or manually.

The rationale of VICS can be summarized as follows:

- Every core is assigned with a VICS in the last level shared cache and has the exclusive privilege to allocate cache lines into it. As is discussed above, memory access is partly contiguous. If the matrix is partitioned by row, every core will execute a contiguous part of the whole sequential computation. This ensures an excellent cache hit rate on every core. Furthermore, with this rule, the complicated cache behavior in shared cache multi-core system is no more an obstacle because no core is capable to evict cache lines located in other core’s VICS. This can significantly alleviate conflict misses.

- Every core is allowed to read cache lines from outside its assigned VICS. According to our observation, most real applications are partly regular and partly disordered. These disordered parts are the main reasons for poor cache performance, as we have examined. Small cache size of VICS exacerbates it, therefore we keep every VICS available for other ones when reading to eliminate this side effect.

These two simple rules make excellent utilization of the SpMV characteristics we observed above and efficiently exploit cache capacity. Based on these experimental results, we can summarize the design space of cache on our future Matrix 3000 GPDSP:

**Cache size.** Our experimental results imply that 8 Mbyte cache size is capable to achieve excellent performance, and still retains scalability. It is a cost-effective choice considering our future possible tasks.

**Cache line size.** As is discussed above, we can benefit little from a larger cache line when it reaches 128 bytes. Worse still is the increasing of conflict misses and cache line access latency when cache line size grows. Thus, we proposed 128 bytes as our cache line size.

**Replacement policy.** It is extremely complicated to analyze qualitatively because it requires instruction-level investigation, that is why we normally use quantitative results to guide our
design. But since we found some patterns in the matrices, we can intuitively infer the better replacement policy, as analyzed above. Our evaluation results also support our assumption: LRU performs slightly better than Random in our experiment.

IV. CONCLUSION AND FUTURE WORK

In this paper, we evaluated the cache miss rate of SpMV kernel in sequential environment, which presented much more inspiring results compared with those on multi-core parallel platforms. We analyzed a wide set of sparse matrices consisting of various domains and recognized some regular patterns. The internal regularity of these matrices is the main reason for the excellent performance. We also proved that cache size affects little in sequential SpMV computation with these matrices.

To take advantage of the significant cache effect of sparse matrices with similar patterns and make full utilization of large cache capacity in state-of-the-art processors, we introduced Virtually Independent Cache Space, namely VICS. VICS is a virtual space in the cache. It protects cache lines inside it from being evicted by other cores to alleviate conflict misses. Meanwhile, it is available for all cores when reading, which is essential in reducing capacity misses.

Finally, we explored our cache design space on Matrix 3000 GPDSP, based on our experimental results. We selected a group of cache parameters consisting of cache size, cache line size, mapping scheme, and replacement policy. Our future work will be conducted in the following directions:

First, we are pursuing a more inclusive VICS approach which applies to more sparse matrix patterns and more sparse kernels, not only SpMV with multi-diagonal-like matrices.

Second, the essential criteria of sparse problems is throughput. We are looking for a more efficient algorithm and low overhead cache implementation with VICS.

Third, some of our experiments in this paper were limited by the simulating tools, thus, we are updating the tools to better simulate real hardware systems.

REFERENCES

A Mixture Density Network Approach to Predicting Response Times in Layered Systems

Zifeng Niu
Department of Computing
Imperial College London
London, UK
zifeng.niu19@imperial.ac.uk

Giuliano Casale
Department of Computing
Imperial College London
London, UK
g.casale@imperial.ac.uk

Abstract—Layering is a common feature in modern service-based systems. The characterization of response times in a layered system is an important but challenging analysis dimension in Quality of Service (QoS) assessment. In this paper, we develop a novel approach to estimate the mean and variance of response time in systems that may be abstracted as layered queueing networks. The core step of the method is to obtain the response time distributions in the submodels that are used to analyze the layered queueing networks by means of decomposition. We model the conditional response time distribution as a mixture of Gamma density functions for which we learn the parameters by means of a Mixture Density Network (MDN). The scheme recursively propagates the MDN predictions through the layers using phase-type distributions and performs convolutions to gain the approximation of the system delay. The experimental results show an accurate match between simulations and MDN predictions and also verify the effectiveness of the approach.

Index Terms—Quality of Service, response time distribution, queueing network, mixture density network, layered system

I. INTRODUCTION

The end-to-end response time experienced by customers is an important aspect of service quality assurance. The response time serves as a key performance index, but in some situations the mean value of this metric is insufficient for system designers, who may want to know the quantile statistics or the extent of variability in order to assess compliance to service level objectives. Although the problem of approximating response time distributions in queueing networks (QN) has been extensively studied during the last few decades, it still remains a challenging task because conventional analytic methods are inherently limited by their underpinning technical assumptions, such as those that prescribe overtake-free job behavior under first come first serve (FCFS) scheduling [1]. In addition, many services today such as distributed software applications or business processes feature concurrent services structured according to a layered structure. Such systems are not simple to model using ordinary queueing networks since layering is a form of simultaneous resource possession that does not meet the product-form solution assumptions [2]. The lack of simple closed-form analytical solutions poses a challenge for the estimation of response time distributions.

In this paper, we propose a method to provide accurate estimations on variance as well as mean value of the response time in layered systems. The method requires estimating response time distributions in a set of closed queueing networks that are constructed from a layered queueing network (LQN). However, the current analytic methods are not stable to use since they are restricted by strict assumptions on service discipline, number of servers, and Markovian networks. Fortunately, machine learning (ML) poses a number of opportunities and one of the goals of this paper is to understand how ML can benefit QN and LQN theory. We use a suitable ML method named Mixture Density Network (MDN) to approximate response time distribution conditional on the given information of a closed queueing network. The MDN uses a mixture model to fit probability distribution and uses a fully-connected neural network to map the conditional relation. Armed with MDNs, our approach is capable of estimating response times in real complex layered systems and it shows a high accuracy in experiments. Because our analytic method is based on the ML technique, there is a training phase and a test phase to use the embedded analyzer, which is not typical in conventional LQN analysis. This approach thus offers a novel contribution to the field of QoS assessment for layered systems.

The rest of the paper is organized as follows: Section II conducts a brief review on response time distribution analysis and prediction. Section III describes the concepts and related techniques that are used in this paper. Section IV presents how we apply MDN. Section V proposes the approach to estimating response times in a layered system. Section VI evaluates the performance of our MDNs and approach. Section VII concludes the paper and gives future research directions.

II. STATE OF THE ART

In closed queueing networks, the tagged customer method is commonly used to derive the response time distributions [3]. A typical method introduced in [1], [3] is to derive the Laplace–Stieltjes transform (LST) of passage time from start to end for the tagged job and decondition the start state to obtain the product-form LST of steady-state response time distribution. However, this method can only be applied when the service discipline is FCFS. In addition, its state-space analysis suffers from state explosion when the population is large. Fluid approach has been used successfully to approximate the response time distributions in closed queueing networks that consist of delay and queueing stations with processor sharing
(PS) discipline [4]. The discrete number of jobs at the station is approximated by a fluid continuous amount so that the state explosion no longer exists. Here, the fluid method is based on Kurtz’s theorem [5] which is hard to use in networks with multiclass FCFS stations. All the methods discussed above are limited to Markovian queueing networks. There are few analytic methods for the networks with non-Markovian service time distributions at the present moment. For what concerns layered queueing networks, they are a type of non product-form queueing network where the service time of one station may depend on other stations [6]. The existing technique approximates the variance of response time in a LQN by taking the sum of variance estimates of associated services given by analytic methods [7]. However, the results can be inaccurate in real complex systems. The limitations of conventional analysis have led to an interest in statistical learning methods which make it possible for us to capture the pattern under more realistic situations. Among these methods, MDN technique is well suited for approximating unknown conditional probability distribution. The MDN was first proposed in [8]. Recently, it has been widely used in distinct areas [9], [10], [11], [12], [13] and proved to be an effective predictor. In the QN area, [14], [15] have used MDNs to predict response time distributions in open queueing networks and service function chains.

III. BACKGROUND

A. Layered Queueing Networks

Fig. 1 shows a LQN model that contains all of the features considered in this paper. In LQN models, resources exposing a service are called tasks and shown as parallelograms. The tasks are executed by hardware resources called host processors, represented by the attached circles. The tasks and processors usually have FCFS and PS service discipline respectively. The stacked notation makes a task or a processor be a multiserver and the multiplicity is indicated in brace. For instance, T5 denotes a task with five threads and runs on P5 which could be a two-CPU multiprocessor. Distinct classes of service are called entries and represented by smaller parallelograms contained by tasks. Each entry is specified by a set of operations called activities shown as rectangles. The host demand of each activity is an independent exponential random variable and the mean duration is indicated in square bracket. For ease of presentation, we consider sequential activities in the paper. Our method can be easily generalized to non-sequential patterns.

A key feature of layered queueing models is that a service may include nested calls to other services. The calls model the service requests and they are represented by directed arrows going from one activity to an entry of another task. There are two main types of call: synchronous and asynchronous. The synchronous call is a blocking request during which the sender waits for the reply (dashed arrow), while the asynchronous call is a send-no-reply request—the execution continues after sending the call. In this paper, we focus on synchronous call, for example, it is the pattern of standard remote procedure calls (RPC) that exists in most layered queueing systems. The amount of calls indicated in parenthesis alongside the request arrow is either deterministic or stochastic, both cases are considered in our method. The amount of stochastic requests is assumed to be geometrically distributed and the number in parenthesis is the mean value. In Fig. 1, the tasks T1 and T2 do not receive any calls. They are system workload generators, or used to represent the end users, and named reference tasks. The think time of a reference task is denoted by Z. The interested readers are referred to the references [16], [17] for a more detailed description of the LQN models.

In the analysis of a LQN, it is decomposed into a set of submodels (ordinary queueing networks) that can be solved by product-form analysis techniques, such as approximate mean value analysis (AMVA) [18]. The solution iterates among all submodels until meeting certain convergence criteria and then we can obtain the mean performance metrics of the LQN [6]. SRVN-layering [19] and MOL-layering [20] are two ways to construct submodels. In this paper, we adopt SRVN-layering whose submodel consists of a delay and a queuing station.

B. Class Switching and Chains

In a closed queueing network with N stations and K job classes, the routing matrix $R = \{r_{iu,jv}\}$, $i, j = 1, \ldots, N$, $u, v = 1, \ldots, K$ defines a finite-state discrete-time Markov chain (DTMC). The station-class pair $(j, v)$ can be reached from pair $(i, u)$ with probability $r_{iu,jv}$. Jobs can switch class in the network if there exists $r_{iu,jv} \neq 0$ for $u \neq v$. All job classes can be divided into $0 < L \leq K$ disjoint sets named chains. Each chain is a set of classes that can switch jobs with each other, but not with classes in other chains. This means jobs inside a chain will never escape from it. Therefore, the number of jobs in each chain remains constant at all times. It is natural to think of transferring a closed queueing network with L chains to a closed queueing network with L independent job classes and measuring the chain performance [21].
methods have been developed to calculate the performance metrics per chain and then per class in each chain [22], [23]. A brief description is as follows. We first derive the population, number of visits at each station, and service time for each of the L chains. Next, the performance metrics per chain can be calculated by AMVA in the same way as for multiclass queueing network. Then, we use the population and throughput of each chain to derive the mean response time and throughput per class in that chain. Based on these two measures, all other metrics per class can be obtained in the end. We refer the interested readers to the reference [24, §7.3.6].

C. Phase-type Distribution and Closure Property

In this paper, we cope with various response time distributions in a layered system by phase-type (PH) distributions. PH distribution is defined as the distribution of time to absorption in a finite continuous-time Markov chain (CTMC) with an absorbing state. It is uniquely determined by a pair (α, T), where α and T are the initial probability vector and subgenerator matrix among the transient states of the CTMC. The PH distribution function and moments can be expressed in terms of α and T [25] as follows

\[ F(x) = 1 - \alpha e^{T x} \mathbf{1} \]
\[ E[X^j] = (-1)^j j! \alpha T^{-j} \mathbf{1}, \quad j = 1, 2, \ldots \]

where \( x \geq 0 \) and \( \mathbf{1} \) is a column vector of ones.

The closure properties of PH distribution are studied in [25]. They are important because these properties allow us to replace numerical representations with matrix ones in complex operations such as finite convolutions, maximum and finite mixtures, which make it possible for us to process probability distribution in an algebraic and tractable form. Here we cite one property relevant to this paper:

If \( X_i \) and \( X_j \) are statistically independent random variables with PH distributions \( (\alpha_i, T_i) \) and \( (\alpha_j, T_j) \), then \( X_i + X_j \) is a PH random variable with \( (\alpha, T) \), where

\[ \alpha = (\alpha_i, 1 - \alpha_i \mathbf{1} \alpha_j), \quad T = \begin{pmatrix} T_i & -T_i \mathbf{1} \alpha_j \\ 0 & T_j \end{pmatrix} \]  

PH distribution is an effective fitting tool in matching the first and second moments. Even the underlying distribution is known, it is preferable to fit a PH distribution to the first two moments instead of using the original distribution that may make it hard to incorporate into certain algebraic operations [26]. In summary, the PH fitting technique is always able to return a distribution \( (\alpha, T) \) with the same mean and variance. The interested readers are referred to the reference [27] for details of the fitting procedure.

IV. MIXTURE DENSITY NETWORKS

Mixture density network (MDN) is an elegant combined structure of a mixture model and a fully-connected neural network. We start from introducing the mixture model—a weighted sum of \( N \) individual probability density functions

\[ f(y) = \sum_{i=1}^{N} \pi_i p_i(y | \theta_i) \]

where \( \pi_i \) are mixing coefficients such that \( 0 \leq \pi_i \leq 1 \) and \( \sum_i \pi_i = 1 \), and \( \theta_i \) are parameters that characterize \( N \) kernel functions. Mixture model has the potential to express arbitrary probability distributions [8], thus we can use it to model various response time distributions in queueing networks. Given the LQN submodel information as an input vector \( x \), we wish to obtain its response time distribution expressed by a mixture model. In other words, our purpose is to build a conditional mixture density function of \( y \) given \( x \)

\[ f(y|x) = \sum_{i=1}^{N} \pi_i(x) p_i(y | \theta_i(x)) \]

where \( \pi_i(x) \) and \( \theta_i(x) \) are the mixture coefficients and parameters, respectively. The central idea of MDNs is to model both functions by a fully-connected neural network which has the flexible fitting capability for complicated relationships. Let \( O \) denote the number of observed samples, a MDN is trained by the dataset \( \{(x_j, y_j) \mid 1 \leq j \leq O\} \) and learns the weights of the neural network through maximizing the likelihood

\[ L = \prod_{j=1}^{O} f(y_j|x_j) \]

In MDNs, there are different types of kernel functions that can be used. The majority of applications choose Gaussian kernels as mixture components [9], [10], [11], [12], [14], [15],...
a few applications have ever used other kernels such as Gamma or Binomial [13]. In this paper, we use Gamma-MDNs to predict response time distributions in submodels. With the same number of components, a MDN with Gamma kernels provides a better performance compared to that with Gaussian kernels, especially for the models under the PS scheduling discipline. For instance, Fig. 2 shows a comparison between one histogram and two density functions obtained from simulation and five-component MDNs respectively. As can be seen from Fig. 2, both MDNs provide accurate predictions for the FCFS case, but for the PS case, the MDN with Gamma kernels provides a more accurate approximation.

The Gamma kernel $\gamma$ is a two-parameter ($\theta=\{\alpha, \beta\}$) probability density function with $\alpha > 0$, $\beta > 0$. Therefore, a Gamma-MDN needs to model three functional relationships $\alpha_i(\cdot)$, $\beta_i(\cdot)$, and $\pi_i(\cdot)$ to determine the conditional density function of variable of interest

$$f(y|x) = \sum_{i=1}^{N} \pi_i(x) \gamma_i(y|\alpha_i(x), \beta_i(x)) \quad (6)$$

Fig. 3 shows the structure of Gamma-MDN. The output layer consists of three types of nodes that return the function values $\pi_i(x)$, $\alpha_i(x)$, and $\beta_i(x)$. The first type employs softmax as an activation function to turn real values into mixing coefficients that sum to one. For the second and third types, the return values should be always positive. We employ the following activation function [12] to ensure this condition

$$g(x) = 1 + ELU(\xi, x) \quad (7)$$

where $ELU(\xi, x)$ is Exponential Linear Unit proposed by [28], and $\xi$ is a positive hyperparameter

$$ELU(\xi, x) = \begin{cases} 
\xi (e^x - 1) & x < 0 \\
\xi & x \geq 0
\end{cases} \quad (8)$$

This activation function is more stable compared to exponential activation function because the growth is not fast on the positive side. It helps to avoid instability during the training process [12]. The training makes the neural network learn its weights through minimizing our objective loss function that is defined as the negative log-likelihood.

V. METHODOLOGY

In this section, we propose an approach to estimate response time distributions in layered systems. We first study SRVN submodels with 1-2 job classes because later we will reduce submodels with >2 classes to this case. To illustrate the principle, four MDNs are trained by four distinct datasets: I. Single class, FCFS, II. Single class, PS, III. Two classes, FCFS, IV. Two classes, PS. Therefore, the input is $x = (N, Z, D, c^2, S)$ from I, II; and $x = (N_1, Z_1, D_1, c_1^2, N_2, Z_2, D_2, c_2^2, S)$ from III, IV. The definitions of input parameters are shown in Table I. When applying our methodology to a target layered system, we build a dataset by simulating randomly generated pairs in which the values of parameters $N, Z, D, c^2, S$ fall within the valid parameter ranges of the potential problems we are trying to solve, and train the embedded MDN analyzer. In this way, our approach possesses the generalization capabilities to analyze real complex layered systems.

A. Mechanism

As mentioned earlier, we use SRVN-layering to construct submodels. Each SRVN submodel can be characterized by a closed queueing network with a delay and a queueing station. The queueing station is either a processor or a task. According to the type of queueing station, we define two terms: processor-submodel or task-submodel. Our method starts from analyzing processor-submodels (Algorithm 1) and then performs convolution of the response time distributions layer by layer, from bottom to top (Algorithm 2). We give each submodel four main job classes: Task(T), Entry(E), Activity(A), and Call(C); the switching happens between each other within the closed network. In a layered system, the operations of each task are executed on the host processor. For this reason, in a processor-submodel we let queueing station serve A-class jobs and delay station serve the rest of the classes. Meanwhile, a task may send requests to the services exposed by other tasks. Therefore, in a task-submodel we let queueing station serve C-class jobs and delay station serve the rest. As the example shown in Fig. 4, the switching circle for processor-submodel B is $(Delay, T) \rightarrow (Delay, E) \rightarrow (Queueing, A) \rightarrow (Delay, C) \rightarrow (Delay, T)$, while the switching circle for task-submodel A is $(Delay, T) \rightarrow (Delay, E) \rightarrow (Delay, A) \rightarrow (Queueing, C) \rightarrow (Delay, T)$. We first obtain mean performance metrics for each submodel through the iterations [6]. Next we use the technique introduced in Section III-B to calculate the metrics for each job class (lines 1-3 in Algorithm 1). In this step we get think times of classes $T$, $A$, and $C$ named $Z_T$, $Z_A$, and $Z_C$. The think
times of processor-submodel and task-submodel are \( Z_T + Z_C \) and \( Z_T + Z_C + Z_A \) (\( Z_C \) is zero if there is no call sent outside the submodel) respectively. After this we fit \( \Phi \) to the service time distribution of each activity and then convolve all associated \( \Phi \) distributions for each entry (line 8 in Algorithm 1). Then we feed submodel information to a Gamma-MDN and get a mixture distribution with mean and variance

\[
\mu = \sum_{i=1}^{N} \pi_i \mu_i, \quad \sigma^2 = \sum_{i=1}^{N} \pi_i \left( \sigma_i^2 - \mu_i^2 \right)
\]

where \( \pi_i, \mu_i, \) and \( \sigma_i^2 \) are the kernel parameters which are returned by this MDN. We therefore can calculate the mean and variance of the obtained mixture model by (9). Next, we begin analyzing bottom task-submodels (line 4 in Algorithm 2). We define this term as a task-submodel whose server has a known response time distribution. For instance, in Fig. 4 the submodel \( A \) becomes a bottom task-submodel after calculating the processor-submodel \( C \) because Gamma-MDN gives us the known response time distribution of \( E_2 \), the server of submodel \( A \). Let \( \Phi \) and \( w \) denote the response time of server and the number of requests to server, we define a new variable \( \delta = \psi w \). Assuming \( \Phi \) and \( w \) are independent random variables, we can derive mean and variance of \( \delta \)

\[
\mu_\delta = \mu_\Phi \mu_w, \quad \sigma_\delta^2 = \sigma_\Phi^2 \sigma_w^2 + \sigma_\Phi^2 \mu_w^2 + \sigma_w^2 \mu_\Phi^2
\]

If the request is deterministic, \( \Phi \) should be an integer with \( \sigma_\Phi^2 = 0 \). If it is stochastic, \( \Phi \) will be a geometric random variable with \( \mu_\Phi = (1 - p)/p \) and \( \sigma_\Phi^2 = (1 - p)/p^2 \). Thus

\[
\sigma_\delta^2 = \begin{cases} 
\sigma_w^2 \mu_\Phi^2 & \text{deterministic} \\
\sigma_w^2 \left( p^2 - 3p + 2 \right) + \mu_w^2 \left( 1 - p \right) & \text{stochastic}
\end{cases}
\]

(11)

We use \((\mu_\delta, \sigma_\delta^2)\) to characterize the service time distribution in a bottom task-submodel. Then we use a Gamma-MDN to forecast the response time distribution and fit \( \Phi \) to this prediction. One entry may have several servers that execute its requests in distinct task-submodels. Once we complete such analysis for each of these, we convolve all of the fitted distributions in its host processor and task-submodels and finally get the response time distribution of this entry (lines 16-18 in Algorithm 2). We repeat this procedure until we obtain the response time distribution for every entry.

### B. Class Aggregation

A submodel may have multiple chains. In this case, we use a two-class MDN to predict response time distribution for each chain. Suppose there are \( n + 1 \) chains in the model and we are interested in the response time of a certain chain, then we need to aggregate the remaining \( n \) chains so that a two-class MDN can be used. We employ the following aggregation method inspired by the utilization law

\[
N_{agg} = \sum_{i=1}^{n} N_i, \quad M_{agg} = \sum_{i=1}^{n} X_i M_i / \sum_{i=1}^{n} X_i
\]

(12)

where \( N_i \) and \( X_i \) are the population and throughput of the \( i \)-th chain, \( M_i \) can be \( Z, D, \) or \( c^2 \) that is defined in Table I.

Fig. 5 shows a special case that we need to consider. The task \( T \) has \( K \) entries and they receive requests from the upper layer. In this case, we model the processor-submodel by a closed network where \( T \) switches to \( E_1, E_2, \ldots, E_K \) with probabilities \( p_1, p_2, \ldots, p_K \). The probability is determined by the throughput of upper layer, that is \( p_k = X_k^{upper} / X_{total}^{upper} \). As the Gamma-MDN method requires constant multiclass populations, we need to apply the following transformation to the model

\[
N_k = \left( \frac{X_{E_k}}{X_{total}} \right) N_T + N_{E_k}
\]

(13)

where \( N_k \) is the population of the \( k \)-th chain, \( X_{total}, X_{E_k}, N_T \) and \( N_{E_k} \) are the mean performance metrics obtained at the beginning of Algorithm 1.

### VI. Evaluation

In our approach, the key to estimate response times in a layered system is to obtain response time distributions in each closed queuing submodel. Therefore, we first evaluate if Gamma-MDNs are capable of providing good predictions for unknown response time distributions. We generate training
Algorithm 1

Input: A LQN model
Output: De for each e
1: construct P and T, create empty De for every e
2: derive the parameters of chains for each s
3: iterate to solve LQN, obtain the mean performance metrics per chain in s and then per class in each chain
4: for each s ∈ P do
5:  Z = Z_T(t) + Z_C(t)
6:  create a set C such that |C| = |ε(t)|, and determine the population of each c by (13)
7:  for each e ∈ ε(t) do
8:      convolve Ð(e) by repeated execution of (2)
9:      extract mean and SCV from the convolution result by (1), build the input x
10:  end for
11: for each c ∈ C do
12:    if |C| > 1 then
13:      aggregate the chains by (12)
14:      two-class input x with Û(PS) MDNIV W
15:    else
16:      single class input x with Û(PS) MDNI W
17:    end if
18:  extract mean and SCV from W by (9)
19:  W ← fitPH(mean, SCV)
20:  put W into Ð_e
21: end for
22: end for

and test pairs falling within the ranges shown in Table I and employ the tools LINE 2.0.8 [29] with JMT 1.0.5 [30] to simulate the pairs under FCFS and PS discipline respectively. For each pair, we tag a particular job and measure its response time at the queueing station. Because the network is closed, the measurements can be made constantly so that we obtain a steady response time distribution. The single class and two-class training sets consist of 22000 and 14000 pairs and the measurements can be made constantly so that we obtain a steady response time distribution. The single class and two-class training sets consist of 22000 and 14000 pairs and the evaluations are conducted on 4400 and 2800 unseen pairs.

We consider two performance measures: mean absolute percentage error (MAPE) and mean KS distance (MKS). The first measure is defined as

\[
\text{MAPE} = 100\% \sum_{n=1}^{N_{\text{test}}} \frac{P_n - G_n}{G_n}
\]  

(14)

where P and G are the predicted and ground-truth values, N_{\text{test}} is the number of test pairs. We calculate MAPE for mean, SCV, and 95th percentile of response time respectively.

KS distance is defined as the maximum vertical distance between two cumulative density functions (CDFs), thus our second measure is

\[
\text{MKS} = \frac{1}{N_{\text{test}}} \sum_{n=1}^{N_{\text{test}}} \max_x \left| F_n^G(x) - F_n^P(x) \right|
\]

(15)

The datasets are available at https://zenodo.org/record/5528034

Algorithm 2

Input: Output of Algorithm 1
Output: Re of LQN

1: Done ← empty set
2: while |Done| < |T| do
3:   for each s ∈ T \ Done do
4:     if s ∈ B then
5:       obtain the set C
6:     end if
7:     for each c ∈ C do
8:       if |C| > 1 then
9:         aggregate the chains by (12)
10:        two-class input x with Û(FCFS) MDNIII W
11:      else
12:        single class input x with Û(FCFS) MDNI W
13:      end if
14:     end for
15:     implement line 18-20 of Algorithm 1
16:     if |D_e| = |E_e| + 1 then
17:       convolve D_e by repeated execution of (2)
18:       Re_e ← convolution result
19:     end if
20:     end for
21: end while

<table>
<thead>
<tr>
<th>Model</th>
<th>Training pairs</th>
<th>Test pairs</th>
<th>MAPE Mean</th>
<th>SCV 95-th</th>
<th>MKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDN I</td>
<td>22000</td>
<td>4400</td>
<td>3.7%</td>
<td>5.5%</td>
<td>4.4%</td>
</tr>
<tr>
<td>MDN II</td>
<td>22000</td>
<td>4400</td>
<td>4.9%</td>
<td>6.5%</td>
<td>5.7%</td>
</tr>
<tr>
<td>MDN III</td>
<td>14000</td>
<td>2800</td>
<td>3.3%</td>
<td>7.0%</td>
<td>5.6%</td>
</tr>
<tr>
<td>MDN IV</td>
<td>14000</td>
<td>2800</td>
<td>6.0%</td>
<td>7.6%</td>
<td>5.7%</td>
</tr>
</tbody>
</table>

where F^G(x) is the empirical CDF from observed samples and F^P(x) is the CDF of a mixture model from prediction.

Each of our MDN models consists of seven Gamma kernels and the evaluation results are shown in Table III. As can be observed, all MAPE values are below 10% and MKS values are less than 0.05, the trained MDNs provide overall good performance. Fig. 6a shows an interesting phenomenon that the vast majority of KS distances are below 0.1, which means the MDNs really learned some relationships and try to get as close to the true response time distributions as they can. Fig. 6b gives a comparison between simulations and sampling distributions from MDN predictions. In this experiment, we simulate the pair and obtain 1169 and 1170 response time samples for cases FCFS and PS. Then we employ MDN I and II to make predictions and generate 1169 and 1170 samples from the returned Gamma mixture distributions. As can be seen from this figure, the MDNs cope well with learning to reproduce unknown distributions. The emulated distribution
Fig. 6. Visualizing prediction performance with box plots (the orange line and green triangle inside box express the median and mean values). (a): Summary statistics on the KS distance for all test pairs of Table III. (b): Summary statistics on the response time given $x = (85, 8, 0.4, 1.5, 2)$. The left and right y-axis represent the response time under FCFS and PS respectively.

Fig. 7. A larger interconnected layered system.

captures a similar symmetric-like or right-skewed trend as the simulation. The next experiment is to compare mean and variance between true distributions and predictions through varying a set of input parameters [13]. Fig. 8a to 8f show a “what if” instance where we can observe that our MDNs give acceptable approximations for mean and variance over the change of distinct types of parameters. This means MDNs are capable of discriminating variables and calculating how each type influences the response time. Fig. 8g and 8h describe the response time distributions in a submodel having two job classes with different service rates. This is the situation where conventional BCMP theorem cannot be used [2]. We can see that the true distribution of class 1 is very well predicted by the MDN. The prediction for class 2 is worse than class 1 (the SCV percentage error for class 2 is greater than 10%), but the overall trend of prediction is still reasonable.

Then, our algorithms are evaluated on the LQN model in Fig. 1 and a larger interconnected layered system in Fig. 7. The parameters of the second system are shown in Table V. In this layered system, the requests made by reference tasks $R_1$, $R_2$, $R_3$, $R_4$ are stochastic with a mean value 0.5 except for the requests $R_3 \rightarrow T_6$, $R_4 \rightarrow T_7$, the mean for these two is 1. The remaining requests made by non-reference tasks are deterministic with a value 1. We calculate the end-to-end delays for both two models and compare the results with lqsim and lqns 6.0 [17], a simulation tool and a de facto standard analytic tool for LQN. As shown in Table IV, the response times of entries in reference tasks are well predicted by our approach. The MATLAB implementation of the algorithms takes 0.96s (including 27 predictions and 9 convolutions) on the second model after getting the mean values by AMVA. The percentage errors (PE) for SCV and mean are less than 10% and 1% respectively. This illustrates the effectiveness of our methodology which serves as a potential solution to response time distribution analysis in layered systems.

VII. Conclusion

In this paper, we have developed an approach to estimate the distributions of response time in a layered system. The approach copes with a set of carefully built submodels and performs convolution of the response time distributions layer by layer, from bottom to up. In this process, Gamma-MDNs predict the conditional response time distribution for each submodel. They have the promising potential to be used as emulators for complex queueing studies. In the end, the mean and variance of system delay are obtained and we employ fitted PH as the approximated response time distribution. The results give users a detailed QoS description upon single mean values. It also allows us to calculate the probability that a customer will get a response in less than a specific time, which can be used to provide service promises in the future. In future work we plan to extend this approach to include other details of layered systems such as asynchronous calls.

REFERENCES


<table>
<thead>
<tr>
<th>Task</th>
<th>T1</th>
<th>T2</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>lqns 1.073</td>
<td>3.135</td>
<td>1.492</td>
<td>0.385</td>
<td>0.038</td>
<td>0.909</td>
</tr>
<tr>
<td>SCV</td>
<td>lqns 1.145</td>
<td>3.221</td>
<td>1.502</td>
<td>0.394</td>
<td>0.067</td>
<td>0.917</td>
</tr>
<tr>
<td>Mean</td>
<td>lqns 1.139</td>
<td>3.241</td>
<td>1.506</td>
<td>0.495</td>
<td>0.063</td>
<td>0.918</td>
</tr>
<tr>
<td>SCV</td>
<td>lqns 1.321</td>
<td>1.126</td>
<td>0.620</td>
<td>1.492</td>
<td>0.930</td>
<td>1.126</td>
</tr>
<tr>
<td>PE</td>
<td>lqns 1.356</td>
<td>0.747</td>
<td>0.689</td>
<td>1.717</td>
<td>1.838</td>
<td>1.126</td>
</tr>
<tr>
<td>SCV</td>
<td>lqns 1.390</td>
<td>0.702</td>
<td>0.740</td>
<td>1.805</td>
<td>1.852</td>
<td>1.066</td>
</tr>
<tr>
<td>PE</td>
<td>lqns 5.797</td>
<td>3.277</td>
<td>0.933</td>
<td>2.020</td>
<td>0.753</td>
<td>0.985</td>
</tr>
<tr>
<td>SCV</td>
<td>lqns 0.533</td>
<td>0.627</td>
<td>0.272</td>
<td>0.200</td>
<td>0.000</td>
<td>0.117</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Task</th>
<th>T1</th>
<th>T2</th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td>lqns 0.667</td>
<td>0.740</td>
<td>0.667</td>
<td>0.740</td>
<td>0.667</td>
<td>0.667</td>
</tr>
<tr>
<td>SCV</td>
<td>lqns 0.767</td>
<td>0.667</td>
<td>0.740</td>
<td>0.667</td>
<td>0.667</td>
<td>0.667</td>
</tr>
</tbody>
</table>

TABLE IV

PERCENTAGE ERROR OF MEAN AND SCV
Fig. 8. (a) to (f): Starting with $x = (90, 8, 0.9, 0.5, 8)$ and changing SCV, number of servers, population over a range of values, comparing the mean (simulation = heavy dashed line, prediction = colored line) and variance (simulation = light dashed line, prediction = colored region) between the true distributions and predictions. (g): (h): Response time distribution functions of two job classes in the submodel $x = (40, 3.5, 0.6, 0.5, 55, 2, 1, 1, 8)$ with FCFS service discipline.

### TABLE V
PARAMETERS OF THE LAYERED SYSTEM IN FIG. 7

<table>
<thead>
<tr>
<th></th>
<th>R1</th>
<th>R2</th>
<th>R3</th>
<th>R4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
<th>T9</th>
<th>T10</th>
<th>T11</th>
<th>T12</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplicity (Task)</td>
<td>30</td>
<td>55</td>
<td>70</td>
<td>40</td>
<td>10</td>
<td>15</td>
<td>10</td>
<td>20</td>
<td>30</td>
<td>25</td>
<td>25</td>
<td>28</td>
</tr>
<tr>
<td>Multiplicity (Processor)</td>
<td>16</td>
<td>16</td>
<td>4</td>
<td>4</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>16</td>
<td>16</td>
<td>20</td>
<td>20</td>
<td>28</td>
</tr>
<tr>
<td>Host Demand Mean</td>
<td>0.83</td>
<td>0.12</td>
<td>0.05</td>
<td>0.29</td>
<td>0.05</td>
<td>0.17</td>
<td>0.26</td>
<td>0.33</td>
<td>0.15</td>
<td>0.34</td>
<td>0.1</td>
<td>0.2</td>
</tr>
</tbody>
</table>
Automated performance prediction of microservice applications using simulation

Clément Courageux-Sudan*, Anne-Cécile Orgerie*, Martin Quinson*
*Univ. Rennes, Inria, CNRS, IRISA, France
Email: {clement.courageux-sudan, anne-cecile.orgerie, martin.quinson}@irisa.fr

Abstract—Microservices transform monolithic applications into simple, scalable, and interacting services. It allows for faster development and fine-grained deployments. However, the cooperation of several services leads to intricate dependencies, hindering the detection of performance bottlenecks. Current microservice performance analysis methods require real deployments, a costly process both in time and resources, while performance prediction through simulation relies on models that are complex to develop and instantiate. In this paper, we propose a microservice performance analysis approach based on simulation. Our contribution first introduces a microservice performance model requiring few instantiation parameters. We then propose a methodology to automatically derive model instantiation values from a single execution trace. We evaluate this methodology on two benchmarks from the literature. Our approach accurately predicts the deployment performance of large-scale microservice applications in various configurations from a single execution trace. This provides valuable insights on the performance of an application prior to its deployment on real platform.

Index Terms—Microservice, Modeling and simulation, Performance evaluation, Web services

I. INTRODUCTION

Microservices allow for splitting large and complex applications into sets of simple, independent, and scalable services. Microservice architecture has many advantages over monolithic applications: separation of concerns allows disjoint teams to develop independently part of an application without complete system knowledge. When deployed, microservices run in separate lightweight containers, allowing for better reliability and adaptation to variable workloads thanks to fault detection mechanisms and autoscaling policies at the scale of individual services. Compared to microservices, monolithic applications have to be managed as a whole, reducing their scalability. As a result, microservices are used by some of the largest internet actors, such as Twitter, Netflix, and Uber [21].

Despite their advantages, microservices require complex interactions to fulfill requests. This makes the analysis of microservice applications on real infrastructures more complex than monolithic applications. Microservice applications can be composed of hundreds of services and serve huge workloads. Because of this complexity, optimizing the deployment settings of microservices is challenging: Given an application and a constrained infrastructure budget, one must evaluate and answer the following questions to obtain the best performances: Q1. How will the execution times of a microservice react to a variable load? Q2. How would a CPU upgrade improve the maximum sustainable load? Q3. Will distributing microservices on more than one node increase performances? Q4. How to optimize the location of services in a computing cluster to obtain the best performances?

Currently, microservice developers answer such questions by deploying their applications on real platforms and monitoring their performances. This requires a combination of several tools. First, the application is deployed using container orchestration systems, such as Kubernetes or Docker-swarm. Using them allows defining resource and location constraints on services, so that the experimenter can deploy the application in a specific configuration. The deployed application then has to be benchmarked to obtain information on its performance in this setup. Gathering and processing applications metrics can be done through the use of Distributed Tracing. This approach is used in most microservice applications [16], allowing to obtain both service-specific and end-to-end execution metrics. Standards such as OpenTracing [20] and OpenTelemetry [19] help developers at instrumenting their application, while trace visualization tools like Jaeger [14] and ZipKin [26] unify the process of observing application bottlenecks.

Using orchestration systems and distributed tracing tools, microservice developers can observe whether a set of deployment settings meets their QoS requirements. However, this approach requires combining complex tools, a costly process both in time and resources. Also, the process must be repeated for all configurations. Obtaining a priori performance estimations for various configurations without real deployments would be useful. In this paper, we focus on simulation techniques adapted to microservice performance studies. Our analysis of previous contributions about microservice simulation shows a costly process to transpose applications into their simulated twins. There is a need for a methodology to ease performance simulation of real applications.

Our contributions are the following:

- We propose a model for microservice-based applications. This model is simple enough to be calibrated without extensive work while being complete enough to simulate real applications.
- We propose a methodology to instantiate our model and accurately study the performances of real applications.
- We validate the accuracy and scalability of our approach using a set of microbenchmarks as well as real use cases based on existing microservice benchmarks.

Section II introduces our microservice abstraction. Section III describes a methodology to simulate real applications.
by using our service model. We validate our contribution in Section IV while Section V compares our contribution to other approaches from the literature. Finally, Section VI concludes this work.

II. A SIMPLE MICROSERVICE MODEL

Modern applications are composed of many services interacting together to fulfill requests. To understand the performance of a complete microservice application, one must first understand the behavior of single, isolated microservices. A microservice offers a well-defined interface, constantly listening for incoming requests. When a request is received, it triggers internal functions computing a result that is then returned to the initiator of the request.

![Diagram of intra-service execution pipeline.](image)

We introduce a model representing microservice request executions. Our design goal for this model is to be as simple as possible to enable fast calibrations while being accurate enough to represent real microservices. We model the execution of requests within a microservice using a simple 3-stages pipeline, as shown in Figure 1. A request executed by a service spends some time in each execution stage depending on both the state of the service when the request is received (queuing, resource overload, etc.), and intrinsic service properties such as its degree of parallelism. The total execution time of request $r$ in service $s$ is given by:

$$D_{exec}(r, s) = d_{queue}(r) + d_{CPU}(r) + d_{IO}(r) + d_{comm}(r)$$

where $d_{queue}(r)$ is the queuing time of request $r$ before starting its execution, $d_{CPU}(r)$ is the CPU time to execute request $r$, $d_{IO}(r)$ the amount of time waiting for I/O operations, and $d_{comm}(r)$ the time spent communicating with external services. We now detail these factors.

(a) Queuing time: When a request is sent to a service, it can experience variable queuing times depending on the service’s state. Most services limit their maximum amount of concurrent requests to avoid resource overloads and performance degradation due to system’s context switches. In our model, a service comprises a waiting queue where incoming requests are stored until their execution starts. The time spent by a request in the reception queue is dependent on both the input load and the scale of the deployed application. Through vertical scaling, the service is deployed on more efficient resources, leading to reduced execution times during step (b.1), and an increased maximum load capacity. Horizontal scaling does not improve the execution time of single requests, but allows for more requests to be executed in parallel through the use of several service instances in parallel during step (b).

(b.1) CPU usage: A request can start being executed once a free execution slot is available in an instance of the service. We describe the execution of a request by its CPU usage: the machine executing the request has a limited amount of CPU resources (in flops/s) and shares them between all active requests in one of the execution slots of the host. The time spent processing a request $d_{CPU}(r)$ is dependent on both the cost of executing $r$ and the amount of requests that are executed concurrently. The cost associated to a request execution further depends on its type. A single microservice can offer more than one function through its interface, each of them leading to different execution times. Because of this, the CPU usage of a service is described by the following values: the provisioned capacity of the executing node, a mapping of request type to CPU costs, and the maximum amount of concurrent executions.

(b.2) I/O idle time: A service execution does not only consist in CPU processing but also in I/O operations. In some cases, I/O can be overlapped with CPU executions. In other cases, I/O can result in periods where the CPU remains IDLE. We define the time spent in I/O by using an active ratio that represents the time spent doing I/O compared to pure CPU execution. Currently, we model I/O as a simple delay, thus we do not take disk contention into account.

(c) Service output: Most microservices request data from other services to compute a final result. Using our model, a request can be forwarded to other services once it has finished executing the request during (b.1) and waiting for I/O to finish in (b.2). The type of the request defines the list of services to be invoked. If the output services called during this step are running on different computing nodes than the current service, a network communication is initiated to forward the request. This enables the observation of performance bottlenecks due to network limitations. Microservices linked through inter-service communications form a Directed Acyclic Graph (DAG) for each request type, where nodes are service executions and edges communications between services. As shown in Figure 1, we separate the execution of a request from outer communications. It is a simplification compared to real microservices that often interleave executions and communications. Our approach does not represent the execution of a request at such a fine grain but conserves the overall execution time as well as communication dependencies.

This model splits the execution of a microservice request in no more than the 3 steps described above. Table I gives a summary of the values used to instantiate our model.

We implemented our model on top of the SimGrid simulation framework. Our implementation (available online) allows to define a microservice application using a simple interface with the parameters from Table I. It can be used to manually define real applications in order to study and evaluate their performances in various configurations. It is also possible to define autoscaling policies, dynamically adapting the amount of resource of each service depending on the

---

1https://github.com/klementc/microservices_simgrid_reproducibility
TABLE I
SUMMARY OF INTER-SERVICE AND INTRA-SERVICE PROPERTIES USED TO CALIBRATE OUR MICROSERVICE MODEL.

<table>
<thead>
<tr>
<th>Granularity</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Request</td>
<td>- Type of the request</td>
</tr>
<tr>
<td></td>
<td>- CPU costs</td>
</tr>
<tr>
<td>Intra-service properties</td>
<td>- I/O ratios</td>
</tr>
<tr>
<td></td>
<td>- Parallelization degree</td>
</tr>
<tr>
<td></td>
<td>- Output services (a DAG for each type of request)</td>
</tr>
<tr>
<td></td>
<td>- Network requests sizes</td>
</tr>
</tbody>
</table>

input load. Even if possible, the manual instantiation of the models for all services of large applications with the correct values remains a challenging task. Both service interactions and individual service behaviors need to be taken into account for possibly hundreds of services. We thus introduce a methodology to simplify the transformation of real applications into simulation models.

III. MODELING REAL MICROSERVICE APPLICATIONS

We propose a methodology to simulate the performance of real microservice applications without extensive efforts. Previous works on microservice simulation such as [25] recommend modifying the code of the application to log additional information that are used as calibration values. However, manual code modification remains burdensome and error-prone for large applications. It is also not evolutive since it has to be made again when the structure of the code is modified. Our goal is to allow developers of microservice applications using state-of-the-art service monitoring techniques to automatically obtain both the description of the structure of their application along with the calibration values for our microservice model. Our approach does not require code modifications provided that the targeted application uses one of the standard service monitoring solutions, as detailed hereafter.

To instantiate our microservice model, we need to gather values for the parameters given in Table I. The request types to study depend on the application and on the goal of the experiment. One can study either a specific request or observe several types of request executable in the application.

Intra-service properties need to be observed at both application- and system-levels. Indeed, overall service execution times can be observed easily from the application, but the ratio of active and IDLE CPU times requires lower-level information.

Inter-service information can be obtained by observing the network interactions between services running in separate containers.

The required accuracy for the calibration values depends on the target of the simulation. For example, approximate values of network request sizes can be adequate for an experiment studying the performance of an application under limited computing resources, as the network does not constitute the bottleneck of the experiment.

The model parameters can be obtained via various methods. From our experience, distributed tracing can provide these values at a low cost. Most real microservice applications are already instrumented to export metrics and information about the state of each of their services. Whereas the typical use case of distributed tracing is to help identifying the services inducing performance issues, leveraging it to calibrate our model is an interesting solution. Traces entails the path followed by requests and the amount of time spent in each service. Each service execution is called a span, and the set of all spans linked to the same request forms the DAG of the overall execution. The collected data provide both inter-service property DAGs and intra-service execution times. In the following, we use the information contained in these traces to instantiate our microservice models.

As summarized in Figure 2, our goal is to build a simulator exploiting the information contained in a single execution trace per type of studied request. The resulting simulator can then predict the performance of the application under various configurations. Figures 3 and 4 show an execution trace automatically extracted from an application of the literature [9]. In Figure 3, a partial execution trace can be observed, directly taken from the web interface of Jaeger [14], whereas Figure 4 is a DAG representation of the same request. Based on this example, we now show how to calibrate our model by following the steps of Figure 2.

Step 1: Obtain calibration traces. The application modeler first needs to gather execution traces of the requests to study. These traces are created upon request executions, once the application is instrumented with a tracing system such as OpenTracing [20], OpenTelemetry [19], or Kieker [24]. The user can then run the instrumented application on a machine.
representative of the targeted platform. Finally, the traces are used to calibrate our model. They contain information that will be used in the following steps. A particular care must be taken in the selection of the traces to ensure that they are representative of the behavior of the application. Using average execution times helps at avoiding outliers. In the case of Figure 3, the application from DeathStarBench [9] is natively instrumented with OpenTracing and we run it with a medium workload for several minutes to ensure that we select a trace corresponding to the application’s behavior in steady-state: none of the services are overloaded, the caches are warm, databases are populated, etc.

Step 2: Process the traces to obtain calibration values.

The second step consists in obtaining the values required for the instantiation of our microservice model. Tracing systems used in Step 1 are usually combined with generic trace processing tools such as Jaeger [14], Zipkin [26], or Kieker’s tools [24]. These tools collect application traces and process them to obtain metrics about the performance of end-to-end executions. They provide enough information to calibrate our model even if low-level metrics may be missing. As shown in Figure 4, using Jaeger, we obtain from the OpenTracing trace a DAG of services execution connected to each other. Thus, we can infer how services interact during request executions. The size of network requests is not provided in this case, but an instrumentation with OpenTelemetry [19] could for example allow us to send request sizes to Prometheus [22]. Intra-service properties are also partially included in our trace. We obtain an accurate estimation of the duration of the request for each service execution. Still, our example trace must be extended regarding the ratio of I/O. Complementary solutions such as Docker monitoring could be used to retrieve I/O ratios.

Step 3: Building and configuring the simulator.

The values obtained during the previous step allow instantiating one microservice model for each service of the target application. We can then observe the duration of end-to-end request executions through simulation. Simulations can be configured to explore the performance of the application under various deployment settings. Because obtaining calibration values is relatively easy thanks to existing monitoring systems, it becomes possible to adjust calibration values after the modification of a service’s implementation.

To profit from the ability of distributed tracing at providing the intra and inter-service properties of an application, we designed a tool to automatically produce the code of a simulator with calibrated microservices given Jaeger traces. This allows for a semi-automatic performance simulation of compatible instrumented applications.

IV. Experimental Validation

To evaluate the accuracy and scalability of our contribution, we rely on both a set of microbenchmarks and published microservice benchmarks use cases. We compare the performance predicted by models instantiated using our approach to real application executions. Based on those results, we show that our contribution can help microservice performance analysis by answering the four questions we stated in Section I.

Experimental setup: All experiments have been launched on the Grid’5000 testbed [2]. We run our experiments on a cluster composed of nodes with 2 × Intel Xeon E5-2630 v3 with 8 cores/CPU, 128 GiB of memory, and 2 × 10 Gbps network interfaces. They run Debian 10 under kernel 4.19.0-16-amd64. Services run within Docker containers and multi-node deployments are done with Docker-swarm.

Source code and reproducibility: The source code of our contribution is available online at https://github.com/klementc/microservices_simgrid_reproducibility along with the scripts used to obtain the results. We also provide notebooks with the code used to generate the figures of this paper and additional experimental results not presented here.

A. Microbenchmarks

Before predicting the performance of real large microservice applications, we need to ensure that our microservice model allows for accurate execution time predictions at the scale of single services. This first experiment aims at showing the ability of our microservice model to reproduce request execution times accurately under a dynamic load.

We launch a microservice application that executes a fixed amount of CPU work for each request. The microservice fetches incoming requests through a RabbitMQ queue, and it is possible to chain multiple services to obtain multi-steps executions. The results detailed hereafter consist of a
single service sending its results directly to a sink. We ran microbenchmarks using 2 chained services leading to similar results but not shown here due to space constraints.

To simulate this application using our model, we proceed in two steps. First, we obtain the execution time of a request given its CPU cost by sending calibration requests. These calibration values represent the execution duration when a service is not overloaded. Through linear regression, we estimate the duration of request executions for any CPU cost assigned to the service. We then calibrate the service model to use these values before running simulations and comparing the results to an execution on a real platform.

We generate a synthetic load using LIMBO, an HTTP load model and generator [23]. It consists of requests spanning over 5 minutes with between 1 and 40 requests per second and three activity spikes for a total amount of around 4.500 requests as shown in Figure 5. We launch this experiment 5 times for each configuration. Configurations vary by the quantity of work to be executed and the maximum amount of parallel executions.

Figure 6 shows the results obtained during the microbenchmark execution with one service, an execution time of approximately 25ms per request and a concurrency degree of either five or ten. The concurrency degree refers to the maximum number of parallel requests for a given service. For each request executed during the experiment, Figure 6 shows the estimated execution time of request obtained using our model in SimGrid and the execution times of a real deployment.

We make two observations. First, both SimGrid predictions and real-world results have higher execution times during request arrival spikes, which happen at 20s, 150s, and 250s. This is caused by the processing of several requests in parallel and queuing. This shows the ability of our service model to reproduce processing times of requests under a dynamic load.

Second, we observe that the execution time per request changes with the concurrency degree of the application. With a concurrency degree of 5, the maximum request execution time (125ms) is approximately two times lower than the maximum request execution time with 10 parallel requests. A smaller concurrency degree decreases single requests execution times at the cost of increased queuing times. Executing an important amount of parallel requests on a single CPU core will also lead to overheads due to operating system thread switching. The execution model of SimGrid does not take into account context switching costs, thus it might underestimate execution times when the number of parallel requests is much higher than the number of CPU cores to execute them. Yet, in the results of this paper, this effect does not impact our observations.

These results show that our microservice model implementation in SimGrid can accurately predict the performance of simple CPU-intensive microservices under variable loads, and thus it answers the first question:

**Q1. How will the execution times of a microservice react to a variable load?**

**B. Use-case 1: TeaStore login requests**

We now observe the performance when modifying the resources (i.e., number of cores) dedicated to the execution of an application within a single computing node. This question is of importance for real deployments to estimate the cost/gain ratio of different hardware options. Our goal is to obtain the same results between real observations and our simulations when changing the number of resources to be used. To evaluate the versatility of our approach, we only rely on one calibration experiment detailed hereafter to instantiate our service models.

For this evaluation, we run TeaStore [8], a microservice application benchmark used in microservice performance evaluation literature [11]. We focus on the LOGIN request of this application. This request involves 4 different services running in separate Docker containers. We study the maximum sustainable load (in Requests Per Second, RPS) of the application under different resource configurations and compare real results to simulated predictions.

TeaStore is natively instrumented with Kieker [24]. We use the average request execution duration of each service to calibrate our service models within SimGrid by doing a real execution under a low load-profile. From this execution we extract a trace with Kieker, providing us the calibration values.
As recommended by TeaStore’s documentation, the load is generated by LIMBO [23]. We benchmark the application under 3 different configurations. For the real experiments, the application is deployed on a machine with either 4, 8, or 16 cores dedicated to the execution of the services. We execute each configuration 20 times. A summary of the results is shown in Figure 7.

The goal of this experiment is to detect the breaking point after which the application is overloaded. This breaking point is detected by our model, for example around 320 requests per second in the 4 core configuration, as shown in Figure 7. Regarding the accuracy of our model, we observe the mean relative error (MRE) between SimGrid predictions and real world values. Over the different workloads shown in Figure 7, we observe an average MRE of 11.8%, 4.9% and 3.6% with maximum values of 21%, 17% and 14.9% in the 4, 8 and 16 cores configurations respectively. While the maximum error observed is non-negligible, especially in the 4 cores configuration, the predictions of our model allow observing trends, and comparing the advantages of one configuration over another. This experiment allowed us to answer the second question:

**Q2.** How would a CPU upgrade improve the maximum sustainable load?

C. Use-case 2: DeathStarBench’s social network

The next step consists in evaluating the performance of an application taking advantage of several physical servers, one of the main assets of the microservice architecture. Yet, finding the best partitioning of services is a complex task.

We chose to study one of the most realistic published microservice benchmarks to our knowledge, the social network from DeathStarBench [9]. We reproduce the most complex request of this application, the COMPOSE request that submits a publication to the social network. It consists of more than 30 spans across 12 different services. We deploy the social network using Docker-swarm, and vary the location of each service and the number of replicas. We compare the maximum sustainable request throughput obtained with our simulator against real executions with 10 real runs per configuration.

Table II shows the server allocations for the microservices required to execute the COMPOSE request in each of the three studied configurations. With configuration 1, all services are using the resources of a single node. It should be the least efficient configuration due to fewer resources available for each service. Configurations 2.a and 2.b divide the 12 services into two randomly selected groups, each running on different nodes. For all configurations, other services of the application, not involved in executing the COMPOSE request, are running on a separate node not considered here. The SimGrid calibrated simulator is generated using the Jaeger trace partially shown in Figure 3 and the code generation script described in Section III. We run the experiment with various CPU constraints. Figure 8 shows the result obtained with 10 cores per node to execute the application.

We observe one limitation of our approach during this experiment as our model does not capture the communication overhead due to Apache Thrift under high load. We choose to reduce this overhead (that is known to the authors of [9]) by executing two instances of each service, as would be done in a real deployment to improve the application throughput. Such fine-grain overheads could be considered in future work.

Figure 8 shows the maximum request throughput estimated by SimGrid and obtained through real-world executions. We observe that SimGrid accurately detects the breaking point where the application becomes saturated, at around 1,500 RPS with 1x10 cores for configuration 1, and 1,750 RPS and 2,200 RPS under configurations 2.a and 2.b with 2x10 cores. We observe similar results when comparing the average latency of requests. A non-proportional maximum throughput between the configurations can be observed. Indeed, configuration 2.a presents an unbalanced grouping of services among the nodes.
which leads to the overloading of one node while the other is still able to process requests. A very naive service model would predict an improvement factor of two between the configurations 1 and 2.a, and would predict identical performance for configurations 2.a and 2.b. Since our model takes into account the processing cost of each service, it is thus able to show that configuration 2.b performs better than configuration 2.a, as in the real executions.

Regarding the accuracy of our predictions, the configurations 1, 2.a and 2.b have an average MRE of respectively 3.5%, 3.7% and 1.4%, whereas the maximum measured errors are 21%, 20% and 6.6%. The maximum error values can be explained, as with the TeaStore experiment, by a different behavior of our model compared to the real application once the breaking point is reached. Before that point, the error remains very low between the simulated predictions and real measurements, while the breaking point is observable in Figure 8, at 1500, 1700 and 2250 requests per second in scenarios 1, 2.b and 2.b.

These experimental results show that our approach, based on a microservice model and code tracing tools, provides accurate estimations of the performance of microservice applications with different configurations, thus answering both questions:

Q3. Will distributing microservices on more than one node increase performance?
Q4. How to optimize the location of services in a computing cluster to obtain the best performance?

V. RELATED WORK

To improve the performance of their applications, microservice developers can, among others: use automatic scaling policies based either on system thresholds [13] or application metrics [10], detect and correct application faults [7] or optimize service placements [1].

To evaluate the advantages of using such methods, two main approaches exist in the literature around microservice performance evaluation. On one hand, real-world experiments [4], [13]. Despite giving meaningful insights on the behavior of an application, because of time and real platform constraints, these methods often limits the validity of their results to a narrow set of contexts. They also do not allow for easily reproducible results. The second approach uses purely analytical models [3], [7]. This approach allows observing some theoretical properties of an application but is not adapted to study complex applications composed of many services.

To overcome the lack of evaluation in various contexts through real experiments and the difficulty of adapting formal methods to real systems, a third approach resorts to simulation. With simulation, one can observe the behavior of an application under several configurations without real deployments and reproduce the obtained results using only commodity hardware. We propose a brief overview of existing simulation software for microservice applications and their limits.

The authors of BigHouse have obtained performance predictions of cloud applications using statistical methods based on a discrete-event simulator [18]. Unfortunately, more recent works have shown that the accuracy of BigHouse is limited when applications are composed of several services due to its very high-level representation of applications, not taking into account some specificities of microservices [25].

Based on the limits of BigHouse, the authors of [25] used a finer-grained representation of microservice applications with µqsim. µqsim allows for a detailed representation of internal microservice executions using sets of execution stages for each type of request a microservice can receive along with communication dependencies between the different services. The relation between services is modeled as a DAG where nodes are services and edges represent the path followed by individual requests processed by the application. With a correct calibration of the different execution stages of each service and their interactions for different types of requests, the authors managed to reproduce the behavior of a complex microservice benchmark [9]. With µqsim, the accuracy of the simulation depends on the description of the different processing stages inside microservices and of service dependencies. Whereas this calibration can be done easily by hand with small applications, the lack of a proper calibration methodology leads to a tedious and error-prone process with large applications. This approach also requires to re-instrument a service each time its code is modified in order to fit its new behavior. Compared to µqsim, our approach introduces a simplified representation of intra-microservice processing by using a fixed set of execution stages (i.e. 3 stages as shown on Figure 1). But, these stages
are automatically calibrated using standard execution traces obtained through well-known distributed tracing systems.

Other tools allow for microservice simulation studies. Simulators dedicated to Fog and edge computing systems often model their applications as DAGs of tasks, like YAFS [15], fogTorch [5], and IFogSim [12]. Yet, these simulators are very specific to the context of Fog, and still require to manually describe most of the dependencies and properties of the simulated applications. For instance, the authors of [6] provide a code generation pipeline with Fog specific properties, such as geographical coordinates, and Fog applications, such as dataflow processing which do not correspond to microservice architectures although both employ DAG representations.

Evaluating the performance of an application based on execution traces is not a new approach [17]. Our approach is a new step towards the seamless transposition of real applications into simulations. Similar approaches require manual descriptions of intra-service and inter-service properties that is both costly and error-prone, especially for large applications.

VI. CONCLUSION AND FUTURE WORK

Microservice applications trade monolithic complexity for intricate interactions between simple services, hindering the system performance evaluation.

In this article, we introduced a microservice simulation model based on a reduced number of calibration values to describe microservice applications. Our contribution is more precise than large grain models, while being easier to instantiate than precise models. We proposed a methodology leveraging distributed tracing systems to instantiate the simulation models of real applications using standard instrumentation solutions.

We implemented this model on top of SimGrid, and applied our methodology on applications instrumented with Jaeger and Kieker. These contributions were evaluated on microservice benchmarks, demonstrating their ability to answer the operational questions Q.1-4 introduced in Section I on such applications. This could be used in various what-if analyses such as the exploration of performance trade-offs under scarce resources that are common in fog infrastructures. More interestingly, it could even be used to dimension a Fog infrastructure given an application and a workload to serve, an intractable problem with other solutions.

In the future, we will provide models of features that are common among microservice applications, such as the Kubernetes autoscaling policies. We also plan to give access to detailed resource usage values, including energy dissipation, through classical instrumentation systems.

ACKNOWLEDGMENT

Experiments presented in this paper were carried out using the Grid’5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

REFERENCES

Abstract—Magnetic tape provides a cost-effective way to retain the exponentially increasing volumes of data. The low cost per gigabyte and the low energy consumption render tape a preferred option over hard disk drives and flash for infrequently accessed data. Assessing the performance of tape library systems is central to achieving appropriate storage provisioning and dimensioning. Performance is affected by the number and the operational characteristics of the tape drives and the robotic arms, and the mount and unmount policies deployed. In this paper, we develop a novel analytical model that accurately captures the principal aspects of tape library operation. Several relevant performance measures including the mean waiting time and the mount/unmount rates are derived. The model provides useful insights into the behavior of the tape library mechanisms and yields results, which enable a better understanding of the design tradeoffs. The validity of the model developed is confirmed by demonstrating a good agreement of the predicted performance with that obtained by simulation across various configurations.

I. INTRODUCTION

The exponential data growth caused by factors such as the continuous rise of the Internet of Things (IoT), the emergence of high-definition 4K/8K videos and AI-based big-data analytics, poses serious challenges for modern storage systems. There are now more than 540 hyperscale data centers in the world storing an estimated 547 exabytes (EB) of data, with more than 170 additional facilities under construction [1]. Unfortunately, increased data storage implies increased energy consumption. Today’s data centers consume 2% of the world’s electricity and it is expected to increase to 8% by 2030 [1]. The one technology with the continued capacity scaling potential to address in a cost-effective way the massive growth of digital data for some of the largest hyperscale data centers in the world and to keep it protected from cybercrime attacks, is a technology more than 60 years old – magnetic tape.

In its more recent history, tape has been used by corporations and institutions for economically storing the bulk of infrequently accessed data [2]. Multi-tiered storage systems are currently deployed to efficiently store data with vastly different access characteristics. The key advantage of tape over hard disk drives and flash is its low cost per gigabyte and substantial energy savings. Steady improvements in tape technology deliver ever-increasing storage capacities at low cost. Furthermore, tape is a highly scalable and reliable media, which guarantees that cloud providers will be capable of accessing long-term stored data well into the future. In addition, tape storage provides high security as it mitigates the attempts of cyberattacks to corrupt data. In particular, quantum-safe tape drives are also being developed to help future-proof tape against potential new threats in the coming decades [3].

Modern enterprise tape has reached a native capacity of 20 TB per cartridge with data rates reaching 400 MB/s with a great potential for further capacity and bandwidth growth [4, 5].

A tape library system typically contains tape drives for performing read/write operations on a large number of tape cartridges, which are mounted in and unmounted from the tape drives through an automated mechanism (robot arms). To perform a read/write access to a tape cartridge, the cartridge should first be mounted in a free drive, then be loaded, and subsequently, after a seek time, be appropriately positioned to read/write the requested data. Also, the unmount time of a tape includes the rewind and unload time before it is removed from the corresponding tape drive. A characteristic of tapes is that access latencies are relatively high and can run into minutes even on lightly loaded systems. Although tape libraries are economically attractive, the tape drive components are expensive compared to tape cartridges and therefore need to be properly provisioned for an efficient and cost-effective operation. The same applies to the drives used in optical disk archives that accept removable media [6].

To properly dimension tape library systems, it is imperative to have the means to evaluate the effect of the various parameters on their performance. In this article, we present a comprehensive realistic theoretical analytical model that captures the effect of robot operation and variability of seek times on system performance. Previous work neglected these effects. This novel model yields performance measures of interest, such as the mean waiting time and the robot mount/unmount rates as a function of the system parameters, including the number of tape cartridges, the number of tape drives, the number of robot arms, and the tape load, seek, rewind, and unload times. The results obtained can subsequently be used to dimension and provision tape systems to achieve service level agreements for given workloads and data access times.

The remainder of the paper is organized as follows. Section II provides a survey of the relevant literature on performance evaluation of tape library systems. Section III describes the operation of a tape library system. Section IV describes the tape system model along with the corresponding parameters and presents the analytical evaluation of the mean waiting times and the robot mount rates. Section V shows numerical results for typical tape systems that demonstrate the effect of robot arms on system performance. Finally, we conclude in Section VI.
II. RELATED WORK

Over the last six decades there is a modest number of publications that studied the performance of tape library systems. The theoretical evaluation of their performance turns out to be an extremely challenging task and consequently the evaluation was mainly conducted by simulations [7-9] and measurements [6]. However, simulations and measurements are time-consuming compared with analytical models that not only provide fast execution times, but also insight into the system dynamics and the effect of the various parameters.

The first analytical attempt resulted in an intractable queueing model, which led the authors to conduct the study by resorting to simulation [7]. Twenty years later, the next approximate analysis was conducted using $M/M/c$ and $M^X/G/c$ queueing models [10, 11]. Another twenty years later, an improved model that captures the polling nature of operation of tape library systems was presented in [12]. It obtained the mean delay for independent and identically distributed (i.i.d) requests submitted to tapes are assumed to be i.i.d. When

$$Q_{max} = \lambda c \theta a$$

Note that $\theta$ is the relative arrival rate, $\theta = \lambda / \lambda_{max} = \lambda B_{max} / d$. The notation is summarized in Table I. The parameters are divided according to whether they are independent or derived and are listed in the upper and the lower part of the table, respectively.

Let $I$ and $\bar{I}$ denote the events that an arbitrary request is initial and non-initial, with corresponding probabilities $P_I$ and $P_{\bar{I}}$, respectively. Let also $I_N$ and $s_N$ denote the respective seek times of initial and non-initial requests. Then the seek time $s$ of an arbitrary request is $s = s_I + s_{\bar{I}}$. Note that $P_I$ and $P_{\bar{I}}$, and in turn $s$, depends on the load of the system. The first and second moments of a random variable $X$ are denoted by $X$ and $X^2$, respectively, such that $\bar{s}$ and $s^2$ denote the first and second moments of the seek time $s$, respectively, obtained by

$$\bar{s} = P_I \bar{s}_I + P_{\bar{I}} \bar{s}_{\bar{I}} \quad \text{and} \quad s^2 = P_I s_I^2 + P_{\bar{I}} s_{\bar{I}}^2 + P_N s_N^2 \cdot$$

Assuming a uniform random workload, the moments of the order statistics yield

$$s = s_{max}/2, \quad s_I^2 = s_{max}^2 / 3, \quad s_{\bar{I}}^2 = s_{max}^2 / 3, \quad s_{N}^2 = s_{max}^2 / 6 \cdot$$

where $s_{max}$ is the maximum seek time corresponding to a request for the data located at the end of a tape.

Each request incurs a seek time $s$ and a transfer time $t_T$ of $Q/b_w$, where $b_w$ denotes the transfer bandwidth. The total time to serve an initial and a non-initial request is $B_T = s_I + t_T$.
and $B_N = s_N + t_T$, respectively. The total time to serve an arbitrary request is

$$B = B_1 1_I + B_N 1_I = s + t_T,$$

such that $B = P_t B_I + P_N B_N = \pi + Q/b_w$, \hspace{1cm} (3)

and $B^2 = P_t B_I^2 + P_N B_N^2 = \bar{\pi}^2 + \bar{Q}^2/b_w + 2 \pi Q/b_w$. \hspace{1cm} (4)

Owing to the assumptions for $s_I$, $s_N$, and $Q$, it follows that the service times $B_1$, $B_N$, and $B$ are i.i.d. random variables.

The load $\rho$ of the system is given by

$$\rho = \frac{\lambda B(\lambda)}{d}. \hspace{1cm} (6)$$

The maximum throughput of the system $\lambda_{\text{max}}$ is obtained by setting $\rho = 1$ and recognizing that $P_t \to 0$, such that $B(\lambda_{\text{max}}) = B_N$. Also, the relative arrival rate $\theta$ is

$$\theta \triangleq \lambda/\lambda_{\text{max}} = \lambda B_N/d, \hspace{0.5cm} \text{where} \hspace{0.5cm} \lambda_{\text{max}} = d/B_N. \hspace{1cm} (7)$$

IV. TAPE SYSTEM MODELING AND ANALYSIS

The performance of a tape library system is assessed using queueing modeling whereby requests submitted to tapes correspond to jobs arriving to queues and tape drives correspond to servers. The system operation and performance can be accurately modeled in the light-load and heavy-load regions. In [12] it was shown that the mean delay in the light- and heavy-load regions is well approximated by the green curve (1) obtained analytically from an $M/G/K$ model, and the brown curve obtained analytically from an $M/G/1$ polling queueing model, as depicted in Fig. 1. Subsequently, the mean delay in the medium-load region is derived through interpolation. The simulation results obtained in [12] and indicated by the blue boxes suggested that, in the medium-load region, the delay curve can be approximated by the unique blue line that is tangent to the green and brown curves. It turns out that this only holds when the seek times are i.i.d. In practice, however, our simulation results presented in Section V show that the mean delay curve typically has a very different shape, as indicated by the blue boxes in Fig. 1.

In this article, we consider novel, elaborate, non-standard queueing theory models and use them to develop a comprehensive and accurate model for the operation of a tape library system. First, we extend in a non-trivial manner the queueing model that was applied in the light-load region and obtain a new green curve (2), as shown in Fig. 1, that no longer saturates at load $\rho = \rho^*$. The model in the heavy-load region is also enhanced to account for the robot arm contention. The difference, however, of the brown curve is negligible. Subsequently, we use the fact that now both curves saturate at $\rho = 1$ to obtain an interpolation between them. After extensive experimentation, we construct a suitable interpolation function that yields the mean waiting time shown by the orange curve, which in the light- and heavy-load regions is essentially determined by the green and brown curves, respectively. Also note that the mean waiting times obtained analytically from the $M/G/K$ and $M/G/1$ polling queueing models depend only on the first and second moments of the corresponding service times.

A. Light-Load Modeling

Here we develop a model for the mean waiting time that is very accurate in the light-load region, that is, when $\lambda$ is relatively small. It builds on the virtual queue model presented in [12], whereby all outstanding requests form a virtual queue that is served by the $d$ tape drives. At light load, all requests are initial (green jobs shown in Fig. 2) with a corresponding fictitious service time $S_f = U + M + B$. Next, we extend the model in two ways. First, we consider the waiting times for the robot. Second, in addition to the initial (green) requests, we also consider the non-initial (blue) pending requests that appear when the load increases. We proceed by analyzing the two policies.

1) Not-Unmount Policy: Consider an initial request to a tape, say tape-A. It can only be mounted to a tape drive after the tape occupying the drive is removed. Thus, it takes a time $W_{\text{UT}}$ until a robot accesses the tape and then a time $R$ for the robot to return this tape to a storage slot. Subsequently, tape-A is scheduled for mounting and the robot immediately fetches tape-A and mounts it after a time $R$, which implies that $W_{\text{MT}} = 0$. Thus, initial requests trigger an unmount/mount cycle where the robot waiting time is $W_{\text{RU,NU}} = W_{\text{UT}}$. As non-initial requests do not incur robot waiting times, the fictitious service time $S_{f,\text{NU}}$ of requests is

$$S_{f,\text{NU}} = S_f = [W_{\text{RU,NU}} + U + M + B_I] 1_I + B_N 1_I,$$

where for ease of reading, and for all variables in the remainder of the section, we suppress the subscript NU. The first two moments of $S_f$ are obtained from (4), (5), and (8), as follows:

$$\overline{S_f} = \bar{B} + P_t \left( W_{\text{RU}} + \bar{U} + \bar{M} \right), \hspace{1cm} (9)$$

$$\overline{S^2_f} = \bar{B}^2 + P_t \left\{ W_{\text{RU}}^2 + \bar{U}^2 + \bar{M}^2 + 2 \bar{U} \bar{M} \right\} + 2 \left\{ W_{\text{RU}} ( B_I ( U + M ) + W_{\text{RU}} B_I ) \right\}. \hspace{1cm} (10)$$

At first glance, it seems that the robot mean waiting time $W_{\text{RU}}$ can be determined by an $M/G/K$ robot queue with

![Fig. 1. Mean queueing delay in the light- and heavy-load regions; interpolation in the medium-load region.](image-url)
a servers, where the service times are $2R$, and the initial
requests form the arrival process with a rate $\lambda_{ib}$ obtained by
$$\lambda_{ib} = P_I \lambda. \quad (11)$$

The tape library operation, however, complicates matters
because it prohibits a direct application of $M/G/K$ results.
This is because an initial request starts waiting for a robot not
when it arrives, but when it triggers an unmount/mount cycle.
This is depicted in Fig. 5 where the waiting time is measured
after a request crosses the red boundary at position $Z$ in
the queue. In the absence of theoretical results for this waiting
time in the literature, we derive it in the Appendix. Let us
consider the maximum number of initial requests $L$ that can
be active, that is, they have triggered an unmount/mount cycle
and either wait for a robot arm to unmount the corresponding
tape or a robot is performing the unmount/unmount operations.

Clearly, for fixed robot transfer times, it holds that
$$E[W^i_{rb}(L)] = E[W^i_{max(L-a,0),0}], \quad i = 1, 2, \ldots, \quad (12)$$
where the moments of $W_{Z,0}$ are obtained from (40) by
considering $\lambda = \lambda_{ib}$, $\mu = 1/(2R)$, $C = 0$, and $K = a$.

Another complication arises from the fact that the boundary
is movable. For instance, when there are no pending or
processed requests, the maximum number of initial requests
that can trigger a cycle and engage a robot is equal to the
number of tape drives, such that $L = d$. On the other hand,
if all drives serve tape requests, none of the initial ones can
trigger an unmount/mount cycle and engage a robot, which
implies that $L = 0$. Evidently, the boundary position depends
on the state of the virtual queue.

Let $A$ be the number of active requests given that there
are $J$ requests present in the system ($A \leq \min(J,d)$).
The number of additional initial requests that can become
active is equal to the number $\max(d-J,0)$ of idle tape
drives. Thus, $L = A + \max(d-J,0)$.

Note that requests in fictitious service in the virtual queue represent requests that are either non-initial being served or initial being served or initial that they have triggered loading/unloading/rewinding
operations or initial active that they have triggered robot
waiting/unmounting/mounting operations. Therefore, the
probability $P_A$ that a request in fictitious service represents an
initial active request is
$$P_A = P_I (\overline{W}_{rb} + 2R) / \overline{S}_f. \quad (13)$$
We assume that $A$ follows a binomial distribution with param-
ter $P_A$. For $j^* = \min(j,d)$ and $i = 0, \ldots, j^*$ we then have
$$P(A = i \mid J = j) = (\begin{smallmatrix} j^* \\ i \end{smallmatrix}) P_A^i (1 - P_A)^{j^* - i}. \quad (14)$$

The conditional steady-state probability density function (pdf)
of $L$ is determined as follows:
$$P(L = l \mid J = j) = P(A = l - \max(d - j,0) \mid J = j)$$
$$= \begin{cases} \binom{j^* - d + j}{l - d + j} P_A^{d+j} (1 - P_A)^{d-l}, & \text{for } 0 < d - j \leq l \leq d \\
\binom{l}{l} P_A^l (1 - P_A)^{d-l}, & \text{for } l \leq d \leq j \\
0, & \text{otherwise}. \quad (15)\end{cases}$$

Unconditioning (15) on $J$, and using (32), yields
$$P(L = l) = \sum_{j=0}^{\infty} P(L = l \mid J = j) q_j. \quad (16)$$

Unconditioning (12) on $L$, and using (16), yields the first two
moments of the robot waiting time
$$E[W^i_{rb}] = \sum_{l=a}^{d} E[W^i_{rb}(l-a)] P(L = l), \quad \text{for } i = 1, 2. \quad (17)$$

Note that for $i = 1$, (17) suggests that $\overline{W}_{rb}$ is a function
of $P(L = l)$, which in turn, and according to (13), (14),
(15), and (16), is a function of $\overline{W}_{rb}$. Consequently, the value
of $\overline{W}_{rb}$ is obtained by an iterative procedure, which upon
convergence also yields the corresponding $E[W^i_{rb}]$. We have
confirmed convergence for various configurations tested and
for any given value of $P_I$, which is yet unknown.

Subsequently, we determine the coefficient of variation
$C_{S_f}$ of $S_f$ via (9), (10), and (17), which we then use in (39) to
derive the fictitious mean waiting time,
$$\overline{W}_f = E[W_{rb}^1(l-a) + S_f]. \quad (18)$$

A request arriving to a tape is initial if the tape does not
contain any request and is unmounted. The probability that a
tape is unmounted is $1 - d/c$. Owing to the symmetric
workload, the steady-state probability $P_e$ that a tape does not
contain any request is $(1 - 1/c)^N$, which for large values of
$c$ is approximately equal to $e^{-N/c}$ given that $(1 - 1/c)^c \approx e$.
Consequently, from the PASTA property we deduce that
$$P_I = (1 - d/c) P_e, \quad \text{where } P_e = e^{-N/c}. \quad (19)$$

Note that (19) suggests that $P_I$ is a function of $N$, which in
turn, and according to (9)–(18), is a function of $P_I$. Therefore,
the value of $P_I$ is obtained by an iterative procedure, which
our investigation showed that it always converges.

As the actual waiting times of requests are defined between
the arrival and service initiation times, they are longer than
the fictitious ones. They also include the seek times and the
robot waiting and mounting/unmounting times. Thus,
$$W_{NU1} = W_f + [W_{rb} + U + M + s_f] 1_f + s_N 1_f, \quad (20)$$
which in turn, and using (1), yields the mean waiting time:
$$\overline{W}_{NU1} = \overline{W}_f + P_I (\overline{W}_{rb} + \overline{U} + \overline{M}) \overline{+}. \quad (21)$$

Also, the rate $\lambda_{ub}$ of the robot mount (or unmount) operations
is determined by (11).

2) Always-Unmount Policy: This policy is equivalent to the
Not-Unmount policy when at the times when tapes finish
serving their requests, it is very likely that other tapes with
pending requests wait to be mounted. This is reflected by the
probability $V$ that all $d$ servers of the fictitious queue are
busy, which is determined in the Appendix by (34). Thus, for
$V \rightarrow 1$, $S_{f,NU} \approx S_{f,NU}$. On the other hand, when $V \rightarrow 0$,
the initial and last tape requests form the mount and unmount
arrival processes to the robot queue with equal rates $\lambda_{rb,MT}$ and $\lambda_{rb,UT}$, respectively. Clearly, compared to the NU policy, the rate of arrivals to the robot queue is twice as large, but the service time is $R$, which is half. Therefore, the load is the same ($\rho_{rb} = \lambda_{rb} R$), but $W_{rb,NU} = W_{rb,NU}/2$. To reduce the waiting time of requests, the mounts are prioritized over unmounts, such that $W_{rb,MT} = (1 - \rho_{rb}) W_{rb,MT}/(1 - \rho_{rb}/2)$ and $W_{rb,UT} = W_{rb,UT}/(1 - \rho_{rb}/2)$ [13, Equation (3.31)]. Also, the second moments are obtained analogously.

Let $IF$, $IF'$, $IF$, and $IF'$ denote the events that an arbitrary request is initial and final, initial and non-final, non-initial and final, and non-initial and non-final, respectively. It can be shown that the corresponding probabilities are $P_{I}^{2}, P_{I} P_{N}, P_{N} P_{I},$ and $P_{N}^{2}$, respectively. Then, the fictitious service time $S_{f, AU, 0}$ of requests is

$$S_{f, AU, 0} = [W_{rb, MT} + B_{I} + U + W_{rb, UT} + B_{N}] \times \frac{1}{2} F' + [W_{rb, MT} + M + B_{I}] \times \frac{1}{2} F + [B_{N} + U + W_{rb, UT}] \times \frac{1}{2} F',$$

and its moments are obtained using the above probabilities. Combining the two cases, we obtain $S_{f, AU}$ as the weighted average of $S_{f, AU, 1}$ and $S_{f, AU, 0}$ as follows:

$$S_{f, AU} = V S_{f, AU, 1} + (1 - V) S_{f, AU, 0} \approx V S_{f, NU} + (1 - V) S_{f, AU, 0},$$

with $S_{f, NU}$ and $S_{f, AU, 0}$ given by (8) and (22), respectively. The two moments and the coefficient of variation $C_{S_{f}}$ of $S_{f, AU}$ are then obtained and used in (39) to derive the fictitious mean waiting time, $\bar{W}_{f} = E[W_{f}]/E[C_{f}^{2}]$. Similar to the case of the NU policy, the value of $P_{I}$ is determined by an iterative procedure using the analogous to (18) and (19) expressions, for where the AU policy it holds that $P_{I} = P_{s} = e^{-\rho_{G}/c}$, given that, after completion of service, tapes are unmounted.

Adding to the fictitious waiting times the seek times and the robot waiting and mount/unmount times, and using (20), (21), (22), and (23), yields

$$W_{AU_1} = W_{f} + s_{N} \times \frac{1}{2} F + [V(W_{rb, NU} + U) + (1 - V) W_{rb, MT} + M + s_{I}] \times \frac{1}{2} F,$$

which in turn, and using (1), yields the mean waiting time:

$$\bar{W}_{AU_1} = \bar{W}_{f} + P_{I} [V(W_{rb, NU} + U) + (1 - V) W_{rb, MT} + M] + \sigma.$$

Also, the rate $\lambda_{rb}$ of the robot mount (or unmount) operations is determined by (11).

Remark 1: It turns out that the light-load model provides accurate results for the $\rho$, $s$, $B$, and $P_{I}$ variables in the entire spectrum of loads.

B. Heavy-Load Analysis

Here we develop a model suitable for the analysis at high loads. It applies to both the NU and AU policies given that at high loads, when all requests for a given tape are served, there are pending requests and therefore the tape is unmounted. It builds on the polling model presented in [12], whereby the system is partitioned in $d$ domains, with each domain containing a set of $n = c/d$ tapes that are served by a single tape drive. The arrival process of requests to each of the domains is Poisson with rate $\lambda_{dm} = n \lambda_{ct} = \lambda / d$, the load is $\rho = \lambda_{dm} \bar{B} = \lambda \bar{B} / d$, and the mean waiting time, $\bar{W}_{h}$, is given by

$$\bar{W}_{h} = \frac{\lambda \bar{B}^2}{2 d (1 - \rho)} + \frac{s_{um}}{2} \left[ \frac{n - \rho}{1 - \rho} + \frac{s_{um}^2 - s_{um}^2}{s_{um}^2} \right],$$

where $s_{um}$ is the setup time, with $s_{um} = U + M$.

Next, we extend the model in two ways by considering the effect of robot arm and that of the various seek times. First, we consider the robot waiting time component $W_{rbh}$ of the setup time, such that

$$s_{um} = W_{rbh} + U + M.$$

In [14] it was shown that the following relation holds

$$E[S] = (1 - \rho) E[C],$$

where $C$ is the period between successive visits to a given queue and $S$ is the time spent for setups in this period. It holds that $E[S] = n s_{um}$, which implies that

$$s_{um} = (1 - \rho) E[C]/n = (1 - \rho) E[T],$$

where $T$ is the time between successive visits to queues. Regarding the robot model presented in Section IV-A1 for the NU policy, these visits correspond to the instants where the initial requests cross the robot queue boundary and become active. This generates a renewal arrival process of initial active requests to the robot queue with a rate of $1/T$, for each of the $d$ domains. The superposition of the $d$ renewal processes forms the arrival process to the robot queue, which, according to the renewal theorem, converges to Poisson with arrival rate $\lambda_{rbh} = d/T$. From (27) and (29) it follows that

$$\lambda_{rbh} = d/T = d (1 - \rho) / (W_{rbh} + U + M).$$

Also, the robot mean waiting time $\bar{W}_{rbh}$ is determined by an $M/G/K$ queue with $\phi$ servers, and service times of $2 \bar{R}$. Its moments $\bar{W}_{rbh}$ and $\bar{W}_{rbh}^{2}$ are obtained from (40) by considering $\lambda = \lambda_{rbh}, \mu = 1/(2R), C = 0, K = \phi$, and $V_{Z} \approx 0$ given that for heavy loads the robot activity is light.

Note that $\bar{W}_{rbh}$ is a function of $\lambda_{rbh}$, which in turn, and according to (30) is a function of $\bar{W}_{rbh}$. Consequently, the value of $\bar{W}_{rbh}$ is obtained by an iterative procedure, which upon convergence also yields the corresponding $\bar{W}_{rbh}^{2}$. Subsequently, using (27), the moments of $s_{um}$ are obtained. Based on Remark 1, substituting the $\rho$ and $\bar{B}^2$ values obtained from the light-load analysis along with the $s_{um}$ moments into (26) yields the mean waiting time.

C. Medium-Load Analysis

In Section IV-A we have obtained in (21) and (25) the mean waiting time $W_{1}$ ($W_{NU1}$ and $W_{AU1}$), which is accurate for light loads and in Section IV-B we have obtained in (26) the mean waiting time $\bar{W}_{h}$, which is accurate for heavy loads. Our objective is to specify an interpolation function between the light-load and heavy-load curves that produces accurate results.
for medium loads as well. After extensive experimentation, we propose the following interpolation function:

$$\overline{W} = P_e \overline{W}_l + (1 - P_e) \overline{W}_h V \tag{31}$$

For a weighted average of $\overline{W}_l$ and $\overline{W}_h$, the weight $P_e$ is appropriate because at light loads $P_e \approx 1$ whereas at heavy loads $P_e \approx 0$. However, at light loads, and for a small $\epsilon$ the value of $(1 - \epsilon) \overline{W}_l + \epsilon \overline{W}_h$ is significantly deviating from $\overline{W}_l$ because the large value of $\overline{W}_h$ causes the product $\epsilon \overline{W}_h$ not to be negligible as desired. We address this issue by raising $\overline{W}_h$ to the power of $V$, which is an indicator of the occupancy of the virtual queue determined in (34). At light loads, $V \approx 0$, which cancels the effect of $\overline{W}_h$ whereas at high loads, $V \approx 1$, which restores the effect of $\overline{W}_h$.

V. Numerical Results

Here we assess the performance of a tape library by using both theoretical predictions and event-driven simulations. We have confirmed the validity of the model by considering scenarios for a range of distributions, parameter values, and for small and large configurations. Owing to lack of space, we present the scenarios for the parameter values listed in Table II. We begin by presenting the specific results for the IBM TS4500 tape library system [15]. The system comprises $c = 3200$ cartridges and $d = 32$ drives, such that $n = 100$. The robot access times are fixed equal to $R = 5$ s and the mount and unmount times are considered to be fixed, equal to $M = 20$ s and $U = 88$ s, respectively. The workload in each tape is assumed to be symmetric with requests incurring random, uniformly distributed seek times in the interval $[0, 118]$ s. As in [12], the distribution of I/O request sizes is taken to be the same as the file size distribution of CERN [16], whose mean $\overline{Q}$ is equal to 843 MB, the standard deviation to 2.8 GB and the second moment $\overline{Q}^2$ to 8.9 GB$^2$. The transfer bandwidth is assumed to be $b_w = 360$ MB/s.

First, we consider the case where the library uses two robot arms ($a = 2$). Fig. 3 shows results for various performance measures vs. the relative arrival rate. The theoretical mean waiting times for the NU policy derived from the light- and heavy-load analysis are obtained from (21) and (26), and shown in Fig. 3(a) by the green and brown dotted lines, respectively. The mean waiting times for the two policies are obtained through interpolation from (31). The red curve for the AU policy is barely visible because it lies just below the blue one for the NU policy. This also occurs for most of the performance measures considered. Note that at $\theta = 0.2$ the mean waiting time starts increasing drastically. The simulation results are indicated by the blue circles, with the 95% confidence intervals being extremely narrow, owing to about a dozen of lengthy replications run for each point, and therefore not shown.

We observe that the theoretical interpolation curves capture the system behavior as they match well with the simulation results with some deviation at medium loads. This is reflected in Fig. 3(b), which shows the ratio of the analytical predictions to the corresponding simulation values. We observe that the deviation for light and heavy loads is less than 6%. At medium loads the deviation increases and exhibits a spike with a peak of about 20% only in a narrow region.

The robot activity measured by the number of mounts per hour is obtained from (11) and shown in Fig. 3(c). Interestingly, the theoretical results match with the simulation ones in the entire range of loads, despite the fact that the expression for $\lambda_{h}$ was obtained by the light-load analysis. As the load increases, the robot activity also increases, but after some point it starts decreasing. Clearly, at high loads there are many requests to be served such that tapes remain mounted for long periods of time. Thus, the time intervals between unmount/mount operations are long, which implies a reduced robot activity.

Fig. 3(d) shows the probability that an arriving request is initial and this measure, too, is assessed accurately in the entire range of loads, despite the fact it was obtained by the light-load analysis in (19). As a result, the load of the system obtained from (6) is also assessed accurately as shown in Fig. 3(e). The difference between $\rho$ and $\theta$ is shown by the dotted line. At high loads, it holds that $\rho \approx \theta$.

Fig. 3(f) shows the results for the ratio of the mean waiting times of the AU to those of the NU policy. As expected, at light loads the AU policy results in lower waiting times because it does not waste any time to unmount cartridges when requests arrive. However, depending on the parameters, this may not always be the case. For instance, if the mount times are much longer than the unmount times, then it may be preferable for a tape to remain mounted in anticipation of future requests arriving to it. As the load increases, the performance of the AU policy approaches that of the NU policy.

Next, we consider the case where the library operates using one robot arm ($a = 1$). Fig. 4 shows the corresponding performance results. Note that, according to (7), the maximum throughput of the system $\lambda_{\text{max}}$ does not depend on the number of robot arms deployed. Consequently, a given value of $\theta$ corresponds to the same arrival rate. Fig. 4(a) demonstrates a dramatic increase of the mean waiting time at $\theta = 0.1$. This is due to the fact that at this load, the robot arm becomes a bottleneck as it is constantly busy performing mount/unmount operations. This is reflected by the flat part of the robot operation activity shown in Fig. 4(c). It can be proven that in general the robotic mechanism becomes a bottleneck.
when \( d/(B_U + U + M) \) is greater than \( a/(2R) \), which is the maximum possible rate of robot activity, or, equivalently, \( a < 2R d/(B_U + U + M) \), which for the values of Table II equals 1.89. However, at high loads the robot is no longer a bottleneck as its activity is reduced. We observe that also in this configuration, the theoretical curves match well with the simulation results. In fact, in Fig. 4(f) the theoretical model predicts that as the load increases, the ratio of the mean waiting times of the AU to those of the NU policy initially decreases. This is also confirmed by the simulation results, which establishes a confidence for the model presented.

From the results presented, it follows that the theoretical model captures the behavior of the tape library system quite accurately.
VI. CONCLUSIONS

The continuing exponential growth of data is fueling demand for cost-effective tape storage. To properly dimension tape library systems, it is imperative to evaluate their performance. Our work is the first to provide a realistic accurate analytical model for assessing the performance of tape library systems and to investigate the effect of the key parameters, namely, the number of cartridges, drives, and robot arms on the mean waiting time and the sustained rate of robot arm operations. We confirmed that the theoretical results are in agreement with simulation results, which establishes the validity of the model developed. It was shown that, under certain conditions, the robot mechanism may become a bottleneck degrading system performance, which can be improved by increasing the number of robot arms. The model also captures the effect of the mount/unmount policy deployed. For light loads, the Always-Unmount policy was shown to yield a mean delay that is lower than that of the Not-Unmount policy, with the difference in certain configurations becoming even larger as the load increases. For high loads though, and in all configurations, this difference becomes negligible.

Extending the theoretical model to incorporate advanced scheduling policies and study the performance of asymmetric and bursty workloads is a subject of further investigation.

APPENDIX

The steady-state probabilities of the number $J$ of jobs in an $M/M/K$ queue are determined based on the load $\rho = \lambda/(K\mu)$ as follows [13, Equations (1.72) and (1.73)]:

$$q_j = \begin{cases} p_0 (K\rho)^j/i! , & \text{for } j = 0, \ldots, K \\ p_0 K^K \rho^j/K! , & \text{for } j = K, K+1, \ldots, \end{cases}$$

(32)

where $p_0 = \{(K\rho)^K/(1-\rho)^{K+1} + \sum_{i=0}^{K-1} (K\rho)^i/i!\}^{-1}$. (33)

Then the probability $V$ that all servers are busy is $\sum_{j=K}^\infty q_j$,

$$V = p_0 (K\rho)^K/[1-(1-\rho)K!],$$

(34)

and the probability $V_Z = P(J \geq Z + K)$ is

$$V_Z = p_0 (K\rho)^K (1-\rho^Z)/[(1-\rho)K!].$$

(35)

The mean waiting time $E[W]$ is then derived as follows:

$$E[W] = p_0 (K\rho)^K/[1-(1-\rho)^2 K!K\mu]$$

(36)

Next, we derive the waiting time $W_Z(J)$ of an arrival that finds $J$ jobs present using the fact that the sequence $\{X_j\}$ of inter-departures are exponential i.i.d random variables with mean $1/(K\mu)$. For $J < K$, $W_Z(J)$ is zero; for $K \leq J < K + Z - 1$ it holds that $W_Z(J) = \sum_{j=K+1}^{J+1} X_j$ such that $E[W_Z(J)] = J - K + 1/(K\mu)$, and for $J \geq K + Z - 1$ it holds that $\sum_{j=K}^{\infty} X_j$ such that $E[W_Z(J)] = Z/(K\mu)$. Then

$$E[W^*_Z] = \sum_{j=K}^\infty E[W_Z(j)] q_j,$$

which for $i = 1, 2$ and using (32) yields

$$E[W_Z] = p_0 (K\rho)^K (1-\rho^Z)/[(1-\rho)^2 K!K\mu],$$

(37)

We now consider the corresponding $M/G/K$ queue where $C$ is the characteristic coefficient of the general service time. For heavy load, that is, when $V_Z \to 1$, the waiting time $W_{Z,C}$ approaches that of the $M/M/K$ queue. On the other hand, when $V_Z \to 0$, it holds that $W_{Z,C} \approx W_{\infty,C}$ and its mean

$$E[W_{\infty,C}] \approx \frac{1 + (C/K)^2}{2} E[W].$$

(39)

Subsequently, we consider that for any value of $Z$, when $V_Z \to 0$, $W_{Z,C} \approx F_C W_Z$, where $F_C = [1 + (C/K)^2]/2$ and when $V_Z \to 1$, $W_{Z,C} \approx W_Z$. Thus,

$$E[W^*_Z,C] \approx [V_Z + (1-V_Z) F_C] E[W^*_Z], \forall i \in Z.$$

(40)

REFERENCES


Abstract—New classes of non-volatile random access memory technology have the potential to unify the memory and the storage layers of modern computers, both for end-users and for data centers. The emerging non-volatile memory technologies have (or are expected to have) a speed close to the speed of current DRAM and are as DRAM byte-addressable, but they have the capacity and low cost of storage technologies. Unlike DRAM, they are non-volatile and they draw energy only for active reading and writing. Some, such as Phase Change Memory, also have a limited resilience, as each overwrite of a cell content can damage the cell removing its ability to store a bit faithfully. In a scenario where a non-volatile RAM replaces the current memory, bit-flip pressure becomes an important criteria for operating system and data structure design. As observed by Bittman and colleagues, replacing traditional data structures with ones where pointers are replaced by the exclusive-or of pointers can save bit-flips. We investigate this observation further and show that this is not the case for very large data structures or for data structures modified over long period of times. We then draw conclusions about virtual memory management in future operating systems optimized for the use of non-volatile random access memories.

I. INTRODUCTION

Emerging technologies for large Non-Volatile Random Access Memory (NVRAM) combine the speed and byte-addressability of Dynamic Random Access Memory (DRAM) with the capacity, the non-volatility, and the low costs of current storage technology. One NVRAM, Phase Change Memory (PCM) has already entered the market for several years, while others are in various phases moving towards commercialization. While based on very different physical effects, all these technologies have in common that energy is only consumed if data is read or written. Typically, the write operation uses up much more energy than a read operation. According to Bittman and colleagues [1], [2], a busy PCM memory can use more energy than a DRAM of the same size. Some of these technologies such as PCM have a limited endurance, in that write operations are slightly destructive. As long as we can distribute bit-flip pressure over a large memory and avoid hot-spots, the $10^8$ overwrites of a cell are more than sufficient for use as a main memory.

Whether cell endurance is a problem or not, if we are to use NVRAM as a replacement of DRAM, it behooves us to consider bit-flip pressure not only at the device level but also at the systems and data structures design level. Bittman and colleagues [1], [2] have argued this point persuasively. They also propose to store the exclusive-or of pointers instead of the pointers themselves and report considerable savings in bitflips at the cost of increased reads. Here, we investigate the success and limitations of this technique.

If two objects of the same size are allocated on the heap in close succession, the memory allocator will place them usually in close proximity. As their addresses do not differ by much, the exclusive-or (XOR in the following) of their addresses tends to have very few bits set. This is not the case if the objects have been allocated at different times.

We can show this with a simple experiment. We allocate a large number of nodes in a linked list consisting of a pointer to the next node, a pointer to a string and various meta-data. We are interested in the number of bit-flips resulting from overwriting the next pointer if the data structure is updated. We allocate a new node and insert it into a random location in the list. This will cause the pointer to the next node in the preceeding node to change. We measure the number of bit-flips in this node. The results are presented in Fig. 1. Not surprisingly, the number of bit-flips increases with the length of the list, which is of course the number of objects already allocated. As the x-axis in Fig. 1 uses a log-scale, it shows that the average number of bit-flips depends logarithmically on the number of objects. Interestingly, the number of bitflips depends on also on the size of the object. We can explain this phenomenon by observing that adding with a larger addend results in more bits potentially affected and therefore in more bit-flips.

In the remainder, we first explain our basic experiments that convince us that allocations can indeed be modeled using an arithmetic progression. We then use a linear model to predict...
the number of bit-flips when overwriting pointers. As heap allocation for NVRAM main memory needs to be redesigned, we draw the conclusion that the size of the allocation unit also matters. This concludes this paper.

II. METHODOLOGY

Singly and doubly-linked lists are a mainstay of modern data structures and algorithms. Bittman et al. proposed to replace doubly-linked lists with XOR lists. Such a list replaces the forward and backward pointer with a single field, the XOR-link containing the exclusive-or (XOR) of the addresses of the nodes to the left and to the right of the current node [7]. This link still allows to move forwards or backwards in the list, but now requires the address of the preceding node to obtain the address of the following node and the address of the succeeding node to obtain the address of the prior node.

Bittman and colleagues observed a reduction of bit-flips at the cost of additional bits read, when using the XOR list instead of the standard double linked list, but did not share the details of their experiment in their write-up. They also propose to extend the technique to other data structures. For instance, pointers in a tree node can be stored as exclusive-ors with other pointers [1], [2].

An XOR-list has disadvantages. Insertion or deletion of a node in an XOR list involves changes to the xor-link at the left and right neighbor of the insertion / deletion point. It is unlikely that insertion or deletion can be implemented with a single Compare-And-Swap (CAS) instruction and therefore lend itself to a thread-safe and lock-free implementation. This is also true if we try the same stratagem for a single linked list and have the next field store the XOR of address of the next node and the address of the current node.

For our investigation, we assume that the double-linked list undergoes many insertions in random locations. For instance, the double-linked list could be used to implement a large Linear Hash table [4]. A Linear Hash table is a key-value store that places records into buckets based on the hash of the key. As the number of records increases, so does the number of buckets. By collecting all records into a linked list in a specific order, a bucket will consist of a contiguous subset of nodes. The bucket organisation allows constant time access to records for the key-based operations of creation, update, read, and deletion. As the Linear Hash table grows, nodes will be inserted at specific locations inside the linked list. This organization is thread-safe and can be made lock-free if we use a single-linked list [6].

Under these assumptions, the neighbors of a given node will appear to be randomly selected among all allocated nodes. As the structure becomes large, the addresses of the pointers will start to diverge and we should observe a steady diversification of the pointer values. As a result, updates to the xor-link values will have more and more bit-flips. Instead of the time consuming emulation of the updates, we gather data on the values of the xor-links and compare them with the alternative, namely a next link field. A node consists of a key-field (unsigned long int), a pointer to the value of the record, and the link, either an xor-link or a next-field. Note that the vast majority of bit-flips would result from the creation of the records, but these would occur independent of the data structure.

III. MEASUREMENTS

To understand the allocation of objects in the heap, we first look at which bits are being set in the addresses of objects allocated in the heap. We allocate $N$ objects on the heap and observe the bits set in their addresses, incrementing $N$ in a geometric progression from 100 to 1,000,000. Figures 2, 3, 4 show the result for a MacOS 11.4 on a Macbook Pro, a Windows 10 i7 machine, and a Linux Xenon server with 32 cores. On the right, we show the bits set in the addresses. Since allocation respects word boundaries, all addresses are divisible by 16, implying that the four least significant bits are never set. If the objects are nodes of a singly or doubly linked list, these addresses (with exception of the last in the case of a singly linked list) will be the values of the next and – if existent – the previous node link field. On the left, we show the bits set in the xor-link for an xor-list. Bits in the addresses proper that are always set or never set have zero frequency in the xor-links as the exclusive-or operation will set them to zero. This is not the case for the first and for the last xor-link as these are the xor of the null pointer with the address of the second or second-last node. We can observe that as $N$ increases, these exceptions become invisible. For smaller $N$, quite a number of bits are set in all the addresses (see the full bar at bit 21 for $N \leq 10,000$ in the Windows 10
As \( N \) increases, this is less and less the case. For all values of \( N \), the set of “active bits”, that is, bits set with a frequency between 15% and 85%, slowly increases. Visually, the left and right graphs in Figures 2, 3, 4 come closer and closer, with the effect more subdued in the case of MacOS.

In general, assume that a certain bit is set with probability \( p \) in the addresses of the objects. When we form the exclusive-or of two addresses, the corresponding bit in the result is set with probability \( p(1-p) + (1-p)p = 2p - p^2 \). We show the corresponding function in Figure 5. The results depend on two assumptions: (1) The allocated objects appear in the xor-list in random order. (2) The only objects allocated on the heap are the nodes of the list. If Assumption (2) is violated, then the active sets on both sides of Figures 2 and 3 will be spread out. If Assumption (1) is violated, the active set for the xor-links are smaller.

From this experiment, we derive the importance of the sizes of the set of bits set and of the active set. We repeated our experiment of allocating nodes in either a singly linked list or an xor list. We then measured the size using three systems, a MacOS 11.4 on a Macbook Pro, a Windows 10 i7 laptop, and a CentOS (Linux) 7 server with 32 cores. In order to make comparisons easier between the average size of the active set and the average number of bits set, we multiplied the latter by two. We give the results in Figure 6. The figure contains error bars for the 95% confidence interval, but the bars are too small to be easily visible.

Only the number of bits set in the next fields, i.e. the raw pointer values, are somewhat constant. Both active set size and twice the number of bits set in the xor-links fall in a relatively small linear band in Fig. 6. As the x-scale is logarithmic, this means that active set size grows logarithmically in the number of objects allocated. We see that the behavior is quite constant for different Operating Systems and C++ compilers. The reason for the next-fields seemingly odd behavior is just setting of some bits in the base address of the allocation unit.

To understand the dependence on the size of the objects, we created \( N \) objects of three different sizes. We then measured the average number of bit-flips when overwriting a pointer to one object to a pointer to another object in the same series (Fig. 7). The experiment shows some jitter for small numbers \( N \), but otherwise relatively smooth behavior. We therefore fitted a linear model that predicts the average number \( \beta \) of bit-flips when overwriting a pointer to one object by a pointer to another object in the allocated list from the logarithm of the number of allocated objects. With other words, we try to
obtain a linear relationship
\[ \beta \sim \alpha_0 + \alpha_1 \log(N). \]

While the linear model is not a good predictor for small values of \( N \), it is a rather good one if we restrict ourselves to \( N > 25 \). Indeed, the \( R^2 \) value (giving the amount of statistical variation explained by the regressor \( \log(N) \)) is 0.99. In Figure 7, we show the experimental results as solid lines and the predicted value from the linear model as dotted lines. The linear coefficients (0.50, 0.52, 0.54) are just a tad above 1/2, as should be expected, as doubling the number of objects requires one more bit in the active set in an ideal situation.

IV. MODEL

A program that uses the heap to allocate objects of the same size will tend to have them allocated at addresses that form an arithmetic progression. This seems to be the ideal situation to reduce bit-flip pressure. In addition, the allocation schemes we have seen align objects to 16B boundaries in order to store an object in as few cache lines as possible.

We would like to predict the number of bit-flips for the “ideal” situation, where objects are allocated at addresses forming an arithmetic progression. Since an exact formula seems to be out of reach, we use a linear model to predict the number of bit-flips between two successive members of an arithmetic progression, given by the base \( B \), the size of a single object \( S \), and the number \( N \) of objects allocated. We assume that the size of the object is a multiple of 16B and denote the multiplier as \( s \). The number of bits set in \( s \) is given by its Hamming weight \( h(s) \), which incidentally is the weight of \( S \) as well since the bit pattern of \( S \) is that of \( s \) shifted by four. The number of elements \( N \) does not seem to enter directly, so we also consider the logarithm base 2 of \( N \) and the square of \( N \).

A first set of numbers was generated with reasonable distributions for \( s \) and \( N \) with a fixed base address \( B = 2^{16} \).
Figure 8 gives the scatter plot of bit-flips in this case (with half a million data points). As we can see, no simple functional relationship can be expected, which justifies our seeking an approximation using a linear model. As regressors we use \( s \), the weight of \( s \) (called weights in Fig. 8), \( N \) (num), \( \log_2(N) \) (lognum), and \( \sqrt{N} \) (sqrt). We also added the weight of \( N \) on the off-chance of this being important. Of course, the weight of \( N \) heavily depends on the size of \( N \). Our sets of regressors is a bad one for regression analysis as there are strong correlations between \( s \) and \( w(s) \) on one hand and all regressors derived from \( N \). Our goal was to capture most of the relationship using as few regressors as possible. We start out using all the regressors. We obtain a model with an \( R^2 \) value of 0.726. We then assessed the importance of each regressor by calculating the standardized regression coefficient [3], Section 3.3.3, which is the regression coefficient divided by the standard deviation of the dependent variable (bit-flips) and multiplied with the standard deviation of the regressor. This favors using \( w(s) \) and \( \sqrt{N} \). A more standard way looks at the \( R^2 \) value of all combinations of regressors. The \( R^2 \) value decreases when regressions are de-selected and we are hence looking for eliminating regressors with the smallest decrease in \( R^2 \). After calculating \( R^2 \) for all possible combinations of two and more regressors, we decided that our best chance was to dish the square-root in favor of the logarithm and obtain

\[
-0.071 + 0.639w(s) + 0.550\log_2(N) \quad (R^2 = 0.659)
\]

as the best linear approximation of the number of bit-flips.

We then repeated our experiment using variable bases generating a data set of 143654 records. We introduced as additional regressors \( B \) and \( w(B) \). As a result, our overall fit shrank (\( R^2 = 0.546 \)), but repeating the same elimination process, neither \( B \) nor \( w(B) \) had high standardized regression coefficients, the standardized regression coefficient for \( \sqrt{N} \) became also insignificant, and the \( R^2 \) analysis lead again to use only \( w(s) \) and \( \log_2(N) \) as regressors. Our linear approximation for the number of bit-flips between addresses of neighboring elements in the arithmetic progression is now

\[
2.234 + 0.551w(s) + 0.421\log_2(N) \quad (R^2 = 0.546)
\]

Using smaller data sets, these numbers did not change noticeably.

While this approximation is rought, it nevertheless allows us to draw an important conclusion that the weight of the allocation size is important. With other words, using additional padding in order to round up \( s \) to a “round” binary number will save on the number of bitflips. Ideally, the allocation size is a power of two. Even though our approximation formulae did not incorporate the base, comparing between the two types of data sets argues in favor of choosing \( B \) as a round binary number, ideally a power of two.

The dependence on the logarithm of \( N \) is fairly obvious. \( \log_2(N) \) is the maximum number of bits that can be set. The dependence on the Hamming weight of \( s \) can be understood through the mechanism of adding with carry. The more bits of \( s \) are set, the larger the potential for bit-flips between a number \( B \) and a number \( B + s \). This observation extends also to the number of bit-flips between \( B \) and \( B + i \cdot s \).

V. CONSEQUENCES FOR NVRAM MEMORY ALLOCATION

We are now able to understand Bittman’s et. al. observation about bit-flips in an xor-list and resulting recommendation for its use over a double-linked list. Our investigation has not convinced us that the savings obtained result from the savings of overwriting xor-links instead of overwriting next-links. Where the xor-list has an advantage is in overwriting a zeroed memory, as fewer bits are set in an xor-link.

However, zeroing memory before use is a frequent programmer ploy to protect against corrupted data. Security concerns in a multi-user system would compel us to overwrite memory before it is allocated to some user thread. As more bits are not set than set, zeroing memory before allocation seems to be the best strategy. If we zero memory before allocation, then using exclusive-or of pointers instead of raw pointers into the heap has indeed the potential to save on bit-flips. This happens first when we zero the memory as there are fewer bits to overwrite and second when memory is returned into the heap and the fewer bits set are reset to zero. Whether there are alternatives to zeroing out memory allocated on the heap is an interesting research question that should be addressed in the future. The permanence of data in a NVRAM would have privacy concerns mitigate against leaving data unaltered, but partial overwrites might suffice.

The use of exclusive-or of pointers should be open to adapting lock-free algorithms on data structures with a single link such as singly linked lists. If we develop data structures
like this, we would need to change the allocation algorithm of the heap. NVRAM memory allocation for NVRAM vis à vis DRAM needs already some adaptations. Schwalb and colleagues lay some out in order to guarantee durability and provide the capacity to recover from a crash in face of a hierarchy of different caches with their own write-back policies and write reordering [5]. They recommend to have pointers to the beginning of a data structure at a well-defined location. As we observed, bit-flip pressure prediction for pointer overwrites is simpler and better if addresses form an arithmetic progression as much as possible. We can therefore take Schwalb’s idea a bit further by allocating not only based on size, but also based on the data structure for which we allocate.

We have also seen that savings can be had by reducing the Hamming weight of the allocation unit. Since NVRAM memories can be big, only loosely fitting objects into a large envelope of allocated memory is no longer a great disadvantage. If endurance is an issue, then placing objects into the beginning of a larger envelope chunk of memory leads to uneven distribution of bit-flips. We can avoid this by shifting objects towards the end of the envelope by a fixed amount, that varies between program executions. For example, assume that we allocate objects of size 80B. We allocate them in dynamic memory in envelope chunks of 128B. We can start the object at an additional offset of 16B, 32B and 48B, where this additional offset does not appear as part of the address of the object. This will alleviate the distribution of bit-flips when the objects change.

VI. Conclusion

We investigated bit-flips for an important data structure based on a large linked list. We found that only for small to moderate sizes, using an xor-link instead of a next-link relieves bit-pressure when updating this data structure. On the positive side, our experiments indicate that allocating many objects of the same size results in pointers to these objects allocated in an arithmetic progression. Our linear models for bit-flips when replacing addresses of one member of an arithmetic progression by another indicate that bit-flips can be saved by reducing the Hamming weight of the allocation size. There is much more work to be done if we are to come up with strategies of saving bit-flips through bit-flip aware data structures. For instance, is it possible to reduce bit-flips by using arrays, where the address of the next element is not stored, but calculated?

Current computers are the result of a chaotic process where improvements of component technologies and in computer organization co-evolve. With new NVRAM technology, we as a community have a chance to come up with a simpler but better standard architecture, but doing so requires a rethinking of many aspects of computing. Bittman’s challenge to design data structures for NVRAM as main memory has to be taken up.

References

Understanding Energy Efficiency of Databases on Single Board Computers for Edge Computing

Jian Liu
Computer Science & Engineering
Louisiana State University
jliu@csc.lsu.edu

Kefei Wang
Computer Science & Engineering
Louisiana State University
kwang@csc.lsu.edu

Feng Chen
Computer Science & Engineering
Louisiana State University
fchen@csc.lsu.edu

Abstract—With the rapid advancement in edge computing, a recent trend is to migrate data processing from data centers to the edge to avoid long data transmission latencies. Databases, as an indispensable component, play a crucial role in efficient data management on edge devices. However, a critical limitation of edge devices is the highly constrained energy resource. Databases often incur a heavy load of CPU and storage I/O activities, which raises a particular concern on power-constrained platforms. In this paper, we have conducted an experimental study on the energy consumption of three representative databases, namely SQLite, LevelDB, and MongoDB, on single board computers for edge computing. We find that by deploying an appropriate database according to specific scenarios, the energy consumption can be reduced by a factor of 58.3, and the bandwidth can be improved by a factor of 54.4. Based on our experimental results, we also present several important system implications associated with our findings. We hope our first-hand data and the obtained insight can provide useful guidance for database and edge system designers and practitioners to develop and deploy energy-efficient databases for edge computing.

Index Terms—Energy efficiency; Single board computer; Database; Edge computing.

I. INTRODUCTION

With the rapid advancement of edge computing technology, a recent trend is to migrate data processing from data centers to the edge of the Internet [1]–[7]. Such a practice brings several benefits. For example, excessively long-latency accesses to the remote data center can be avoided, the traffic over the network can be significantly reduced, and the risk of leaking user data can also be minimized for improved data privacy.

Databases, as a key component in data management, play a crucial role in data processing on the edge for various applications, such as image recognition, augmented reality, virtual reality, autonomous vehicles, and many others [8]–[11]. Traditionally, the design of database systems focuses on performance optimizations, such as increasing throughput and reducing latency. On edge systems, a unique and must-have consideration is energy efficiency. Edge devices are often deployed in environments with unstable or limited power sources. Some systems are powered solely by battery or by a reproducible but constrained energy source, such as solar power. In such environments, energy is an extremely scarce resource but crucial to the normal operations of devices in the wild. A critical challenge of deploying databases on edge systems is the associated energy constraint.

Databases on edge systems have several unique issues. First, most edge systems adopt an ARM-based architecture, which has distinct power properties compared to traditional x86-based architectures. Second, due to space constraint, edge systems typically rely on flash storage, such as Secure Digital (SD) and eMMC cards, while conventional servers normally adopt hard disk drives as storage. The two different storage technologies carry very distinct characteristics in terms of both performance and energy efficiency. Finally, edge systems have limited resources in almost all aspects, such as computing power, memory capacity, and I/O bandwidth, etc. All these distinctions demand a systematic study on the power implications of deploying databases on edge devices.

Unfortunately, it still remains unclear how databases could affect energy consumption on a typical edge device, not to mention a thorough understanding on the associated system implications. In this paper, we select SQLite [12], LevelDB [13], and MongoDB [14], three representative relational and non-relational databases for our studies. We deploy these database systems on three popular low-power and compact-size Single Board Computers (SBCs), namely Raspberry Pi (RPi) 3 and 4 [15], and ODROID C2 [16] boards. In order to benchmark the devices coupled with databases, we create a set of representative workloads using Yahoo! Cloud Serving Benchmark (YCSB) [17] with different workload patterns. Through extensive experimental studies, we have obtained the first-hand results by deploying three typical databases on various edge platforms with distinct capabilities. Our purpose is to study the impact of workloads, device hardware, and databases on the energy consumption of edge systems. In particular, we desire to answer the following three important questions through our experimental studies.

- **Question #1**: Workload patterns (e.g., object size distribution, read and write ratio, etc.) can have different effects on database activities. What is the impact of workload patterns on energy consumption of databases?
- **Question #2**: Edge systems have different capabilities in terms of computing, memory, and storage, and the energy consumption characteristics are inherently distinct. What is the impact of edge platforms on energy consumption?
- **Question #3**: Database performance may not always be linearly consistent with energy consumption. What is the
relationship between performance and energy consumption when handling different database workloads?

Addressing the above questions can provide us important insight on energy consumption of deploying databases on edge devices. To the best of our knowledge, this paper is the first work focusing on energy consumption of modern databases on edge systems. We hope that the findings and insight presented in this paper can provide valuable guidance for system designers and practitioners to develop and deploy energy-efficient databases for edge computing.

The rest of the paper is organized as follows. Section II introduces the background of this work. Section III describes the methodology and experimental setup for the evaluation. Section IV presents the experimental results. Section V discusses the future work. Section VI gives the related work. The final section concludes this paper.

II. BACKGROUND

A. Databases

SQLite [12] is a light-weight but full-featured embedded SQL database engine, which has been widely deployed in mobile devices such as smartphone, tablet, etc. SQLite manages data in the form of tables based on the rigid relational data model, and it supports complex query statements within one table or across multiple tables. In addition, a complete SQLite database is contained in a single database file, which provides the desirable cross-platform flexibility.

LevelDB [13] is a high-performance NoSQL database developed by Google, which can be used for serving different edge applications. LevelDB is a light-weight database system with only 350-KB library size, favoring resource-scarce edge devices. Meanwhile, it is an embedded database with no client-server model support, making it a better fit in the small-form-factor devices. LevelDB has also been selected as one of the best databases for the edge by AZ Big Media [19].

MongoDB [14] is a light-weight, document-oriented NoSQL database. It is also becoming an important platform for intelligent edge [20]. Unlike LevelDB and SQLite, it uses JavaScript Object Notation (JSON) [21] API for data exchange and Binary JSON (BSON) [22] for data storage. The flexible data structure enables developers to use the same data model and syntax from the edge to the data center for fast application development.

B. Single Board Computers

With the increasing popularity of edge computing, the low-power ARM architecture has been widely adopted in a variety of edge devices (e.g., ODROID, RPi, smartphone, tablet). Compared with x86-based architectures, which are widely used in conventional server systems, the ARM architecture can significantly improve power efficiency, though at the cost of performance to some extent.

Single board computers, such as RPi and ODROID boards, are being widely used in various areas, such as education, home automation, industrial automation, etc. With customized Linux systems (e.g., Ubuntu, Raspbian), it is friendly to Linux practitioners without a steep learning curve. More recently, in order to enhance the usability and computing power of the edge devices, 64-bit devices (e.g., RPi 3 B+, RPi 4, ODROID C2) have been developed to replace the original 32-bit devices (e.g., RPi 2, ODROID C1). The computing power is increased significantly (e.g., the 64-bit ARMv8 Cortex-A72 adopted in RPi 4). More details about the SBCs used in this study are shown in Table I.

III. METHODOLOGY AND EXPERIMENTAL SETUP

A. Single Board Computers

In order to understand energy consumption with different hardware platforms and edge devices, we perform our experimental studies on a diverse selection of representative Single Board Computers (SBCs) and storage devices. We select three SBC platforms, namely ODROID C2 (OC2), RPi 3 model B+, and RPi 4 model B, covering hardware from different vendors and various generations of products of the same product family. All SBCs are equipped with the identical 64-GB SanDisk Ultra microSDXC card. A high-speed 64-GB eMMC v5.0 flash module is used in OC2 through the on-board eMMC module socket. A 256-GB Samsung 850 PRO SSD is connected to RPi 4 through the USB interface. Since RPi 4 provides two types of USB interfaces (USB 2.0 and 3.0), which have very different data transfer speeds, we also compare energy consumption of SSDs based on the two USB interfaces. In this way, we can make a comprehensive comparison on energy consumption between different platforms and storage devices under the same condition.

In our experiments, we install Ubuntu 18.04 Linux on the three SBCs. Although it only supports up to 1-GB memory on RPi4 [23], it enables us to compare the three devices consistently with the same operating system. All the SD cards, eMMC module, and SSD use Ext4 file system. Unless otherwise specified, we use default setting for the other parameters. We deploy three database systems on each SBC, namely SQLite 3.22.0, LevelDB 1.1, and MongoDB 3.6.6. In order to minimize the interference from other components, we disconnect all the peripheral devices (e.g., monitor, keyboard, mouse) from the main board. The wireless communications (e.g., Wi-Fi, Bluetooth) are also disabled during the experiments.

B. Benchmark Tools

In order to make the performance comparison between different databases with various workloads, we use Yahoo! Cloud Serving Benchmark (YCSB) [17] to generate three workload distributions, Zipfian, Latest, and Uniform. The Zipfian workloads access items according to the Zipfian distribution, where some items are frequently accessed while the others are not; the Latest workloads are similar to Zipfian workloads except that the most recently inserted items are more popular; the Uniform workloads access items randomly.

Since YCSB works on JAVA virtual machine, its data loading process demands a large capacity of memory. However, some edge devices are equipped with very limited memory (e.g., 1 GB), which can cause the data loading phase to be
TABLE I: Device hardware specifications of the single board computers.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Chipset</td>
<td>Amlogic S905</td>
<td>Broadcom BCM2837B0</td>
<td>Broadcom BCM2711</td>
</tr>
<tr>
<td>CPU</td>
<td>Quad Core 1.5GHz, 64-bit ARMv8 Cortex-A53</td>
<td>Quad Core 1.4GHz, 64-bit ARMv8 Cortex-A53</td>
<td>Quad Core 1.5GHz, 64-bit ARMv8 Cortex-A72</td>
</tr>
<tr>
<td>Memory</td>
<td>2GB DDR3</td>
<td>1GB LPDDR2</td>
<td>2GB LPDDR4</td>
</tr>
<tr>
<td>Flash Storage</td>
<td>MicroSD, eMMC5.0</td>
<td>MicroSD</td>
<td>MicroSD</td>
</tr>
<tr>
<td>Release Year</td>
<td>2016</td>
<td>2018</td>
<td>2019</td>
</tr>
<tr>
<td>Price</td>
<td>$40</td>
<td>$35</td>
<td>$35</td>
</tr>
</tbody>
</table>

TABLE II: Performance characteristics of storage devices.

<table>
<thead>
<tr>
<th>Storage Devices</th>
<th>Write (MB/s)</th>
<th>Read (MB/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SD card</td>
<td>14.1</td>
<td>41.3</td>
</tr>
<tr>
<td>eMMC</td>
<td>128</td>
<td>140</td>
</tr>
<tr>
<td>SSD with USB 2.0</td>
<td>33.3</td>
<td>35.7</td>
</tr>
<tr>
<td>SSD with USB 3.0</td>
<td>153</td>
<td>224</td>
</tr>
</tbody>
</table>

Note: Data is measured using Linux dd command with 4-KB blocks.

interrupted in the middle. In order to deal with this issue, we have developed a simple data-loading tool (around 600 lines of C++ code in Linux) to directly load the data into the databases via the database interfaces. In particular, we first generate workloads using YCSB, and then use our tool to load data into the target database, which ensures that the memory demand would not exceed the available capacity on board. After data loading is completed, we start benchmarking the database by running the workloads as usual.

C. Testbed Setup

![Fig. 1: The Monsoon power monitor and ODROID C2 board.](image)

We use Monsoon Mobile Device Power Monitor [24] coupled with PowerTool 4.0.5.2 to measure the power and energy consumption of the SBC boards. Specifically, the SBC under test is connected to the USB-Out channel on the power monitor, which provides power for the normal run of the SBC. Another USB-In channel of the power monitor is connected to a DC power supply. The USB-In and USB-Out form a closed circuit, so that the power monitor can intercept and monitor the current variations, thereby providing the real-time power consumption data based on the current and voltage.

D. Measurement Metrics and Settings

Our measurement focuses on energy consumption, which is the amount of energy consumed to complete a task (in units of Joules), and power consumption, which is the energy consumption per unit time (in units of Watts). We also measure bandwidth and latency, two key performance metrics for databases, and study the relationship between energy consumption and database performance.

Our main experimental studies use the following settings for workloads, hardware platforms, and databases.

- **Workloads**: The Zipfian workload is chosen as a representative workload for the experiments. The total dataset size is around 4 GB. Item size is set to 1 KB. The total number of requests issued to the database reaches 2 millions, and the total amount of accessed data is about 2 GB. In addition, we run both read-intensive and write-intensive workloads respectively in each experiment for evaluating the database. For fair comparison, we flush the operating system page cache before each run. In our studies on the impact of workloads, we also include Latest and Uniform as additional workloads for studies. Specific settings can be found in Section IV-B.

- **Hardware platforms**: OC2 and RPi4 are selected as two representative edge devices. They are both equipped with 64-GB SanDisk SD cards. To fully exercise the quad-core processors on the devices, we use 4 threads to run most workloads. In the experiment of running the write-intensive workload on SQLite, we find that concurrent writes to SQLite can incur errors (e.g., an error signal SQLITE_BUSY is raised due to integrity issues), thus we use only one thread in this test. In our studies on the impact of hardware platform, we also include RPi3 as additional hardware for comparison. Specific settings are available in Section IV-C.

- **Databases**: In our experiments, the three databases, SQLite, LevelDB, and MongoDB run on the Ext4 file system. For all the three databases, we use the asynchronous I/O mode for better performance. Unless otherwise specified, the other parameters of databases use the default settings.

IV. EXPERIMENTAL RESULTS

In this section, we present our experimental results and give a comprehensive analysis. We desire to seek answers in depth to the three key questions as mentioned previously. We will first give an overall comparison between the three database systems, and then present detailed experimental analysis from aspects of workloads, platforms, storage devices to obtain more insight on the effect of each individual component.

A. Overall Comparison

Figure 2 shows that SQLite achieves the best performance among the three databases for both read- and write-intensive workloads across the two platforms, OC2 and RPi4. For example, SQLite improves the bandwidth by a factor of 2.5 and 10.4 and reduces the latency by 60% and 90.4%, compared
with MongoDB and LevelDB under read-intensive workloads on OC2. To explain the results, we have examined the CPU and disk usage to identify the bottleneck. We find that the CPU usage of MongoDB is about 43.9%. Although its is 38 percentage point (p.p) higher than that of SQLite, it is far from overloading the CPU. The main bottleneck is the limited storage I/O bandwidth on the SBCs (only dozens of megabytes per second). Unfortunately, both MongoDB and LevelDB generate more storage I/Os in terms of data amount than SQLite by a factor of 7.6 and 32.3, respectively.

We can also see that performance dominates the total energy consumption when the difference of power consumption is insignificant. Since SQLite achieves the highest bandwidth across different platforms, it takes the least time to complete the same workload. As such, the comparatively less execution time enables SQLite to consume the least energy. For example, although the power consumption of SQLite is 4.9% higher than that of LevelDB under read-intensive workloads on OC2, the significantly lower bandwidth of LevelDB makes it take longer time to complete the same task, which in turn causes it to consume more energy than SQLite by a factor of 9.9.

**Observation #1:** Generally, SQLite is the best choice if the specific information about the platform, storage medium, workload, etc. is unknown, due to its notably higher performance. Improving database performance is an effective method to reduce the energy consumption.

### B. Effect of Workloads

Workloads can have a significant impact on energy consumption and performance of the databases. In this section, we benchmark the databases by adjusting the access patterns, read and write ratios, and key-value sizes to study the effect of workloads on databases. In the figures shown in this section, the bars represent the data for the left axis, and the points represent the data for the right axis.

1) **Access pattern:** We run three workload distributions in this test. As shown in Figure 3, the three databases in general achieve higher bandwidth and lower latency under Latest workload than the other two workloads on both OC2 and RPi4. For example, under read-intensive workloads on OC2, compared with Zipfian and Uniform workloads, the Latest workload with SQLite database achieves a higher bandwidth by a factor of 2.2 and 4.1, respectively. Meanwhile the latency is 54.1% and 75.6% lower, respectively. It is mainly because the Latest workload shows a stronger locality and more data accesses can be absorbed in memory. In the other two workloads, more storage I/Os are incurred due to the weaker locality. In fact, the total I/O amounts of SQLite under Zipfian and Uniform workloads are 2.2 times and 3.9 times of that running under Latest read-intensive workload on OC2.

2) **Read and write ratio:** Workloads with different amounts of read and write requests can impact the database behaviors, and consequently the performance and energy consumption. We configure YCSB to generate three representative workloads with different read and write ratios as read-intensive (R/W: 100/0), moderate (R/W: 50/50), and write-intensive (R/W: 0/100) using Zipfian distribution.

As shown in Figure 4, we find that SQLite and MongoDB achieve better performance under read-intensive workload on both OC2 and RPi4. For example, on OC2, SQLite and MongoDB achieve higher bandwidths with read-intensive workload than with write-intensive workloads by a factor of 2.6 and 2.1, respectively. Unlike the other two databases, LevelDB adopts the LSM-tree based data structure, which makes its write bandwidth 22.2% higher than its read bandwidth, and the total I/O amount under the write-intensive workload is only 23.8% of that under the read-intensive workload on OC2.
3) Key-value size: Key-value size also has a critical impact on databases. We select three representative key-value sizes, namely 64 B, 1 KB, and 16 KB to make comparison in Figure 5. In order to compare the incurred data amount of I/Os, we ensure the workloads with different key-value sizes access the same amount of data. We can see that the bandwidth increases significantly with the growing item size, because the I/O bandwidth can be better utilized with a larger key-value size. Latency is affected by I/O amplification, especially for small key-values. For example, the latency of SQLite with 64-B key-value size under read-intensive workloads on OC2 is 91.3% larger than that with 1-KB key-value size. This result can be explained by the much higher I/O amount involved with 64-B key-value size, which is 30.1 times of that with 1-KB key-value size. Further increasing the key-value size reduces I/O amplification, but the latency increases due to the larger amount of data that needs to be transferred for each request.

Compared to the other two databases, LevelDB benefits from the write-friendly LSM-tree data structure and performs well at handling small items under the write-intensive workload. For example, LevelDB outperforms SQLite and MongoDB in bandwidth under the write-intensive workload with 64-B key-value by a factor of 31.5 and 54.4 on OC2, and the corresponding I/O amount of LevelDB is much smaller,
which is only 20.3% and 3.5% of that caused by the other two databases respectively. MongoDB fits better for handling large-size key-value items under read-intensive workload, in terms of both bandwidth and latency, mainly due to the much less incurred I/O amount. For example, compared with SQLite and LevelDB under read-intensive workloads with 16-KB key-value based on OC2, MongoDB improves the bandwidth by a factor of 2.4 and 23.1, respectively, meanwhile reducing the latency by 59% and 95.6%.

The significant performance difference is also reflected in energy consumption. For example, for the small-size (64 B) write-intensive workload, LevelDB outperforms MongoDB in terms of bandwidth by a factor of 54.4, and MongoDB’s energy consumption is 58.3 times of that with LevelDB.

**Observation #2:** LevelDB is more preferable to handle small-size write-intensive workloads, while MongoDB is more appropriate for working with large-size read-intensive workloads. The difference in power consumption caused by workloads has weak impact on the energy consumption compared to the significant performance gap.

### C. SBC Platforms and Storage Devices

To reflect the effect of different platforms and storage devices, we select three representative SBCs, namely ODROID C2, Raspberry Pi3 B+, and Raspberry Pi4 B, which cover the SBCs from different vendors as well as different generations of devices from the same vendor. We also use different flash storage devices, namely SD Card, eMMC, and SSD, to understand the effect of storage devices on databases. We use Zipfian workloads with 1-KB key-value size in this test.

1) **SBC platforms:** Figure 6 shows that OC2 achieves the best overall performance among the three platforms, while the performance of RPi3 is the worst. The key differentiating factor is the memory on device. OC2 is equipped with 2-GB memory on board, which allows it to serve more data requests from main memory rather than the slow SD card. For example, compared to running on RPi3, MongoDB generates 35.9% less I/O amount on OC2 under the read-intensive workload, outperforming RPi3 by a factor of 2.1 in bandwidth. Although the overall performance of RPi4 is relatively lower than OC2, it also outperforms RPi3 due to the faster memory (LPDDR4 vs. LPDDR2) and the better CPU (1.5GHz Cortex-A72 vs. 1.4GHz Cortex-A53).

2) **Storage devices:** Storage devices significantly affect the performance, as well as the power and energy consumption. We compare different databases running on SD, eMMC, and SSD. Since SSD with SATA interface cannot be directly connected to the SBC, we use a SATA/USB converter to connect the SSD and the SBC via the USB interface. RPi4 provides two different USB interfaces (USB 2.0 and 3.0), we benchmark the SSD using both interfaces.

By default, journaling of Ext4 is disabled for managing the system device (i.e., SD card). For fairness, we also evaluate the other attached devices including eMMC and SSD with journaling mechanism disabled. To further study the impact of journaling on energy efficiency of storage devices, we choose eMMC as an example to make comparison under two scenarios, with and without journaling. We use tune2fs tool in Linux to switch between the two journaling modes. We use the default journaling method (i.e., data=ordered) of Ext4, which only journals metadata operations [25].

In general, databases on SSD connected via the USB 3.0 interface achieve the highest performance, while databases on SD card have the lowest performance. For example, SQLite on SSD with USB 3.0 outperforms that on SD card by a factor of 2.5 in terms of bandwidth under the read-intensive workload.
The database performance on eMMC and SSD with USB 2.0 falls in the middle of the above-mentioned two. The overall performance is better on eMMC compared to that on SSD. For example, SQLite on eMMC improves the bandwidth by 31.6% compared with that on SSD with USB 2.0 under the read-intensive workload.

In addition, we also find that the default journaling method (i.e., data=ordered) has negligible impact on the performance and energy consumption, since it only journals the metadata. For example, SQLite based on eMMC without journaling improves the bandwidth only by 6.5% in contrast to that with journaling under write-intensive workload. Due to the space limit, it is not shown in the figure.

Since a relatively higher voltage is required, databases on SSD have much higher power consumption than running on eMMC. For example, the power consumption of SQLite on SSD with USB 3.0 is larger than that on eMMC under read-intensive workload by a factor of 2. Interestingly, although the database on SSD with USB 3.0 has a higher bandwidth and less running time compared to that running on eMMC, it has a larger energy consumption under the same situation. For example, SQLite running on SSD with USB 3.0 consumes 83% more energy than running on eMMC under read-intensive workload. It means that unlike our previous finding that performance largely determines energy consumption, power consumption is the deterministic factor in this experiment.

### Observation #3:
The hardware resources on device, especially memory and storage, have a notable impact on the database performance due to the critical I/O bottleneck. Although SSD with USB 3.0 provides higher performance, the eMMC flash storage is more energy efficient due to the lower power consumption.

---

**V. DISCUSSIONS**

Our experimental studies have provided a set of valuable findings and system implications. In the meantime, we also note that due to the resource constraints, several aspects are worth further studies in the future. (1) Our current studies do not further differentiate energy consumption by different components of the database. An interesting question is, which part of the database design (e.g., logging, DB engine, compaction, etc.) contributes the most to the energy consumption, and how to optimize it by tuning or redesigning the database. In this work, we mainly treat the database as a whole. Disassembling the database for analysis could gain more insight. (2) In this work, our experiments are performed on SBC boards. Such compact computing devices are highly versatile and can be adopted to handle various kinds of edge workloads. In some application environments, however, such general-purpose SBC boards may not be suitable for the target workloads, such as machine learning, pattern recognition on the edge, which demand more computational power and storage capacity. In our future work, we plan to expand the device selection to cover a broader scope of application scenarios.

**VI. RELATED WORK**

The limited energy resource is critical to edge devices. Prior research works have studied power and energy consumption on edge devices [26]–[33]. Khan et al. focus on the energy consumption of different data structures in edge computing. They find that the concurrent and locality-aware Delta tree outperforms B-link tree significantly in terms of energy efficiency and performance [28]. Chandra et al. propose three offloading energy-efficient approaches to improve the battery life of mobile devices, which includes using low-power CPU coexisting with the main CPU, using battery-backed RAM to reduce the flash I/Os, and offloading the computation-intensive
tasks to the nearby cloud [29]. Kaup et al. measure the relationship between power consumption and system resource utilization including CPU, Ethernet, Wi-Fi on Raspberry Pi (RPi), and further build a power model to improve the energy efficiency [27]. Bekaro et al. measure the power consumption of the predefined key functionalities (e.g., device start-up, downloading a file, etc.) across five different devices including RPi, smartphone, etc. [30] Ardito et al. use sysbench and iperf to measure the power consumption of CPU as well as the network adapter based on RPi, and use linear regression to model the power consumption [31].

Recently, energy consumption on storage component of edge devices attracts interests in the community. Prior works find that energy consumption on storage components is a significant contributor that should not be neglected [8]–[11]. Mohan et al. measure the energy consumption using different workloads on the SQLite database deployed on an Android smartphone, and they also analyze the impacts of various SQLite operations on the energy consumption [8]. Kim et al. develop an energy consumption model for I/O subsystem of wearable devices, thereby reducing energy consumption on I/O activities of SQLite database significantly [11]. Li et al. find that the energy consumption on storage software stack is 200 times more than the hardware through experiments on mobile platforms, and they also build a storage energy consumption model to better optimize energy utilization [10].

### VII. CONCLUSION

In this paper, we present a comprehensive experimental study to understand the energy efficiency of databases on single board computers for edge computing. We have made several important findings on the performance, power, and energy consumption of three representative databases on edge devices and also discussed the related system implications. We hope that this work can provide valuable guidance for system designers and practitioners to design and deploy databases in an energy-efficient way in edge computing environments.

### ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback and comments. This work was supported in part by U.S. National Science Foundation under Grant CCF-1910958.

### REFERENCES

A Multiple Snapshot Attack on Deniable Storage Systems

Kyle Fredrickson, Austen Barker, and Darrell D. E. Long
University of California, Santa Cruz

Abstract—While disk encryption is suitable for use in most situations where confidentiality of disks is required, stronger guarantees are required in situations where adversaries may employ coercive tactics to gain access to cryptographic keys. Deniable volumes are one such solution in which the security goal is to prevent an adversary from discovering that there is an encrypted volume. Multiple snapshot attacks, where an adversary is able to gain access to two or more images of a disk, have often been proposed in the deniable storage system literature; however, there have been no concrete attacks proposed or carried out. We present the first multiple snapshot attack, and we find that it is applicable to most, if not all, implemented deniable storage systems. Our attack leverages the pattern of consecutive block changes an adversary would have access to with two snapshots, and demonstrate that with high probability it detects moderately sized and large hidden volumes, while maintaining a low false positive rate.

Index Terms—Security, Steganography, Storage

I. INTRODUCTION

Conventional disk encryption has been very successful at ensuring the confidentiality and integrity of disks. In many applications reducing these properties to corresponding key management problems is sufficient to ensure security. However, when faced with adversaries capable of using coercive tactics to reveal keys, conventional encryption comes up short. In these situations the possession of an encrypted drive can be considered suspicious enough to justify coercive attacks euphemistically known as rubber hose attacks. These situations require that the mere presence of encrypted data be unknown to the adversary to avoid the use of coercive tactics. Deniable storage systems have been developed in response to this threat. These systems are used to create a hidden volume on the user’s device, the existence of which is plausibly deniable. One common strategy among deniable storage systems is to encrypt data and randomly write it throughout the disk’s free space. Assuming that the free space is filled with pseudorandom bytes, this renders hidden data indistinguishable from unallocated blocks. This achieves the goal of deniability when an adversary is restricted to viewing the disk at one point in time and there are no other sources of information leakage.

A multiple snapshot attack is an attack on a hidden volume where an adversary is able to gain access to a machine and make observations at two or more points in time. Information gained from comparing these snapshots would then be analyzed for abnormalities that could imply the existence of a hidden volume. In many circumstances where a deniable volume may be used multiple snapshot attacks are feasible. As an example, suppose a journalist is entering a repressive country with the intent to exfiltrate some data. The adversary first takes a snapshot of all devices entering the country, the journalist collects the data to be exfiltrated and constructs a hidden volume to hide it, then the adversary takes another snapshot when the device leaves the country. This leaves the pattern of changes on the file system as a new source of information leakage.

While multiple snapshot attacks on hidden volumes have been described in the literature and defenses against them proposed [1]–[4], to our knowledge there has never been a thoroughly described or attempted example of this class of attack. Among the reasons for this are not only the difficulty in obtaining enough disk images to establish what a normal pattern of changes is, but also the difficulty in identifying meaningful features in observed change patterns that would be invariant across normal use cases.

Using features derived from the change patterns on disks, we have defined and implemented the first multiple snapshot attack against deniable storage systems. The main contribution of our work is to affirm the relevance of multiple snapshot attacks to deniable storage systems by demonstrating a practical attack, while also identifying the limitations of our techniques. We propose analyzing the distribution on the lengths of consecutive block changes, which we call chains, as a new metric for quantifying disk behavior, and leverage this information to distinguish between disks containing hidden volumes and those that do not. Specifically we attack Artifice [5], a deniable storage system, but our attack if broadly applicable, and is able to reveal most, if not all, implemented deniable storage systems. In response to this attack we propose additional design requirements for deniable systems, and discuss the implications of these requirements on Artifice. Through this work we hope to guide the design and implementation of future systems, improving their security even against powerful adversaries with the ability to gain access to devices at several points in time.

We first give background on deniable storage systems, going into detail on Artifice, and describe past attacks on these systems. We then propose our attack, describe our data collection and simulation methods, and give results of the attack on our dataset. Finally, we describe mitigation schemes and conclude.

This research was supported in part by the National Science Foundation grant number IIP-1266400, award CNS-1814347, and by the industrial partners of the Center for Research in Storage Systems.
II. BACKGROUND AND RELATED WORK

As previously stated, the goal of a deniable storage system is to conceal the existence of a volume from an adversary that would otherwise view an encrypted volume as suspicious and consequently coerce the user to disclose their secret keys. With a hidden volume the user is able to reveal keys to known encrypted drives while allowing the user to deny that hidden volumes exist on the device.

Many deniable or steganographic storage systems have been proposed in past years. One of the key concerns of these systems has been defending against multiple snapshot attacks. In this section we will describe existing deniable storage systems and attacks against those systems.

A. Deniable and Steganographic Storage Systems

Anderson, et al. [6] were the first to propose a steganographic file system and described two possible approaches. The first approach consists of a set of cover files filled with random information, of which a subset are combined with hidden files using an additive secret sharing scheme. The second construction hides data within the unallocated space of another file system. Although the proposal lacked an implementation of the two ideas, most deniable storage systems follow the second approach.

McDonald and Kuhn implemented Anderson et al.’s second scheme as a Linux file system based on ext2 known as StegFS [7]. Although the scheme provides deniability when the adversary can only view the device once, McDonald et al.’s StegFS and similar systems like TrueCrypt [8] or Mobilflage [9] cannot provide a reasonable defense against an adversary that can view the disk multiple times and compare snapshots.

Pang, et al. [1], [10] implemented a variant of StegFS that attempts to defend against multiple snapshot adversaries by performing dummy operations to the disk that obscure hidden writes. Similar approaches to this have been implemented using derivatives of Oblivious RAM [11], [12] or similar techniques to render accesses to a hidden volume indistinguishable from accesses to the public volume [2]–[4]. Although these approaches render hidden and public writes indistinguishable, they incur other abnormalities such as fully random write patterns not exhibited by common file systems and significant performance losses that would betray the existence of a hidden volume or the ability to construct a hidden volume.

A perennial weakness of many of these systems is that they do not take measures to hide the existence of the software used to access the hidden volume, often claiming that widespread adoption would ensure the ability to create a hidden volume is not suspicious. The fact that disk encryption, though widespread, can still be seen as suspicious by many organizations calls this assumption into doubt.

B. Attacks on Steganographic Storage

Most existing storage systems assume that the adversary has the ability to closely analyze characteristics of the user’s device for any clues that the device contains a hidden volume. With respect to multiple snapshot analysis and related attacks, previous work [1], [2] has categorized adversary capabilities into three general categories.

- **Single snapshot** A single snapshot adversary can only view a device once. All existing deniable systems have some level of resilience against this sort of attack.
- **Multiple snapshot** In this case the adversary can view the device two or more times. These snapshots can be compared, and the changes analyzed for anomalies that could reveal a hidden volume on the user’s device. It is important to note that in this class of attack the adversary is only able to view static snapshots of the disk a limited number of times.
- **Continuous observation** This is when an adversary has the ability to continuously observe writes to the user’s device or make a snapshot of the device for each write. This adversary capability is sometimes also called *continuous traffic analysis*. This sort of attack would likely require a form of malware to be installed on the user’s device for the purposes of information gathering. OS level spyware would be able to monitor writes to the user's device.

Proposed attacks and serious attempts to break deniable storage systems are limited when compared to the variety of proposed systems. One vulnerability found by Czeskis et al. when analyzing TrueCrypt was its susceptibility to leaking information from the hidden to public volumes through operating system utilities and common applications such as word processors. Another work by Troncoso et al. describes and implements a continuous traffic analysis attack against Pang’s StegFS that exploits repeated write patterns that correspond to writes made to a hidden volume. While this attack shows the effectiveness of monitoring disk operations in finding a hidden volume, it is assumed that the adversary has enough power to continuously observe the disk. The multiple snapshot adversary is considerably weaker than the continuous observation adversary, yet there has been no previous work towards demonstrating a multiple snapshot attack against a deniable storage system.

C. Artifice

For the purposes of testing a multiple snapshot attack we will use Artifice as our deniable storage system. Artifice follows the common model of hiding information in unallocated blocks but with a few additional features that address some pitfalls of other existing file systems [5]. Most importantly it addresses problems with hiding its driver software and provides a layer of protection against malware and information leakage by putting the Artifice driver on a separate Linux live USB drive. To access a hidden volume, the user boots into an Artifice-aware OS contained on this drive instead of the normal public OS. This isolation does not leave behind suspicious drivers on the user’s machine and mitigates the impact of malware and information leakage.

Artifice writes data by splitting data blocks into pseudo-random *carrier blocks* using an information dispersal algorithm (IDA) such as Shamir Secret Sharing [13]. These carrier blocks
are then uniformly distributed throughout the unallocated space of the drive, which is assumed to be full of pseudorandom blocks due to a secure deletion utility or similarly deniable means. Since the public file system is not aware of Artifice, it is imperative to protect the carrier blocks from accidental overwrite. IDAs provide Artifice overwrite tolerance by writing redundant carrier blocks in excess of the number needed to normally reconstruct the data. This allows Artifice to carry out self-repair operations whenever accessed by the user and increases the probability that an Artifice instance will survive many writes made by the public file system.

Currently, Artifice aims to address the problem of multiple snapshot attacks through writing hidden blocks under the guise of a suitable deniable operation, such as defragmentation, where the contents of a disk are relocated to be contiguous, routine file deletion, or by operational security measures that render previous snapshots useless, such as reinstalling the public operating system or wiping the storage prior to constructing an Artifice instance.

III. ATTACK FRAMEWORK

The primary goal of an attack on a deniable storage system is not to recover the plaintext files, but to discover the presence of hidden volumes. In Artifice’s adversary model, it is assumed that the adversary can coerce a user to reveal their keys provided there is sufficient suspicion that a user possesses a hidden volume [5]. The success of a particular attack then is dependent on how well the attack can discriminate between disks containing hidden volumes and those that do not. In particular, the rates of false positives and false negatives should both be very low for an attack to be considered successful.

As noted most deniable storage systems achieve their aims by encrypting and splitting data into redundant shares and writing these blocks uniformly on the disk [5], [10]. This increases the odds of survival of the files when the public file system, which is unaware of their existence, makes its own writes. Other approaches write public and hidden data pseudorandomly so that public and hidden writes are indistinguishable [2], [3].

The commonality between these approaches is that deniable storage systems make many uniform writes. Depending on the size of the free space, writes made uniformly are very likely to result in isolated changes on disk, which we call singletons. This is in contrast to normal file systems, which do not make their writes uniformly, and are much more likely to make writes that are part of longer strings of consecutive changes that we call chains. We call a chain of \( c \) consecutive changes a \( c \)-chain.

**Example III.1.** Say that in a change record a 1 denotes a change and a 0 denotes no change between two snapshots. Then in \( (1) \) there are two singletons (or 1-chains) and one 3-chain.

\[
\begin{array}{cccccccc}
1 & 0 & 1 & 0 & 1 & 1 & 1
\end{array}
\]  

(1)

An adversary observing the change records produced from a pair of disk snapshots would be able to observe the lengths of the chains those changes produce. Crucially if the disk contains a hidden volume, the adversary would also see changes made by writes to that volume. In Appendix A we give the theoretical distribution of chains given the size of the disk and the number of changes made. We find they are distributed according to lemma A.1. Fig. 1 shows that as the free space grows relative to the number of writes, the probability of a singleton increases. Fig. 2 shows that for real disks, the probability of a singleton is much smaller, and the tail of the distribution is typically much heavier. Together they show the disparity between the distributions of chains due to a hidden volume and the chains due to a public file system. This becomes more pronounced as the hidden volume makes more writes. Our task then is to construct features to distinguish between the distribution of consecutive changes made by a public file system and changes made by a public file system and a hidden volume.
To carry out this attack, we assume that the attacker has access to a large set of disk images, both from disks that contain hidden volumes and from those that do not. Images in this set will be organized into pairs of images from the same disk at different points in time. Comparing these pairs of images results in a change record for a given disk. We will call pairs that do not have an instance of a hidden volume clean and those that do dirty. Assuming our adversary is well-funded and motivated these data requirements are easily attainable.

Since for our proposed attack we only need to determine whether individual blocks have changed, we can take snapshots of the clean and dirty disks in a space efficient manner by hashing each block on the disk, and constructing a Merkle tree [14] over the hashed blocks. This gives us a very efficient method of finding changes, and producing change records. These change records are further processed into lists of integers, $D_i$, recording the lengths of chains found in each change record. We will denote these lists of chains $\{D_i\} = D$.

From $D$ the adversary has several options for constructing an arbitrary $n$ number of features for use in a classification algorithm. The first is to remove a set of clean disks that we will call $C$. Using $C$ the adversary estimates the probability of $c$-changes from 1 to $n$, the number of features, for each $D_i$ by counting the occurrences of each $c$-chain and dividing by the total number of chains in $D_i$. Using these probabilities on $D - C$, the adversary can then estimate the probability of a disk containing more than $k$ $c$-chains with the cumulative distribution function (CDF) of the binomial distribution, $F(k; n, p_c)$ where $n$ is the length of the change record and $p_c$ is the estimated probability of a chain of $c$ consecutive changes. We will denote the event of an adversary observing $k$ consecutive changes of length $c$ as $X_c$. Then,

$$P(X_c > k) = 1 - F(k; n, p_c). \quad (2)$$

Using these values we construct our final features $F$ for $D$. This method has greater sensitivity to small variations in probabilities for small disks; however, on large disks it tends to underflow when computing equation (2). We give pseudocode for this in algorithm (1).

For large datasets, we take a simpler approach to computing $F$ by computing the probabilities of chains of length 1 to $n$ for each $D_i$ and feeding these to our classifier.

Recall that the dataset is entirely constructed by the adversary, so it has ground truth labels describing whether each row in $F$ corresponds to a disk containing a hidden volume or not. The adversary now trains a supervised classification algorithm on $F$ split into standard train and test sets. On new pairs of disks the adversary runs through the feature construction process, then runs the classification algorithm on those feature and responds accordingly. Furthermore, if the adversary can confirm that some disks did in fact contain a hidden volume, it can update its model using various online learning techniques [15], further refining its model.

As a note on selection of the number of features, we observe from lemma A.1 that when the free space on a disk is large relative to the number of writes, the vast majority of writes will result in singleton changes. As a consequence, the adversary could learn based on a single feature derived from singletons. This may be desirable in some situations for the sake of efficiency; however, Artifice could simply be modified so that when writing blocks they would be grouped in chains of two or more, thereby defeating the attack as described. The adversary can in turn thwart this countermeasure by increasing the number of features, $n$. We go into more detail regarding this problem in Section VI.

\begin{algorithm}
\textbf{Algorithm 1: Feature Construction.} This method is suitable for smaller disks, where greater sensitivity is required, but suffers from underflow on large disks.

\textbf{Input:} $D$, a set of processed disks; $\{p_1, ..., p_t\}$, the estimated probabilities of consecutive changes of length 1...$c$.

\textbf{Output:} $F$, a $|D| \times c$ matrix.

$F := \{\}$

\textbf{foreach} $D_i \in D$ \textbf{do}

$f := \{\}$

\textbf{foreach} $c_i \in \{1,...,c\}$ \textbf{do}

$k := \text{number of } c\text{-consecutive changes in } D_i$

$a := 1 - F(k; |D_i|, p_c)$

append($a, f$)

end

append($f, F$)

end

\textbf{return} $F$
\end{algorithm}

IV. DATA COLLECTION AND EXPERIMENT METHODOLOGY

As noted previously, one of the challenges of carrying out a multiple snapshot attack is the availability of pairs of disk images. These are necessary to learn what normal chains look like. While collecting hundreds, or thousands, of disk images may be feasible for a nation state level adversary (or a large IT department), we were unable to collect such a large amount of data.

Instead, we have collected several months worth of snapshots from a 1TB NVMe SSD formatted with ext4 and in use as the boot disk of a desktop computer running Ubuntu 18.04. We collected 53 snapshots in total, giving us 52 change records\(^1\) which we have made publicly available. By observing the distribution of lengths of changes over our collected data (Figure 2), and the theoretical distribution of consecutive changes (Figure 1) when changes are made uniformly, the potential strangeness of a disk running a deniable volume becomes clear.

Because just 52 data points are insufficient to train a classifier, and moreover, would be inconclusive regarding performance of that classifier, we instead used this data to generate a synthetic dataset on which to train and test our classifier. While in the real world different file systems may produce different patterns

\(^1\)This data is available at https://files.ssrc.us/data/disk-change-data.zip. The code for these experiments is available at https://github.com/ucsc-ssl/multiple-snapshot-attack.
of changes, this does not change the reality that a deniable volume writing many single blocks, and thereby causing many singleton chains in the change record, would be considered abnormal regardless of the public file system in use. Though the fact that our attack only utilizes data from ext4 file systems is a limitation, no file system in widespread use writes blocks randomly, so we expect our attack will generalize to other data sources.

Our experiments are conservative in terms of the operational security measures the user of a hidden volume might take. We discuss them here to motivate our experimental design.

There are several things that the user of a deniable volume could do to decrease the odds of detection. To start, assume that a single snapshot has been taken, and no deniable volume yet exists on the drive. A prudent user would make many changes to the disk through the public file system. The reason for this being that if there are overwhelmingly many chains distributed according to normal disk behavior, the singletons made by writing to the hidden volume could be made to look like noise. To illustrate this, consider a user that does not produce a single change through the public file system after the deniable volume is created and written to. In this case the adversary would see, after taking a second snapshot and computing the differences, only chains produced by the deniable volume. These chains would be principally singletons, which would surely be conspicuous. As an extreme measure the user could simply wipe the disk and then create the deniable volume, but this may be considered suspicious or may be undesirable for other reasons. As a less drastic alternative, the user could produce an overwhelming number of changes to the disk through the public file system. In our experiments we chose this middle ground by using our real data to simulate 25 GB worth of public changes on a 1 TB disk with 100 GB of free space. This produces a sufficient number of public changes to hide private changes while still behaving as a normal user might.

Each pair of disk snapshots can be regarded as producing a distribution over consecutive change lengths, so in order to construct our synthetic dataset we simply draw chains from these distributions. Realistically, the fraction of disks containing hidden volumes would be relatively small, and the size of hidden volumes would also be variable. For our hidden volumes, we assume that we have instances that are from 250 MB to 1.25 GB in increments of 250 MB. In addition to its realism, this allows us to determine a point at which the number of uniform writes becomes conspicuous. For our Artifice parameters we chose those that minimized the number of writes, while achieving survival probabilities over 80% with 25 GB of cover changes. This led us to copy data blocks 6 times, where the survival of a single block is sufficient for reconstructing the data. Since our adversary is able to generate an arbitrary number of disk snapshots with and without hidden volumes, in order to allow our classifier to learn to distinguish disks more quickly our training set contains an even split of disks with and without hidden volumes. However, to reflect the rarity of hidden volumes in the real world only 5% of our test set contains disks with a hidden volume. We generate a training set of size 10000 and a test set of size 2500. We repeat this generation, training and testing cycle 100 times to ensure the reliability of our results.

V. RESULTS

By implementing the experimental methodology described in Section IV and running it against our dataset we collected a set of results that show the efficacy of our proposed multiple snapshot attack. It should be noted from the start that at the core of our implementation is a simple logistic regression that takes only the probability of a single block change on the disk into account. In choosing to implement such a simple learning algorithm we highlight the distinguishing power of analyzing consecutive block changes in detecting anomalous disk behavior.

We collected five different metrics on our classifier: accuracy, precision, recall, false positive rate, and false negative rate. Because only 5% of our test set contains Artifice instances accuracy is not a very informative metric, and we include it for completeness only. Precision is the ratio of true positives to predicted positives. Recall is the ratio of true positives to true positives and false negatives, giving the ratio of Artifice instances that were identified from the test set. False positive rates and false negative rates are useful for understanding how frequently our classifier makes errors in both directions. We consider false classification rates to be the most important metrics for an attacker.

In our experiments the 250 MB Artifice instances were often able to pass undetected, implying that 25 GB of cover changes were sufficient to hide these volumes. However, the largest four sizes were reliably detected, with the three largest sizes, 0.75 GB, 1.0 GB and 1.25 GB, being detected nearly 100% of the time. This highlights a feature of our attack, namely that for 25 GB of cover changes every Artifice instance above a certain size will be detected with high probability. This is because of our use of logistic regression, and because the probability of singletons is so overwhelming. Eventually as free space fills up, Artifice will begin to make changes that are parts of longer chains, but if the reconstruction threshold is low, this will severely impact the survivability of the volume.

The presence of a point where the FNR becomes negligible also offers an explanation for the relatively constant false positive rate. This being that a certain percentage of simulated clean disks will naturally have disproportionately many singletons and thus get misclassified as containing an Artifice volume. Interestingly, there were very few disks that naturally had enough singletons to exceed the learned threshold.

Future work may combine the features we use in this attack with other features. Such as what proportion of changes are made to blocks in free space versus allocated blocks. The introduction of more features would serve to better characterize disk behavior and further improve the efficacy of the attack.

VI. ATTACK MITIGATION

Artifice’s authors propose an operational security based approach to defend against multiple snapshot attacks [5]. The
Hidden volumes in excess of 0.75 GB are always identified successfully.

<table>
<thead>
<tr>
<th>Size (GB)</th>
<th>Acc.</th>
<th>Precision</th>
<th>Recall</th>
<th>FPR</th>
<th>FNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.25</td>
<td>0.981</td>
<td>0.911</td>
<td>0.680</td>
<td>0.004</td>
<td>0.320</td>
</tr>
<tr>
<td>0.5</td>
<td>0.993</td>
<td>0.934</td>
<td>0.937</td>
<td>0.004</td>
<td>0.063</td>
</tr>
<tr>
<td>0.75</td>
<td>0.997</td>
<td>0.942</td>
<td>0.999</td>
<td>0.003</td>
<td>0.001</td>
</tr>
<tr>
<td>1.0</td>
<td>0.996</td>
<td>0.935</td>
<td>1.0</td>
<td>0.004</td>
<td>0.0</td>
</tr>
<tr>
<td>1.25</td>
<td>0.997</td>
<td>0.939</td>
<td>1.0</td>
<td>0.004</td>
<td>0.0</td>
</tr>
</tbody>
</table>

In our proposed attack, we use only one feature associated with single block changes to the disk. A naive approach would be to simply write all blocks in pairs, producing consecutive changes of two blocks. This would defeat our attack, but the adversary could expand its feature set to include changes two blocks long and so on, forcing the user of the hidden volume to mimic the public file system or risk detection. Furthermore, given the myriad measurable features in the average file system and that the adversary is unlikely to publish details of its attack, we can conclude it would be difficult to accurately determine what features are and are not being tracked by the adversary, making mimicry the safest option.

In order to accurately mimic expected access patterns, much more work needs to be done to quantify exactly what a change pattern for a disk without a deniable volume looks like. A significant body of work has been published in the context of network steganography and pattern mimicking cryptography and could be used to inform designers of future multiple snapshot resistance systems. For instance, initial work into format transforming encryption by Dyer et al. [16] showed that it was possible to efficiently encrypt data so that it would conform to a target regular expression. An application of this is found in censorship resistant networking. In this case a user might be running a blacklisted protocol, such as Tor, but transforms the protocol messages in such a way that they look like HTTPS. Similar techniques could be applied to deniable storage to disguise suspicious write patterns. Unfortunately, there is also evidence that more capable adversaries utilizing more sophisticated attacks can easily identify these false protocols. Houmansadr et al. [17] argue that it is unlikely that one could mimic a protocol perfectly without running the actual protocol, because there are often sub-protocols one would also need to mimic or differences in implementations that allow for version fingerprinting. Assuming snapshot analysis becomes more sophisticated it is likely that mimicry techniques applied to deniable storage would also need to evolve.

Since Artifice relies on pseudorandom data in free space and the use of secure deletion utilities to produce cover changes, one way to potentially sidestep the issues of artificial mimicry could be found in actually using these deleted files. Artifice could keep track of changes in free space on the public file system, and when it sees a block added to free space it could overwrite this with an Artifice block. As noted above this is not uniformly written and so may risk corruption of files, but it also may provide stronger mimicry guarantees than other methods. We leave it to future work to investigate this and the other techniques we have presented for mitigation.
VII. Conclusion

We have demonstrated the first implemented multiple snapshot attack against deniable storage systems. In doing so we showed the broad usefulness of change records to an adversary seeking to detect hidden volumes. Furthermore, we presented a concrete way to analyze changing disks, even when the contents and sizes of those disks are radically different. Through our data collection we were able to show that this measure of computing consecutive change lengths of a disk is relatively stable. For our data we collected over fifty images of ext4 disks and used records of changes across these images to train a classifier on probabilities of chains. We demonstrated that this classifier is able to differentiate between disks containing a hidden volume from disks without a hidden volume in a variety of configurations. In the process we have also identified limitations to our technique and from those limitations have proposed possible countermeasures to our attack.

As future work we would like to gather substantially more data from more varied sources. In addition to covering the major file systems, collecting data from many different types of computer users would give us greater confidence in the stability of our metrics. Finally, the attack we propose has great potential for expansion. Many more features could be included, such as number of writes and location of writes in free and used space. Adding additional features would be trivial, and have the potential to further improve the performance of the attack, lowering the false positive rate, and the size of hidden volumes that can be expected to evade detection.

References


Appendix

A. Theoretical Probability of Consecutive Changes

Suppose we have an array, A, of size n and we make k changes to it at random, where 1 denotes a change and 0 denotes no change. What is the probability of selecting a chain of exactly c consecutive changes? We give an example and then give the general statement and proof.

Example A.1. When n = 7 and k = 4, what is the probability of drawing a chain of length 2 from the array?

In this case it is feasible to enumerate all \( \binom{7}{4} = 35 \) possible arrangements of the disk. Doing so we see there are 12 ways to get arrays with one chain of length 2 and two chains of length 1. We also see that there are 6 ways to get arrays with two chains of length 2. Therefore, \( \Pr(C = 2; n = 7, k = 4) = \frac{12}{35} + \frac{12}{35} = \frac{1}{2} \)

For larger values of n and k this quickly becomes infeasible. Instead, we describe a method whose complexity only depends on k.

Lemma A.1. Let A be an array of size n with k \( \leq n \) entries made at random. Then the probability of a chain of c consecutive changes in A is

\[
\Pr(C = c) = \sum_{p \in P} \Pr(C = c \mid p),
\]

where P is the set of partitions of k, \( |p| \) is the number of elements in a partition p, and \( \Pr(C = c \mid p) \) is the probability of c in a partition p.

Proof. Let A be an array of length n with k changes made uniformly at random. Then A can be represented as \( p = (p_1, p_2, ..., p_k) \), an ordered partition of k, where each \( p_i \in \mathbb{Z} \) represents the i-th string of \( p_i \) consecutive 1s separated by one or more 0s. Let \( |p| \) be the length of the partition of k. By our construction of p, A is uniquely represented by p and

\[
p_1 + p_2 + ... + p_{|p|} = k.
\]

Since any array A can be represented by \( p \in P \), where P is the partition of k, we can compute \( \Pr(C = c) \) as

\[
\Pr(C = c) = \sum_{p \in P} \Pr(C = c \mid p) \Pr(p),
\]
by marginalizing over $P$, the size of which is $\binom{2k-1}{k-1}$. We conclude the proof by computing $\Pr(p)$. Since there are $\binom{n}{k}$ possible arrays, counting the number of arrays represented by $p$ is sufficient to compute $\Pr(p)$.

**Example A.2.** (continued from example A.1) The ordered partitions of 4 are

$$(4), (3, 1), (1, 3), (2, 2), (2, 1, 1), (1, 2, 1), (1, 1, 2), (1, 1, 1, 1).$$

*Taking $c = 2$, we have $(2, 2), (2, 1, 1), (1, 2, 1), (1, 1, 2)$ all contain at least one 2.*

Notice that when $p = (2, 1, 1)$, the array will take the form $\star 110 \star 10 \star 1\star$, where $\star$ represents zero or more 0s. Therefore, there is a single 0 whose location is not fixed by $p$, and there are $\binom{4}{1} = 4$ ways to place it.

Arrays that are represented by $p$ must have the form

$$\star 11 \cdots 10 \star \cdots \star 11 \cdots 1 \star$$

by our construction. Notice that there are $k$ 1s and $|p| - 1$ 0s in the string above, so there are $n - k - (|p| - 1)$ 0s whose locations are unfixed. There are $\binom{n-k-1}{|p|}$ different ways to place these 0s, thus

$$\Pr(p) = \frac{\binom{n-k-1}{|p|}}{\binom{n}{k}}, \quad (6)$$

completing the proof.

\[\square\]
Towards Efficient I/O Scheduling for Collaborative Multi-Level Checkpointing

Avinash Maurya*, Bogdan Nicolae†, M. Mustafa Rafique*, Thierry Tonellot‡, Franck Cappello†
*Rochester Institute of Technology, USA
†Argonne National Laboratory, USA
‡Exploration and Petroleum Engineering Advanced Research Center, Saudi Aramco, Dhahran, Saudi Arabia
Email: *{amb429, mrafiq}@cs.rit.edu; †{bnicolae, cappello}@anl.gov; ‡thierrylaurent.tonellot@aramco.com

Abstract—Efficient checkpointing of distributed data structures periodically at key moments during runtime is a recurring fundamental pattern in a large number of use cases: fault tolerance based on checkpoint-restart, in-situ or post-analytics, reproducibility, adjoint computations, etc. In this context, multi-level checkpointing is a popular technique: distributed processes can write their shard of the data independently to fast local storage tiers, then flush asynchronously to a shared, slower tier of higher capacity. However, given the limited capacity of fast tiers (e.g., GPU memory) and the increasing checkpoint frequency, the processes often run out of space and need to fall back to blocking writes to the slow tiers. To mitigate this problem, compression is often applied in order to reduce the checkpoint sizes. Unfortunately, this reduction is not uniform: some processes will have spare capacity left on the fast tiers, while others still run out of space. In this paper, we study the problem of how to leverage this imbalance in order to reduce I/O overheads for multi-level checkpointing. To this end, we solve an optimization problem of how much data to send from each process that runs out of space to the processes that have spare capacity in order to minimize the amount of time spent blocking in I/O. We propose two algorithms: one based on a greedy approach and the other based on modified minimum cost flows. We evaluate our proposal using synthetic and real-life application traces. Our evaluation shows that both algorithms achieve significant improvements in checkpoint performance over traditional multi-level checkpointing.

Index Terms—GPU checkpointing, asynchronous I/O, peer-to-peer collaborative caching, multi-level checkpointing

I. INTRODUCTION

High-Performance Computing (HPC) applications produce massive amounts of distributed intermediate data during runtime that needs to be checkpointed concurrently. This is a fundamental I/O pattern used in a wide range of scenarios: fault tolerance based on checkpoint-restart, offline or in-situ analytics, reproducibility (validate intermediate states in addition to the end result), etc.

Of particular interest is the use of checkpointing for the purpose of revisiting previous states in order to advance a computation. For example, adjoint computations are often used to adjust the parameters of a function used to predict an outcome by minimizing the differences between the actual and expected output. Adjoint computations usually consist of two stages: a forward pass used to obtain the actual output that checkpoints the intermediate states, followed by a backward pass that uses the checkpoints in reverse order to make the adjustments. Such adjoint techniques are ubiquitous, from climate and ocean modeling to seismic imaging in the oil industry. The training of deep learning (DL) models is also an adjoint computation: techniques such as stochastic gradient descent use a forward pass and a backward pass in order to adjust the parameters of a deep neural network (DNN) model composed of multiple layers.

The increasing performance requirements of HPC and DL applications result in the rapid adoption of accelerators such as GPUs. Naturally, this also introduces the need to checkpoint more frequently, especially for scenarios such as adjoint computations, where the checkpoint intervals are in the order of milliseconds. In this context, GPUs are equipped with high bandwidth memory (HBM) capable of keeping up with the massive amount of computational resources without causing data bottlenecks. Unfortunately, HBM is expensive, and thus its capacity is limited. Consequently, applications commonly do not have enough room to keep all checkpoints in HBM for the entire runtime for large problem sizes. Therefore, it is essential to devise checkpointing techniques that combine fast tiers of limited capacity (e.g., HBM of GPUs) with slower tiers of larger capacity (e.g., DDR4 host memory).

In this context, asynchronous multi-level checkpointing techniques are popular. They rely on a simple idea: all processes write their checkpoints to a fast tier, then flush them in the background to a slower tier while the application continues running in the foreground. Given that the slow I/O operations needed to flush the checkpoints from the fast tiers to the slow tiers can often be overlapped with the application runtime, multi-level checkpointing techniques also open up opportunities to mask data movement latency either partially or entirely. However, the constant accumulation of checkpointing data on the fast tiers means there is often not enough free space to write the entire checkpoint on the fast tiers. Therefore, the processes need to directly write to the slow tiers in a blocking fashion, which leads to increased I/O latency.

In a quest to alleviate this issue, various approaches have been explored to reduce the checkpoint sizes, e.g., compression, decimation, and interpolation. These techniques mitigate but do not fully eliminate this challenge. For example, when compression is used, the compression ratio may be high for some checkpoints but low for others. Thus, even if most processes are capable of fitting their checkpoints in the free
space of their fast local tiers, it is enough for some stragglers that were not capable of doing so to delay the whole group especially for tightly coupled applications.

In this paper, we aim to solve the aforementioned challenge. Our approach is based on two key observations. First, if there is a significant difference between the checkpoint sizes, then the processes will fall into two groups: some will have a checkpoint size that does not fit in the free space of their fast local tiers, while others will have free space left even after completely writing their checkpoint to their fast local tier. Second, modern accelerators, particularly GPUs are typically linked with high bandwidth interconnects that are significantly faster than the links between the accelerators and the host memory. This is also true for other classes of fast vs. slow tiers, e.g., the bisection bandwidth of remote transfers between main memories and SSDs is faster than the aggregated bandwidth of a parallel file system. Based on these observations, we propose sending some of the checkpointing data to the processes with spare free space to reduce the amount of data that needs to be written directly to the slow tiers.

While simple as a concept, this is a complex optimization problem, i.e., to obtain a schedule which identifies processes which must transfer a specific amount of data to one or more processes for minimizing execution stalls during checkpoints. We refer to this as the collaborative multi-level checkpointing problem. This is non-trivial both because of link heterogeneity, i.e., each fast tier may be connected to multiple other fast tiers with links of different bandwidths, and because these links can be used concurrently, i.e., it is suboptimal to send all data over the fastest link even if the destination has enough capacity to fit the entire data. To this end, we contribute with two algorithms, which we evaluate in a series of simulated scenarios using both synthetic and real-life traces of checkpoints of variable sizes. We summarize our contributions below:

- We formulate the problem of collaborative multi-level checkpointing, introducing a set of general considerations and assumptions for the design of scheduling algorithms (§ II).
- We propose two scheduling algorithms based on a greedy approach and a variation of min-cost max-flow in transport networks, which leverage the differences between the checkpoint sizes and the peer-to-peer interconnects of the fast tiers to minimize the blocking time of collaborative multi-level checkpointing (§ IV).
- We evaluate our proposed algorithms using the performance model in a variety of experimental scenarios using both a real-world application trace and a synthetically generated trace. Our evaluation shows that our algorithm based on min-cost max-flow approach outperforms the greedy and traditional checkpointing approach while incurring comparable execution time (§ V).

II. PROBLEM FORMULATION

Collaborative multi-level checkpointing is a fundamental I/O pattern applicable in a large variety of scenarios. For simplicity, consider the case of a single node equipped with $N$ GPUs, each of which is assigned to a process $i$. At a given moment, all processes need to simultaneously checkpoint some data structures stored on their GPUs that add up to size $C_{kpt, i}$. Each GPU $G_i$ has a free space $F_i$. We call the processes for which $C_{kpt, i} > F_i$ senders and those for which $C_{kpt, i} < F_i$ receivers. Each GPU $G_i$ is connected to a subset of the other GPUs through a link with a maximum bandwidth $B_{ij}$ (with $B_{ij} = 0$ if $G_i$ is not connected with $G_j$), as well as to the host memory $H$ through a link of bandwidth $B_h$. Each sender can concurrently use any of its peer-to-peer links and the device-to-host link to transfer an arbitrary amount of its remainder ($C_i = C_{kpt, i} - F_i$) to the receivers or to the host memory. The goal is to identify the amount of data that each sender needs to transfer to each receiver or host memory such that the following two conditions are satisfied: (1) the total amount of data sent to each receiver $j$ is smaller or equal to its total spare capacity $S_j = F_j - C_{kpt, j}$; and (2) the total duration of all concurrent data transfers (denoted $t_{ckpt}$ and referred to as blocking time) is minimized. We call this the optimal schedule.

We elaborate this problem by considering Nvidia DGX-1 [1], which has a hybrid cube-mesh GPU interconnect with 8 GPUs that are interconnected with each other using either single (24 GB/s) or dual NVLinks (48 GB/s), as well as with the host memory through PCIe links (12 GB/s). As shown in Figure 1, the senders have a positive amount of data (remainder), the receivers have a negative amount (spare capacity), and the GPUs whose local checkpoints fit exactly in their free space have a spare capacity of zero. In this case, $G_0$ could send all the data to $G_4$ over a fast dual NVLink, but this is suboptimal for two reasons: (1) if $G_0$ writes 1/4 of its data to the host memory in parallel, it would finish 25% faster; (2) $G_6$ is not linked to $G_1$, therefore it is forced to write its data to the host memory while $G_1$'s spare capacity remains unused. Thus, accurately identifying senders and corresponding receivers, and the amount of data that should be sent to each receiver is critical in reducing the overall checkpoint time.

Once the optimal schedule is determined, collaborative multi-level checkpointing can be easily implemented by splitting the checkpoints into chunks on the senders, which then transfer them to the receivers. Next, the chunks can be transferred from their intermediate locations to the host memory in the background while the application continues its execution.

In this paper, we focus on multiple GPUs on a single node. However, it is important to note that the problem can be...
generalized to any number of compute nodes and any other combinations of fast and slow storage tiers, e.g., the main memory of compute nodes and the parallel file system.

III. RELATED WORK

Checkpoint-Restart: Both loosely and tightly coupled applications use checkpoint-restart to support resilience. When I/O bandwidth is a concern (especially for tightly-coupled HPC applications running at a large scale), multi-level checkpointing [2] can be used to leverage complementary strategies (partner replication, erasure coding) adapted for HPC storage hierarchies in asynchronous mode [3]. Currently, such techniques use local storage independently on each compute node via a single shared link, but can be complemented to leverage local storage of remote nodes. Additionally, checkpoint-restart techniques are also used for accommodating on-demand jobs with batch jobs [4], [5] and workload migration [6], [7].

Checkpoint size-reduction: There are a variety of techniques that can be used to reduce the checkpoint sizes for specific objectives: space (i.e., within the same checkpoint), time (i.e., across consecutive checkpoints), scope (i.e., individual checkpoints vs. global checkpoints aggregated from all processes). They include lossy [8] and lossless [9], [10] compression, de-duplication [11], [12], incremental checkpointing based on dirty page tracking [13], etc. Such approaches can be used together with our proposal and often lead to differences in the checkpoint sizes leveraged by our proposal.

GPU checkpointing: With the increasing popularity of GPUs, checkpointing techniques are used both for migration [14], [15] and resilience [15]–[20]. System-level checkpoint-restart libraries, e.g., CheCUDA [16] and NVCR [15] transparently record and replay all the memory-based API calls. Efforts such as MLBS [21] and Check-Freq [22] leverage multi-level memory subsystem starting from GPU memory to minimize checkpoint time. However, none of these approaches analyze the imbalance of checkpoint sizes across multiple devices, nor do they leverage peer-to-peer transfers to reduce the checkpointing overheads.

Schedule optimization: Maximum flows in transportation networks have been studied for decades, and the complexity of the algorithms is constantly improving [23]. Such algorithms can be extended to produce the minimum cost for a given flow [24], which is closely related to our problem if we model it using graph theory. However, these algorithms assume that each transported unit incurs a cost that is the sum of the costs of all links it passes through, while in our case the cost (i.e., blocking time) only increases if we transfer more data over the slowest link. Linear programming [25] is a well-researched area for solving complex optimization problems, with a variety of tools available. However, such tools are typically heavyweight and are not designed to be easily used in real-time for system-level runtimes.

To the best of our knowledge, we are the first to study the problem of optimal schedule for collaborative multi-level checkpointing for different checkpoint sizes and spare capacities, heterogeneous peer-to-peer links, and concurrent transfers.

IV. OPTIMAL SCHEDULE FOR COLLABORATIVE MULTI-LEVEL CHECKPOINTING

In this section, we propose two lightweight algorithms to solve the problem introduced in § II.

A. Greedy-Based Schedule

Our first algorithm is based on a greedy strategy. While it does not guarantee to find the optimal solution, it is easy to implement and has a short runtime due to low computational complexity. The key idea we exploit in this context is to transfer as much data as possible from the sender with the highest checkpoint remainder to the fastest receivers with spare capacity it is connected to. The intuition behind this idea is that the sender with the highest remainder is likely to be forced to transfer most of its remainder to the host memory if other senders occupy the spare capacities of the receivers, thereby it would cause the longest transfer delay in the whole group. Thus, by sorting senders in descending order of their remainder, we minimize the occurrence of this undesired effect.

The complete algorithm is listed in Algorithm 1. The notations used in the algorithm are consistent with those used in § II. Specifically, for each sender i, C_i > 0 is the remainder after it partially wrote its checkpoint to G_i, leaving the spare capacity S_i = 0. Similarly, for each receiver j, C_j = 0 and S_j > 0. The algorithm computes the total blocking time t_{ckpt} and the schedule P: for each sender i, P_i is a set of tuples (k, j), where j is a receiver of the checkpoint chunk of size k. In the worst case, each node is connected with every other node, therefore the complexity of this algorithm is \( O(N^2) \)

<table>
<thead>
<tr>
<th>Algorithm 1: Greedy-based Scheduling Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong>: number of processes ( N ), remainder ( C ), spare capacity ( S ), bandwidth matrix ( B )</td>
</tr>
<tr>
<td><strong>Output</strong>: total blocking time ( t_{ckpt} ), schedule ( P )</td>
</tr>
<tr>
<td>1. <strong>while</strong> ( i \in \arg\max (C) {C &gt; 0} ) <strong>do</strong></td>
</tr>
<tr>
<td>2. <strong>while</strong> ( C_i &gt; 0 ) <strong>do</strong></td>
</tr>
<tr>
<td>3. <strong>if</strong> ( \exists j = \arg\max (B) {B_{ij} &gt; 0 \text{ and } S_j &gt; 0} ) <strong>then</strong></td>
</tr>
<tr>
<td>4. ( k \leftarrow \min (C_i, S_j) )</td>
</tr>
<tr>
<td>5. ( C_i \leftarrow C_i - k )</td>
</tr>
<tr>
<td>6. ( S_j \leftarrow S_j - k )</td>
</tr>
<tr>
<td>7. ( P_i \leftarrow P_i \cup {(k, j)} )</td>
</tr>
<tr>
<td>8. ( t_{ckpt} \leftarrow \max (t_{ckpt}, k / B_{ij}) )</td>
</tr>
<tr>
<td><strong>else</strong></td>
</tr>
<tr>
<td><strong>break</strong></td>
</tr>
<tr>
<td><strong>if</strong> ( C_i &gt; 0 ) <strong>then</strong></td>
</tr>
<tr>
<td>10. <strong>// Transfer to host</strong></td>
</tr>
<tr>
<td>11. ( C_i \leftarrow 0 )</td>
</tr>
<tr>
<td>12. ( P_i \leftarrow P_i \cup {(C_i, H)} )</td>
</tr>
<tr>
<td>13. ( t_{ckpt} \leftarrow \max (t_{ckpt}, k / B_{ih}) )</td>
</tr>
<tr>
<td><strong>return</strong> ( t_{ckpt}, P )</td>
</tr>
</tbody>
</table>
B. Min-Time Flow-based Schedule

The greedy algorithm fails to find the optimal schedule because every transfer decision is irreversible and limits future opportunities for other senders to exploit fast links to their receiver peers. One way to address this limitation is to model the problem as a min-cost max-flow transportation network [24]. In the simplest form, min-cost max-flow aims to find the minimum cost of moving \( F \) items (flow) through a graph of \( N \) vertices (if possible), starting from a source node to a sink node. In this graph, each directed edge represents a maximum number of items that can be transported between the vertices and the cost of each item. In our case, we can create a virtual source and a virtual sink, then link the senders to the source, the receivers with the sink and finally the senders and receivers with themselves. Specifically, the edges from the source to the receivers have capacity \( C_i \) and cost 0, which is needed to limit the amount of data exiting the sender to \( C_i \). Then, the edges from sender \( i \) to receiver \( j \) have capacity \( S_j \) and cost \( 1/B_{ij} \) (meaning sender \( i \) is allowed to transfer up to \( S_j \) and each unit of data takes \( 1/B_{ij} \) time to transfer). Finally, the edges from the receivers to the sink have capacity \( S_j \) and cost 0 (to limit the amount of data exiting receiver \( j \) from all its senders).

Figure 2 depicts the transportation network corresponding to the example shown in Figure 1. A classic algorithm to solve this problem is to iteratively increase the number of transported items (flow) by \( K \), where \( K \) is the minimum capacity among all edges on the minimum cost path from the source to the sink. Note that at each iteration, all edges on the minimum cost path will have their capacity reduced by \( K \). By itself, this is not enough to reverse previous decisions if they proved to be suboptimal. Therefore, the graph is augmented such that for each edge, there is a back-edge whose capacity is increased by \( K \) at each iteration and whose cost is the negative cost of the opposite edge. Using this approach, previous decisions that proved suboptimal can be reversed by allowing back-edges in the minimum cost path. Note that it is not possible to use Dijkstra's algorithm to calculate the minimum cost path, because of the negative cost of the back edges. However, algorithms, such as Bellman-Ford, can be used.

Unfortunately, using the unmodified classic min-cost max-flow algorithm to solve the optimal schedule problem is not possible, because the cost incurred by the concurrent transfers is the duration of the transfer over the slowest link, not the sum of the durations of the transfers over the links along the min-cost path. This has two important consequences: (1) we need to change how the cost of the minimum cost path is calculated by including both latency gain and loss due to back-edges; and (2) it may be suboptimal to increase the flow by the full capacity of the minimum edge along the minimum-cost path as it may increase the overall blocking time. We solve both aspects by extending the classic min-cost max-flow algorithm as listed.

Algorithm 2: Min-Time Flow-based Schedule

\begin{algorithm}
\SetAlgoLined
\KwIn{\( N \) (number of nodes), \( \text{Capacity} \) (capacities of edges), \( \text{Cost} \) (cost of edges)\}
\KwOut{total blocking time \( t_{\text{ckpt}} \), schedule \( P \)}
\While{true}{
    \( d_i \leftarrow \infty \), \( i = 0 \ldots N - 1 \) \tcp*{min distance to \( i \)}
    \( p_i \leftarrow -1 \), \( i = 0 \ldots N - 1 \) \tcp*{parent of \( i \) along path} 
    \( Q \leftarrow Q \cup \{ \text{source} \} \)
    \While{\( Q \neq \emptyset \)}{
        \( u \leftarrow \text{head}(Q) \)
        \( Q \leftarrow Q \setminus \{ u \} \)
        \For{\( v \in \mathcal{B}_u|B_{uv} > 0 \)}{
            \If{\( \text{Cost}_{uv} > 0 \)}{\( f \leftarrow \text{Capacity}_{uv} \)}
            \Else{\( f \leftarrow \text{Capacity}_{uv} \)}
            \( t \leftarrow \max(d_u, \text{abs}(f \cdot \text{Cost}_{uv}) + \text{Cost}_{uv}) \)
            \If{\( \text{Capacity}_{uv} > 0 \) and \( d_v > t \)}{
                \( d_v \leftarrow t \)
                \( p_v \leftarrow u \)
                \( Q \leftarrow Q \cup \{ v \} \)
            }
        }
        \If{\( d_{\text{sink}} = \infty \)}{break}
        \( curr \leftarrow \text{sink} \)
    }
    \While{\( curr \neq \text{source} \)}{
        \( \text{pred} \leftarrow p_{curr} \)
        \( \text{Capacity}_{\text{pred}, curr} \leftarrow \text{Capacity}_{\text{pred}, curr} - 1 \)
        \( \text{Capacity}_{\text{curr}, \text{pred}} \leftarrow \text{Capacity}_{\text{curr}, \text{pred}} + 1 \)
        \( curr \leftarrow \text{pred} \)
    }
    \ForAll{\((u, v) \in \mathcal{E} | u \neq \text{source} \} \text{ and } v \neq \text{sink} \}}{
        \( P_u \leftarrow P_u \cup \{(\text{Capacity}_{uv}, v)\} \)
    }
    \( t_{\text{ckpt}} \leftarrow \max(t_{\text{ckpt}}, \text{Capacity}_{uv} \cdot \text{Cost}_{uv}) \)
\Return{\( t_{\text{ckpt}}, P \)}
\end{algorithm}
in Algorithm 2, which we refer to as min-time max-flow. The algorithm needs the capacities and costs of the edges of the augmented graph, which are initialized as discussed above. To incorporate (2), our algorithm increases the flow by one unit in each iteration, which uses a modified form of Bellman-Ford. Therefore, the worst case complexity is \( O(F \cdot N \cdot |E|) \), where \( F = \sum_{i=0}^{N-1} C_i \) and \( |E| \) is the number of edges in the graph. Note that we used an optimized implementation of Bellman-Ford that prunes the number of attempted relaxations, which can often run in \( O(|E|) \) in practice.

Unlike the greedy algorithm, for the example in Figure 1, the flow based algorithm finds the optimal schedule as follows: \( G_0 \) transfers 137 MB to \( G_1 \), 275 MB to \( G_4 \), and the remaining 68 MB to the host memory. \( G_6 \) transfers 160 MB to \( G_4 \) and 80 MB to the host. Thus, the blocking time is 6.7 ms, which is 3\( \times \) faster than the greedy algorithm.

V. EXPERIMENTAL EVALUATION

A. Methodology

We base our evaluations on two checkpoint traces: one is obtained from a real run of a reverse time migration (RTM) application used in the oil industry, the other is generated synthetically. Each trace consists of a series of global checkpoints, which we refer to as snapshots. In turn, each snapshot records the checkpoint size of each process after applying compression to reduce its size. Additionally, we explore several fixed configurations of free space available on each GPU, which is used to calculate the remainder \( C_i = \text{Ckpt}_i - F_i \) of each sender and the spare capacity \( S_i = F_i - \text{Ckpt}_i \) of each receiver.

We compare our algorithms with a baseline approach that uses a standard strategy adopted by multi-level checkpointing approaches where each process \( i \) writes its checkpoint to \( G_i \). Then, if a remainder \( C_i \) is left, it is written to the host memory. These comparisons are independently performed for each snapshot in order to study the tradeoff between the quality of the checkpoint schedule (blocking time \( t_{\text{ckpt}} \)) returned by each algorithm and the required runtime to obtain it.

We implemented our proposed algorithms and the baseline in a simulation framework written in Python, which is responsible to parse the traces, calculate the remainder \( C_i \) and the spare capacity \( S_i \) used by the algorithms, generate the flow graph corresponding to snapshots in the case of the min-time max-flow algorithm, and finally run the algorithms to obtain the blocking time \( t_{\text{ckpt}} \) and runtimes. The simulations are performed on a Dell PowerEdge C6320 server on the Cloudlab testbed [26], which is equipped with an Intel Xeon CPU E5-2683 v3 CPU and 256 GB of memory, and runs Python 3.8.10 over Ubuntu 20.04.2 distribution.

B. Checkpoint Traces

We start by analyzing the distribution of checkpoint sizes for all snapshots across the runtime of a real-life application. We then use the distribution of checkpoint sizes obtained from the real-life workload to generate synthetic checkpoint traces.

RTM: Adjoint-state methods are commonly used in the oil and gas industry to generate subsurface images from seismic data [27]. Reverse time migration (RTM) [28] is an example of such an application. First, forward and backward propagated seismic wavefields are calculated by solving the three-dimensional direct and adjoint wave equations in a known propagation model. Next, the two wavefields are cross-correlated in time to form the subsurface image. The correlation step requires combining the two wavefields at identical propagation times, and hence one of those wavefields needs to be reversed in time using checkpointing techniques [29], [30]. RTM usually relies on time-domain explicit finite difference (TDFD) solvers of the wave equation in its acoustic or elastic approximation. The wave equation derivatives are commonly approximated with finite-difference stencils of order 2 to 4 in time derivatives and up to 16 in space. The solver generates a snapshot of the wavefield at every step that needs to be stored for time reversal. The total wavefield commonly occupies several terabytes for production-size applications. Compression techniques are thus critical to reducing the overall size and speeding up the transfer from the computation unit to the storage layer. We run the RTM application on Nvidia’s DGX-1 platform consisting of 8 Tesla V100 GPUs, each containing 32 GB of HBM2 memory and interconnected through a hybrid-mesh cube topology. The wavefield reversed in time consists of 776 snapshots with an aggregated compressed checkpoint size across all snapshots of \( \sim 53 \) GB per GPU. We study the variability of checkpoint sizes across different snapshots and show the minimum, maximum, and average checkpoint size across all GPUs for every snapshot in Figure 3. The application starts with smaller average checkpoint sizes, which increase almost linearly up to snapshot 400 and then follow an arbitrary distribution. Figure 4 shows the variability of checkpoint sizes across all GPUs for 5 representative snapshots. Depending upon the compression ratio achieved on each GPU, we observe up to 107 MB (26\( \times \)) difference in the smallest and largest checkpoint size for a given snapshot.

Synthetic: We perform a statistical analysis of checkpoint sizes of the RTM application to develop a model for generating synthetic checkpoint traces. We use the \texttt{rv_histogram} distribution of the \texttt{scipy.stats} Python package to obtain a template distribution from the binned data sample of the RTM application. Note that unlike the RTM trace, in this case each snapshot is allowed to capture the full range of checkpoint sizes observed during the RTM runtime. This is done to study a complementary behavior where the evolution of the checkpoint sizes from one snapshot to another is not smooth, and can change significantly. The distribution of the checkpoint sizes corresponding to this case is depicted in Figure 5.

C. GPU Configurations

We explore two dimensions of GPU configurations:

**GPU Checkpoint Cache Size:** We fix the free space available on the GPUs to 80, 128 and 160 MB, which roughly corresponds to the 50, 75 and 98 percentile of the average checkpoint size of the RTM application. Since the checkpoint sizes of snapshots are increasing (as seen in Figure 3), these configurations are representative of the average checkpoint size.
sizes at various runtime stages, which corresponds to the free space that an application can afford for checkpoint caching.

**Increasing Number of GPUs:** In addition to the DGX-1 configuration with 8 GPUs, shown in Figure 1, we also study the behavior of all three approaches for GPU dense systems by increasing the number of GPUs. Although currently, the maximum number of interconnected GPUs in a single server is only 16 [31], multi-instance GPUs (MIG) and an increasing adoption of GPU dense systems and fast interconnects, e.g., NVLink and NVSwitch, advocate for a scalability evaluation.

**D. Results**

**Checkpoint Schedule:** For every snapshot of the RTM and synthetic traces, we obtain the blocking time $t_{\text{ckpt}}$ returned by each of the three approaches (greedy, min-time max-flow, and baseline) using our Python-based simulator. We assume a granularity of 1 MB for the chunk sizes of the checkpoints, which means the minimum transferable data from a sender to a receiver is 1 MB. To better emphasize the differences between the three approaches, we use the optimal blocking time obtained by min-time, max-flow approach as a reference (ranging from less than 1 ms to 12 ms), and plot the relative percentage increase of the other two approaches in Figure 6 (RTM trace) and Figure 7 (synthetic trace).

Correlating Figure 6a (RTM trace) with Figure 3, for a GPU cache size of 80 MB, we observe a high overhead for the greedy (up to 50%) and the baseline (up to 200%) approaches in the snapshot range 350-450. Outside this range, all the three approaches produce comparable schedules. Specifically, for the snapshots higher than 450, the minimum checkpoint size grows beyond the free capacity, therefore there are only senders and no receivers, which means every process needs to checkpoint their remainder to the host memory. For the snapshots smaller than 350, the opposite is true: the maximum checkpoint size is smaller than 80 MB, therefore there are only receivers and no senders, which means every process can checkpoint directly to their GPUs. With increasing free space available on the GPU checkpoint caches (Figures 6b and 6c), performance differences begin to emerge between the three compared approaches for the higher snapshot numbers. This is expected as the average checkpoint size is increasing with the snapshot number. In these scenarios, greedy and baseline approaches are up to $2 \times$ and $5 \times$ slower, respectively, as compared to the optimal approach. Based on these results, we can draw two conclusions: (1) a sub-optimal schedule can dramatically lower the performance of multi-level checkpointing, especially when the free space on the GPU checkpoint caches are close to the average (compressed) checkpoint size; and (2) it is non-trivial to choose an optimal fixed GPU cache size, because the average checkpoint size varies during runtime.

For the synthetic trace, due to the high variance in the checkpoint sizes for every snapshot (Figure 5), we observe high overheads, i.e., greedy and baseline approaches experience 200% and 800% higher blocking time, respectively, relative to the min-time, max-flow approach. Since the average checkpoint size does not vary significantly during runtime, we observe a consistent speed-up for the min-time, max-flow approach across all snapshots.

**Execution Time:** Next, we focus on the runtime overhead of the three approaches necessary to obtain the checkpoint schedule. Figures 8 and 9 depict the execution time of the three approaches for the real-world and synthetic traces respectively. Due to the limited number of GPUs and therefore graph edges, the min-time, max flow approach runs in nearly same amount of time as the other two approaches despite a higher computational complexity. For the synthetic trace, in Figure 9, we observe that for larger GPU cache sizes, the flow-approach actually runs faster than greedy and the baseline. This is because for increasingly larger GPU cache sizes, the flow-approach becomes smaller, thereby reducing the total flow. Overall, we observe that the execution time is negligible for all three approaches, ranging between 10 to 170 µs.
the min-time, max flow algorithm has no significant runtime penalty compared with the other two approaches and is an all-around winner given that it always produces an equal or better checkpoint schedule than the other two approaches.

**Scalability Study:** We evaluate the scalability of the three approaches for an increasing number of GPUs, ranging from 16 (DGX-2) up to 128 GPUs. To this end, we generate a synthetic snapshot with different checkpoint sizes per GPU, whose distribution is depicted in Figure 10a. We fix the free size of each GPU checkpoint cache at 160 MB. The GPUs are assumed to be connected in an all-to-all pattern using an NVSwitch, similar to the DGX-2 topology.

As the number of GPUs is increasing, we observe in Figure 10b, a significant increase in the blocking time for the greedy and baseline approaches relative to the flow-based approach. This is because the likelihood to obtain non-trivial optimal schedules increases for an increasing number of senders and receivers. Figure 10c illustrates the increase in execution times for all three approaches as a function of the number of GPUs. Again, despite the higher complexity, the flow-based approach outperforms the greedy approach and stays below 1 ms.

Overall, we conclude that with increasing scale, the optimal schedule found by the flow-based approach has an increasingly larger impact, while the execution time remains negligible.

**VI. Conclusions**

In this paper we studied the problem of efficient collaborative multi-level checkpointing, focusing on two algorithms that leverage the spare capacity of the fast local storage of remote peers to minimize the blocking phase needed to cache...
all checkpoints such that they can be flushed asynchronously in the background to shared storage.

Based on simulations with two traces that compare our algorithms with a baseline that does not leverage remote local storage, we make the following observations: (1) the effectiveness of remote transfers depends on the imbalance between the checkpoint sizes (which often occurs due to compression) and the spare capacities (which can vary during runtime); (2) our min-time, max-flow algorithm finds the optimal schedule and significantly reduces the blocking time over the baseline and the greedy algorithm; and (3) the execution time of our algorithms is negligible even at scale.

Encouraged by these promising results, in future work we plan to implement the proposed algorithms in production-ready multi-level checkpointing systems such as VELOC [2]. Another interesting direction to consider is the case when the GPU cache accumulates the checkpoints of successive snapshots. In this case, the free space on each GPU is not fixed and can be dynamically adjusted, e.g. by dropping older checkpoints from the cache, which opens additional optimization opportunities.

ACKNOWLEDGMENTS

This work is supported in part by the ARAMCO Services Company and the U.S. Department of Energy (DOE), Office of Science, Office of Advanced Scientific Computing Research and Argonne National Laboratory, under contract numbers PRJ1008127, Argonne: 0F-60169/DOE: DE-AC02-06CH11357. Results presented in this paper are obtained using the Chameleon [32] and CloudLab [26] testbeds supported by the National Science Foundation.

REFERENCES


Abstract—Network virtualization (NV) becomes an essential technology in cloud computing that isolates network flows for tenants. However, because existing NV technologies like overlay do not enable tenants to directly program (i.e., provision, control, and monitor) network resources, software-defined networking (SDN)-based NV (SDN-NV) has been proposed. Despite its great benefits, SDN-NV has been believed to bring considerable overheads due to the network hypervisor (NH). However, to date, there is no definite performance evaluation that proves the overheads of SDN-NV. To this end, this paper comprehensively investigates the performance and overheads of SDN-NV. Our experiment results reveal that SDN-NV provides the data plane performance comparable to or even better (up to 10.5x better TCP throughput) than the existing NV technologies. Also, the results on NH show that its overheads remain mostly constant, even when the number of switches, virtual networks, or network flows increases. In short, our evaluation indicates that the overhead of SDN-NV should not deter its practical use in datacenters.

Index Terms—Network virtualization, Performance evaluation, Cloud computing, Software-defined networking

I. INTRODUCTION

Network virtualization (NV) is indispensable in cloud computing for providing various network services within the physical infrastructure. NV is responsible for providing isolated network connections—called virtual networks (VNs)—between virtual machines (VMs) and containers. Typically, NV is realized through overlay networking, which attaches additional headers in front of each packet. In overlay networking, however, the tenants are not allowed to control the data plane switches because only data center operators configure overlay networking. To overcome this limitation, NV-based software-defined networking (SDN) has been proposed. SDN is a network architecture that centralizes the control functionalities of switches into an SDN controller (control plane). Then the switches become packet-processing devices and operate on the actions given by the SDN controller. As the SDN controller enables the central management of all switches, the NV of the switches becomes feasible, which introduced SDN-based NV (SDN-NV). SDN-NV places a network hypervisor (NH) above the physical network (PN) of switches (Fig. 1). The NH abstracts the underlying PN and creates multiple VNs in the context of SDN. Thus, each tenant controls its VN through its SDN controller (VN control plane in Fig. 1). This way, SDN-NV opens up new possibilities for tenants that can make VNs programmable [1].

Recent studies added the various functionalities to SDN-NV [2] (e.g., network slicing [3], policy composition [4], address virtualization [5], platform scalability [6], [7], and features for datacenters such as VM migration support [1], [8] and performance management [9]). However, the performance results published so far pose a serious challenge to SDN-NV in that it still needs to prove its practical value beyond research efforts. It is like the early days of (server) virtualization technology, which has been around for a long time but was rarely used in practice due to its overhead. When the overhead proved to be rather small [10], virtualization technology took off and became a fundamental technology for cloud computing. This paper aims to comprehensively evaluate the overhead of SDN-NV to prove whether it is small enough to be deployed in cloud computing infrastructures. Thus, we conduct a critical review of evaluations in previous studies and categorize them based on PN and NH.

First, PN consists of a set of network devices (switches) that forward packets. Previous studies showed the PN performance, but they have critical limitations. First, the key performance of the PN, such as network throughput or packet latency, varies with the number of network switches or flows (which we call variables). Yet, the performance metrics have been measured over only a small range of variables (e.g., one to ten in the number of switches [8]), which is far from realistic scenarios. In addition, evaluations have been conducted with only a single tenant, although the SDN-NV system provides multiple VNs for multi-tenancy. More importantly, the PN evaluations have not been compared with existing NV technologies such as overlay (§II-C1). Such a comparison is essential because SDN-NV is a new generation of NV that enables the control of VNs by tenants. Thus, it is necessary to compare the performance of the PN with that of the existing NV technologies because PN performance determines the user service qualities.

Second, the NH is another source of the SDN-NV overhead because the NH puts additional processing in the control plane. We find that, for the NH, its overheads over the number of flows have not been benchmarked thoroughly. Also, previous studies have evaluated only a single tenant. In addition, most
studies have evaluated NHs with the out-of-date version of OpenFlow (OF) and NV technologies. To summarize, the performance evaluation of previous studies on NHs has missed out key aspects of performance, multi-tenancy, and up-to-date system specifications (§II-C2).

Therefore, it still remains whether the overhead of SDN-NV is small enough to be deployable. To this end, this paper comprehensively investigates the performance and overhead of SDN-NV technology, bringing the following differences and novelties compared with the previous studies:

- Comparison with existing NV technologies, especially two dominant technologies for NV, overlay and NAT [11]. Our experiments are carried out with containers because it is the de-facto in cloud computing due to its high portability and little overhead [12].
- All-inclusive benchmarking on the data plane and NH covering the aspects missed in the previous studies—experiments over the number of switches, flows, and VNs and also multi-tenant environments.

Our benchmarking includes numerous measurements (more than 300 cases), and key results are as follows:

- Surprisingly, in the data plane performance, SDN-NV achieves 10.5× higher throughput (§III-B) and 33% better per-packet latency than overlay. Furthermore, our results show that SDN-NV has superior data plane performance over the existing NV technologies (§III-C).
- The overheads (message processing delay and CPU cycles) of NH mostly stay constant as the number of switches, flows, and VNs increase.
- The NH creates multiple VNs and processes flow rules for them. Therefore, it is expected that when the number of tenants increases, the NH becomes overloaded so that the processing delay on each flow rule can increase. Interestingly, our results show that NH can reduce its processing delay when the number of tenants increases (VNs). We discuss the results and reasons in §IV-B2.

II. BACKGROUND

A. SDN-NV

SDN-NV consists of three layers (Fig. 1): 1) PN composed of physical switches, 2) NH, and 3) VN control plane that runs SDN controllers of tenants. Each tenant can request the creation of its VN topology, which includes the virtual switches and links that connect switches and their hosts. The NH receives the request and allocates the VN resources for the topology. After the VN resources are allocated, the tenant can manage and control the resources.

To illustrate how SDN-NV works, we explain the packet forwarding in SDN-NV. When a new packet arrives at the physical switch, the switch requests the NH for a flow rule to process the packet through a “PACKETIN” message. NH then delivers the PACKETIN to the corresponding SDN controller by determining which tenant the packet is destined for. Upon receiving the PACKETIN, the SDN controller calculates a path that forwards the packets and creates flow rules for the switches in the path. Then, the flow rules are wrapped as FLOWMOD messages and sent to NH, and NH installs the flow rules in the switches. The messages, such as PACKETIN and FLOWMOD, are called control messages. Between the switches and SDN controllers, the NH includes address virtualization mechanisms, which will be explained in the next section.

B. Address Virtualization Mechanisms in SDN-NV

The key difference between NH schemes lies in address virtualization. Address here refers to the network address of hosts, such as IP addresses. SDN-NV allows tenants to select arbitrary addresses for their hosts (virtual address) [2], and NH performs address virtualization to avoid address conflicts between tenants. Existing NHs proposed four kinds of address virtualization schemes: slicing, TID embedding, address mapping, and locator embedding.

First, slicing [3] divides the address space (e.g., of IP) and allocates a subspace to each tenant; thus, each tenant can use the addresses in the subspace, not an arbitrary network address. So, as the number of tenants increases, the subspace that each tenant uses becomes limited. Second, TID embedding embeds a tenant ID (TID) in every packet. Physical switches distinguish the virtual addresses through the tenant IDs. For example, FlowN used VLAN for implementing TID [13], and Libera used an existing field of the TCP/IP header (e.g., source MAC address) to embed the TID [1]. Third, address mapping [5] maintains a mapping between the virtual address and the physical address. So, each virtual address is linked to one physical address. Therefore, tenants can use their own virtual addresses via address mapping. Lastly, locator embedding uses a combination of TID and the switch’s address (called LITE). This scheme supports the migration of a virtual host, which frequently occurs in datacenters for resilience and energy efficiency [8], [14]. With the LITE values, physical switches distinguish packets based on the switches where the packets come from. Note that for slicing, TID embedding, and address mapping, NH immediately processes each flow rule when it arrives at the NH. Conversely, for locator embedding, NH should wait for all the flow rules constituting a path between the source and destination hosts in order to know LITE values.

C. Literature Review

Seven major studies on SDN-NV are summarized in Table I. We categorize the studies into two subcategories: 1) benchmarking on SDN-NV and 2) proposing enhanced functionalities to SDN-NV. We specify the evaluation setting for each study that shows the methodology (simulation or emulation) and southbound interface (OF 1.0 or 1.3). The rest of Table I shows the evaluation metrics for the PN and NH, which are
TABLE I: Previous SDN-NV Studies Comparison in Terms of Performance Evaluation

<table>
<thead>
<tr>
<th>Categorization</th>
<th>Benchmarking</th>
<th>Proposing enhanced functionalities to SDN-NV</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evaluation setting</td>
<td>Emulation (OF 1.3)</td>
<td>Simulation (OF 1.0)</td>
</tr>
<tr>
<td>PN</td>
<td>Flow</td>
<td>Throughput</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Per-packet latency</td>
</tr>
<tr>
<td></td>
<td>Switch</td>
<td>Switch CPU</td>
</tr>
<tr>
<td></td>
<td></td>
<td># of flow rules</td>
</tr>
<tr>
<td>NH</td>
<td>Message processing delay</td>
<td>PACKETIN</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PACKETOUT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PACKETIN–FLOWMOD</td>
</tr>
<tr>
<td></td>
<td>Resource consumption</td>
<td>CPU</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Memory</td>
</tr>
</tbody>
</table>

marked with checks (✓). The symbols in Table I have the following meanings.

- Rectangles: experimental variables—the number of VNs (■), number of physical switches (■), number of flows in PN (■), and no variations (□).
- Triangles: topology virtualization—multiple VNs created and also physical switches are shared by multiple VNs (▲), multiple VNs created but physical switches are not shared (△), and single tenant (●).
- Circles: the level of comparison—comparison to the previous NHs (●), to existing NV technologies (○), and no comparison (●).

1) PN: PN benchmarking has two metrics: flow and switch. Measurements on flow are used for analyzing the performance of SDN-NV on each flow (e.g., TCP or UDP), and measurements on switches are used for analyzing the resource consumption (e.g., CPU and memory) that a switch requires for packet processing.

For the flow metric (throughput and latency), the previous studies have the following missing points. First, the flow throughput can be changed differently on the number of VNs, physical switches, and flows (we call experimental variables or simply variables). However, paper [13] measured the flow throughput without any variable changes (rectangle □ of [13] in Table I). Also, the paper [8] measured flow throughput by changing only the number of switches. This paper conducts experiments on all three variables (i.e., switches, flows, and VNs). Second, per-packet latency is critical in many applications such as data mining [16], but it has not been measured previously. This paper includes the measurements on per-packet latency.

Third, all existing evaluations on PN are conducted with a single tenant (triangle ▲ in Table I), indicating that the performance and overhead of multi-tenancy have not been thoroughly investigated. Thus, we evaluate the performance of PN with multi-tenants. Lastly, existing studies did not compare the PN performance to existing NV technologies. This paper compares SDN-NV with overlay and NAT in the container environment (§III-A). Note that we do not measure the switch metrics (i.e., switch CPU and memory) because such metrics are heavily dependent on switch architectures [17], [18]. Thus, we focus on measurements on flow metrics.

2) NH: For NH metrics, message processing delay, message throughput, and resource consumption are considered. First, message processing delay is the time for processing a control message within NH. It is measured for each message that NH processes, such as PACKETIN, FLOWMOD, and PACKETOUT. As shown in Table I, previous studies lacked experiments in terms of variables (rectangles) [8], [13]. Also, two studies [3], [5] measured the message processing delay of the messages without separation of messages (denoted as “PACKETIN–FLOWMOD” in Table I). This paper measures the processing delay of individual messages (i.e., PACKETIN, FLOWMOD, and PACKETOUT) for all the variables. Also, PACKETIN–FLOWMOD can be derived through the aggregation of messages.

Second, the message throughput is the number of messages processed per second by NH. Only one study [15] measured the message throughput because measuring the message throughput requires the arbitrary generation of messages from both PN and SDN controller, which is only possible in a simulation environment. Because this paper performs experiments in Fig. 2, where software switches (e.g., Open vSwitch), NH, and SDN controllers are utilized with actual network traffic, measuring the throughput is inevitably limited. Note that other studies in Table I did not measure the message throughput for the same reason. Rather, we focus on measuring the message delay, which exhibits the overheads.

Third, for the resource consumption of NH, two studies measured CPU cycles, and one study provided memory usage. In this paper, we measure both CPU cycles and memory consumptions but present the results on CPU cycles due to the page limit. We omit the results for memory consumption, but they tend to be very similar to CPU utilization.

Regarding the OF version, except Libera [1], all previous studies measured the NH overheads with OF 1.0. Considering that most SDN switches in datacenters support OF 1.1 or higher versions, the studies with OF 1.0 are in some sense obsolete. This paper conducts comprehensive evaluations based...
on Libera with the OF 1.3 interface. Lastly, most studies are based on a single tenant (Triangle △ of [1], [8], [13] in Table I). Even in experiments with the multi-tenants [3], [5], each physical switch is dedicated to each tenant (Triangle △), which means that the PN is not actually virtualized. Therefore, this paper carries out experiments where several tenants share physical switches.

III. EMPIRICAL BENCHMARKING ON PN

This section explains the evaluation methodology determined upon the literature review (§II-C1). Then, the evaluation results are presented and analyzed.

A. Evaluation Methodology for PN

Experiment environments. We run experiments with five comparative groups: 1) overlay, 2) NAT, 3) address mapping, 4) TID embedding, and 5) locator embedding. The first two are the popular NV techniques, and the latter three are for SDN-NV. Locator embedding is implemented with VLAN. Fig. 2 shows the experiment environments for the comparative groups. For overlay (Fig. 2a), a bridge is used with VXLAN and veth interfaces. Containers act as servers and clients of connections. Each container is bound to the bridge by a veth interface. Also, each bridge has a vxlans interface to perform packet encapsulation and decapsulation on each packet. In addition, for NAT (Fig. 2b), a bridge performs address translation on each packet. For SDN-NV (Fig. 8c), we use Open vSwitch (OVS) that supports the OF protocol. These settings are the representative and widely used ones for NV in containers [11]. The software switch (e.g., bridge or OVS) that processes the packets from the containers directly is located in Machine 1 and Machine 3. In Machine 1 and Machine 3, multiple containers are created to run TCP and UDP connections. Between Machine 1 and Machine 3, we create a number of software switches (OVS) in Machine 2.

Metrics. We measure throughput and latency on flow. Throughput determines the maximum amount of data that each server and client pair can transmit per second. We measure throughput using iperf3 with TCP flows. Each TCP segment size is set as the maximum transmission unit\(^2\). A TCP connection lasts 180 s, and the average throughput per second is shown. Latency is the round-trip time (RTT) between a server and client pair that shows the per-packet processing delay. We measure the per-packet latencies using sockperf with UDP packets as the pure overheads of NV technologies. For each delay experiment, the RTT measurement is carried out for 2 minutes in succession and repeated ten times.

Experiment variables. Table II lists the experimental variables for PN. For throughput, we vary the number of switches, flows, and VNs. The range of each variable is selected as the maximum possible range in the experimental machine. Table II means that when the number of switches changes from 3 to 100, a single TCP flow is generated with a single VN. In the case of variable VN, we create 20 flows with 3 switches, which produce 20 TCP flows to transmit the maximum number of packets. We evenly distribute 20 flows for all VNs. For the variable flow (i.e., 1–100), the number of switches is fixed at 3, and the number of VN is at 1.

However, for latency, we change only the number of switches. Latency measurements aim to compare the processing latency on each packet without network congestions. Because the latency is affected by background traffic (e.g., the addition of queuing delays), we do not evaluate latency in the varied number of flows and VNs.

B. PN Evaluation Results: Throughput

Fig. 3 plots the TCP throughput with experimental variables. Each dot is the aggregated throughput of the TCP connections. First, as the number of VNs increases (x-axis), Fig. 3a shows that the throughputs of four comparative groups maintain relatively consistent values: approximately 8 Gbits/s on average.

\(^2\)We deploy different underlying software switches to enable NV technologies (e.g., bridge and OVS) as the NV technologies are deployed with them [11]. Also, we know that packet size affects performance because of the per-packet processing overheads whose variations depend on the switches. Such variations reduce by setting the packet size as the maximum transmission unit.
However, overlay shows quite a poor throughput—1.38 Gbits/s on average.

In fact, the poor performance of overlay is investigated in previous studies [11], [19], [20] and is proven due to the heavy packet transformation using kernel features with bridges. To encapsulate and decapsulate packets for overlay (e.g., vxlan), the packet should traverse the networking stack of the kernel two times, which explains the poor throughput. On the other hand, NAT, address mapping, and TID embedding process packets only within the software switch, so their throughput is 5.8× higher than that of the overlay.

Note that locator embedding in SDN-NV also performs packet encapsulation and decapsulation. Specifically, locator embedding performs the packet encapsulation and decapsulation twice to include LITE values (§II-B) for the source and destination hosts on each packet. Locator embedding lowers its overhead by using SDN-enabled software switches (e.g., OVS)—OVS can offload the overheads of packet encapsulation and decapsulation on NIC (vxlan offloading); thus, the throughput of locator embedding approaches results similar to those of NAT, address mapping, and TID embedding.

Second, when the number of switches increases (Fig. 3b), the throughput decreases for all comparative groups. The throughput of overlay, NAT, address mapping, TID embedding, and locator embedding decreases by up to 90%, 58%, 59%, 59%, and 63%, respectively (comparing three switches and 100 switches). Also, between the five groups, overlay also shows the lowest throughput due to its encapsulation and decapsulation overhead—on average, the overlay throughput values are 88% lower than those of the other four groups.

Third, when the number of flows changes (Fig. 3c), all comparative groups show relatively constant throughput. Specifically, NAT, address mapping, TID embedding, and locator embedding show similar throughputs, 8–9 Gbits/s, while the overlay shows the lowest throughput—1.38 Gbits/s on average. The throughput of overlay is 83% lower than the average throughput of the other four comparative groups.

C. PN Evaluation Results: Per-packet Latency

We have measured the entire distribution of per-packet latencies, but due to the space limit, only the median latency (50%) is presented in Fig. 4. For all comparative groups, the latency broadly increases as the number of switches arises. By comparing the comparative groups, Fig. 4 shows that address mapping, TID embedding, and locator embedding have lower latencies than both NAT and overlay (e.g., 0.65×, 0.58×, and 0.73×, respectively, of averaged latencies of NAT and overlay) when the number of switches is small (1–50). This is because SDN-NV schemes are deployed with OVS, which has a better utilization than the kernel-based approach (i.e., bridge). Considering that typical connections in datacenters go through quite less than 50 switches [21], the results indicate that SDN-NV schemes have better per-packet latency than NAT and overlay in cloud computing.

On the other hand, as the number of switches increases, the latencies of address mapping, TID embedding, and locator embedding increase rapidly. Thus, all comparative groups show similar latencies (approximately 400–500 µs at 100 switches). The comparative groups have different implementations of NV, so that they should have produced a range of respective latencies. However, the latencies are amortized with the packet processing delays within the switches as the number of switches increases. Thus, the latencies of the comparative groups become similar when the number of switches is high.

IV. EMPIRICAL BENCHMARKING ON NH

A. Evaluation Methodology for NH

Experiment environment. Different from the PN experiments (Fig. 2c) that create PN on three separated physical machines, PNs in NH experiments are created on a single machine. This is because NH experiments aim to see the overheads of NHs, not the PN, so we reduce the complexity of PN configuration. Mininet of OVS emulates PN topologies—linear and fat-tree topologies, as shown in Table III. In the linear topology, the number of switches in a forwarding path between a source and destination hosts is as many hops as switches in the network. For the 4-ary fat-tree topology, a path consists of five switches on average.

Flows are generated as TCP connections using iperf3. The created flow does not stop until the last flow is created so that NH receives control messages for flows consecutively. With the emulated PN on a physical machine, two additional machines run NH and SDN controllers of tenants, respectively. Each machine is equipped with two Intel Xeon E5-2690 CPUs (24 cores) and 64 GB of memory. A 10 GbE Ethernet connects three machines. For SDN controllers, we use ONOS. The match fields of flow rules from ONOS include ethernet, IP, and port addresses to process the new flows in the PN with a separate flow rule so that the number of control message processing tasks of NH is the same as the number of flows3.

Metrics. We measure two metrics: message processing delay and resource consumption of NH. The message processing delay is presented for each control message type (e.g., PACKETIN, FLOWMOD, and PACKETOUT). For resource consumption, we measure the CPU cycle. CPU cycles are measured while control messages are processed.

TABLE III: Variable Changes for NH.

<table>
<thead>
<tr>
<th>Changing variable</th>
<th>Switch</th>
<th>VN</th>
<th>Flow</th>
<th>PN topology</th>
<th>Hosts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch</td>
<td>1 to 100</td>
<td>1</td>
<td>1</td>
<td>Linear</td>
<td>2 for a VN</td>
</tr>
<tr>
<td>Flow</td>
<td>20</td>
<td>1</td>
<td>1 to 200</td>
<td>4-ary fat-tree</td>
<td>16 for a VN</td>
</tr>
<tr>
<td>VN</td>
<td>1</td>
<td>10 to 70</td>
<td>70</td>
<td>Linear</td>
<td>2 for a VN</td>
</tr>
</tbody>
</table>

3If the rules from ONOS match up to IP addresses, multiple flows from the same host pairs can be processed with the single flow rule.
Experimental variables. We use three variables in experiments: the number of physical switches, flows, and VNs, as shown in Table III. The range of each variable (e.g., 10–100 for switches, 10–200 for flows, and 10–70 for VNs) is determined under the condition where the CPU, NIC, and memory of the experiment machines do not become bottlenecks. Their ranges are wide and sufficient compared to those of previous studies. As shown in Table III, when each variable is changed, the other variables are fixed at a value. We set the inter-flow generation time to 1 s. To change the number of VNs, we evenly distribute the flows for all VNs. For example, when the number of VNs is seven, each VN processes ten flows. The VN topology is cloned into a PN topology.

Comparative groups. NHs differ from each other based on the address virtualization schemes (§II-B); thus, we compare the above metrics and variables for three SDN-NV comparative groups: 1) address mapping, 2) TID embedding, and 3) locator embedding. These three comparative groups represent the overheads of NH. In SDN-NV, NH is a standalone component between PN and VN control plane. Thus, the metrics, such as message processing delays and CPU cycles, are the overheads additionally added to the tenant’s control plane due to the SDN-NV. The comparative groups are run by Libera because it has the implementation of three comparative groups. Note that for the experiments on NH, we do not measure overheads of the existing NV technologies because the NH is the component that only exists in the SDN-NV. Therefore, the results presented in this section are overheads of SDN-NV. There are several applications of SDN without NH (non-virtualized SDN) on existing NV technology to enhance network management [22]. The performance or overheads of the application can be found in previous studies that measured performance in non-virtualized SDN [23], [24], [25], [26].

B. NH Evaluation Results: Message Processing Delay

Figs. 5, 6, and 7 show the message processing delays of PACKETIN, FLOWMOD, and PACKETOUT messages. For each message, we analyze the delay by varying the experimental variables (x-axis), so the subfigure (a) is the result of changing the number of VNs, (b) switches, and (c) flows.

1) PACKETIN messages: In Fig. 5a, when the number of VNs (x-axis) increases, the message processing delays have little variance among the three comparative groups: 1.84 ms on average. Also, when the number of switches (Fig. 5b) increases, the delays are similar between comparative groups: 4.65 ms on average. When the number of flows increases (Fig. 5c), address mapping and TID embedding show approximately constant delays (1.02 ms on average), but locator embedding shows much longer delays (2× higher than both address mapping and TID embedding).

We analyze further details when the number of flows increases. The delay for PACKETIN depends on the number of host addresses because for each PACKETIN message, NH looks up the objects that represent the hosts: virtual addresses (address mapping), corresponding TID (TID embedding), or LITE values (locator embedding). Such lookup delay increases as the number of objects increases. However, according to Table III, the number of the total hosts is identical, which means that the delay is expected to be constant. That is consistent with what Fig. 5a and Fig. 5b show. However, for locator embedding, the delay increases up to 39% (when comparing 10 and 200 flows).

Regarding the locator embedding, we find that the semantics in matching fields of SDN controller and NH make the difference. The PACKETIN message triggers the flow rule installations for packet forwarding. The locator embedding in NH distinguishes the flow rules from the SDN controller by generating flowID, which is a hashed key for the addresses in the flow rules. However, the locator embedding creates flowID keys without transport layer addresses and is only based on the data link and IP layer addresses. Our SDN controller creates flow rules to match the port addresses of the transport layer. This leads to mismatching flow rule processing between the SDN controller and NH. In other words, flow rules are created from the SDN controller, but they cannot be interpreted correctly in NH. Thus, they are not installed correctly at the physical switches. Accordingly, the PACKETIN messages are generated repeatedly from the physical switches, which results in the high delay in Fig. 5c. We present the optimization results on this problem and show that the processing delay can be significantly reduced to a level similar to those of address mapping and locator embedding (§IV-D).

2) FLOWMOD messages: The FLOWMOD message (Fig. 6) is used for installing or modifying new flow rules. In Fig. 6a, when the number of VNs increases, the processing delay...
decreases by up to 39%, 41%, and 23% for 10 and 70 VNs, respectively, for address mapping, TID embedding, and locator embedding. This decrease is somewhat surprising because the bottleneck for NH and message processing delays could increase as the number of VNs. This experiment sets the number of flows to 70 (Table III). Thus, as the number of VNs increases, the number of flows that each VN processes becomes smaller because 70 flows are distributed equally to each VN. Accordingly, the flow rules for each virtual switch are reduced, which contributes to reducing the delay of NH. In Fig. 6b, when the number of switches increases, address mapping and TID embedding show relatively constant delay—5.2 ms and 3.7 ms on average, respectively. The delays of the two comparative groups do not increase because NH we use (Libera) parallelizes the flow rule processing. NH handles each flow rule with a separate thread. Therefore, as long as CPU is not the bottleneck, flow rule processing is not affected. However, in the case of locator embedding, the delay increases linearly—up to 3.57× when the number of switches grows 10 to 100. This is because locator embedding processes flow rules when a set of rules composing a path is all arrived at NH; thus, the delay increases as the number of switches composing a path increases.

In Fig 6c, as the number of flows increases, the processing delays of all comparative groups increase—from 10 to 200 flows, 2.19x, 2.47x, and 4.69x times, respectively, for experimental variables. The experiments use a single VN with 20 switches (Table III). The reason for the increase is the increased flow rules in the NH with the number of flows, which results in the delay increase.

3) PACKETOUT messages: The PACKETOUT message is used to inject packets into the network in order to deliver data packets to the destination host. During the flow rule installation, the packet forwarding in data plane can pause when network switches do not have flow rules; so, through PACKETOUT, NH tries to reduce such pause in packet forwarding. In Fig. 7, for all varying experimental variables (i.e., VNs, switches, and flows), the delays are quite constant for address mapping and TID embedding. For locator embedding, the delays are constant for the increasing number of VNs and switches. However, when the number of flows increases, locator embedding shows the delay increased up to 1.5 ms (50% higher than that of the 10 flows). The result comes from the semantic difference problem of locator embedding described above.

C. NH Evaluation Results: CPU Cycle

Fig. 8 shows the CPU cycles consumed in NH when the three messages are being processed. The CPU cycles are measured along with the experiments for Figs. 5, 6, and 7. When the number of VNs changes (Fig. 8a), the cycles of address mapping and TID embedding are similar: 17.4% and 16.26% on average, respectively. Locator embedding shows 34.4% average CPU cycles, much higher than the other two. It is because locator embedding has to track the source and destination hosts and their attached switches to generate LITE values (§II-B).

In Fig. 8b, when the number of switches increases from 10 to 60, all three comparative groups show increasing CPU cycles (2.15x, 2.43x, and 1.98x higher for address mapping, TID embedding, and locator embedding, respectively). However, when the number of switches increases from 80 to 100, the comparative groups show relatively constant CPU cycles—1.81 cores, 1.48 cores, and 1.37 cores, on average, for address mapping, TID embedding, and locator embedding.

The main reason for the constant CPU cycles is the topology discovery where NH regularly updates the topology (e.g., link connections between switches and hosts) through echoing switches. For echoing, NH and switches exchange a small size of TCP packets (e.g., 190 bytes on average) in a very short time (mostly within 50–110 ms). In cloud computing, this kind of network communication is known as “short-lived TCP,” and it is well-known that networking stacks of operating systems become bottlenecked with the high number of short-lived TCP packets [27]. In our experiments, 60 switches used in the experiments cause the bottlenecks on the networking stack, so the CPU cycles of NH is dominated by the underlining kernel, not NH itself. Thus, the CPU cycles of NH become constant values from the 60 switches.

In Fig. 8c, the CPU cycles increase with the number of flows. The CPU cycles of 10 and 200 flows increase by 0.8x, 0.5x, and 10x, respectively, for address mapping, TID embedding, and locator embedding. Note that the high increase of locator embedding is due to the flow rule semantic difference.

D. Optimization on Locator Embedding

Herein, we introduce optimization on locator embedding that solves its semantics problem (§IV-B1). We modify the flowID implementation of Libera’s locator embedding so that
the new implementation includes port addresses (transport layer) in addition to MAC and IP addresses. The modifications include additional improvements, such as the elimination of redundant table lookup and unnecessary loggings. Except for the flowID creation, other operations in Libera (such as managing ARP and physical flow row creation) are based on IP addresses without considering port addresses. We do not change any operation of Libera, but only modify the code for handling flowID.

Fig. 9 shows the enhanced results on locator embedding. Fig. 9a, Fig. 9b, and Fig. 9c show PACKETIN processing delay, PACKETOUT processing delay, and CPU cycles. The line with circles represents the evaluation results with the old locator embedding, and the line with * marks represents the ones with our newly modified locator embedding. In Fig. 9a and Fig. 9b, PACKETIN and PACKETOUT delays of the new locator embedding show relatively constant values (1.47 ms and 0.82 ms, respectively, on average) while PACKETIN and PACKETOUT delays of the old locator embedding continuously increases. Specifically, the new locator embedding improves the processing delay of PACKETIN and PACKETOUT by about 32% and 31% on average, respectively which becomes similar to that of address mapping and TID embedding. In Fig. 9c, the CPU cycle of the new locator embedding shows constant values (36.5% on average), which is 7.3x lower than existing locator embedding on average and similar to address mapping TID embedding.

V. CONCLUSION

The goal of this paper is to investigate the performance and overhead of SDN-NV comprehensively. We run a large number of benchmarks to evaluate SDN-NV with varying numbers of switches, flows, and VNs. Then, we compare the results with existing NV technologies. Contrary to the popular belief that SDN-NV has little practical value due to its overheads, this paper reports that in the data plane performance, SDN-NV is competitive over the existing NV technologies like overlay, and sometimes it is even superior to them. In addition, even when the number of VNs, switches, or flows increases, our results show that the overheads of NHs mostly remain constant. We hope our benchmarking results make a case for SDN-NV to be adopted in cloud datacenters.

REFERENCES

REBAL: Channel Balancing for Payment Channel Networks

Nitin Awathare†, Suraj†, Akash ∗, Vinay Joseph Ribeiro† and Umesh Bellur†

†Department of Computer Science and Engineering and ∗Electrical Engineering, Indian Institute of Technology Bombay

{nitina, surajyadav}@cse.iitb.ac.in, {akash.kumar,vinayr, umesh.bellur}@iitb.ac.in

Abstract—Cryptocurrency networks are a promising infrastructure for pseudonymous online payments. However, low throughput has prevented their widespread acceptance. A promising solution to scale throughput is the Payment channel network (PCN), exemplified by the Lightning Network (LN), that uses a network of off-chain bidirectional payment channels between parties that wish to transact often. Since payments use the shortest paths with sufficient funds over this network, channel balances get exhausted in the direction transactions flow and eventually become unidirectional. This results in transactions failing and consequently a lower transaction success ratio. Our observations on the production LN show that over 63% of the channels lose over 80% of the channel balance in one direction over time, which makes the success ratio of a real-world workload drop from 71% to 29%. A unidirectional channel along a path results in a failure message back to the source that recomputes the path, excluding the failed channel and reattempts the transaction, thus adding to the completion latency even for those transactions that do complete.

We propose REBAL, a distributed re-balancing mechanism, and a new routing scheme to address the above issues. REBAL maximizes the extent to which channels can be re-balanced across the entire network. REBAL addresses the completion latency issue by re-routing transactions from intermediate nodes around a unidirectional channel rather than propagating the failure back to the source.

Our comprehensive evaluation of REBAL shows that the success ratio improves from 30.18% to 79.54% and success volume from 3.98% to 29.99% for a real-world workload derived from the Ripple network, without adversely impacting the transaction latency. Even at very high transaction rates, REBAL outperforms Lightning Network Daemon (LND - a Golang implementation of LN) (12%) with a success ratio of 43.76%.

I. INTRODUCTION

Blockchains are decentralized platforms that support cryptocurrency applications. Bitcoin is the first such cryptocurrency built on a permissionless blockchain. Despite being popular, its practicality is questionable because of two main factors: 1) high transaction fees and; 2) long block confirmation time and consequently low throughput (≈7 transactions per second).

One solution to these issues is a Payment Channel Network (PCN) [3], [15], [21] that consists of a set of off-chain bidirectional payment channels between interested parties. Once a channel is created between users on the PCN, they can transact off-chain as long as there are sufficient funds along the path connecting them. Since establishing a channel requires on-chain transactions that are expensive, it is recommended not to create a direct payment channel for every pair of users that need to transact. By routing payments through intermediate payment channels, participants in the PCN can transfer funds even if they do not share a direct channel.

Lightning Network (LN) [21], the best known PCN, has over 22K nodes [4]. However, PCNs, including the LN, have their challenges. A major issue is that as transactions flow through the network, channel balances of those channels in the transaction path get progressively exhausted. This makes such channels "unidirectional or imbalance" over time, resulting in failed transactions on paths containing these channels and consequently results in a lower success ratio (ratio of successful to total transactions issued).

Existing work to improve the transaction success ratio can be categorized as follows: 1) replenishing capacity by going to the main chain [20], 2) re-balancing the channels in-place [13], [16] and 3) load balancing transactions across paths from the source so that it takes longer to exhaust the capacity of all paths between a source-destination pair, but it doesn’t entirely avoid it [14], [18]–[20]. Replenishing channel capacity is expensive since it requires going to the main chain. Dynamic re-balancing in place can be centrally controlled or distributed. The former suffers from the issue of halting the PCN’s transactions while re-balancing is in progress. It also requires the nodes to disclose its individual contribution in the channel to the central controller, thus violating privacy. The distributed re-balancing strategy in [16] attempts to split the capacity of a node equally across all channels it is connected to and suffers from not being cognizant of the node’s workload, which is shown to be non-circular and deals with transactions primarily in one direction [9].

Based on our analysis of real world workloads and our simulations of these transactions on the real LN topology, we propose REBAL - a distributed in-place re-balancing approach. It re-balances the maximum number of channels in the network to the greatest possible extent by preferentially selecting the longest cycle each node is involved in to start the re-balancing. Unlike previous work [13], REBAL preserves privacy and does not halt the PCN when the re-balancing process is active. Finally, REBAL takes cognizance of the workload history while making decisions about the re-balancing amounts.

To avoid rejection of transactions when a channel balance is temporarily locked during re-balancing, REBAL allows an intermediate node to re-route the transaction through an alternative path to the next node.
We have implemented REBAL and evaluated its effectiveness using a series of real-world LN topology snapshots in addition to a few synthetic topologies \cite{1}, \cite{8} against the Ripple \cite{5} transaction workload. REBAL outperforms all the state-of-the-art algorithms, even when the transaction generation rate is high. For the LN topology, REBAL improves the success ratio (ratio of successful to total transactions issued) from 30.18% to 79.54% and success volume (ratio of successful to total transactions amount processed) from 3.98% to 29.99%, without significantly affecting the transaction latency.

In summary, we make the following contributions in this paper:

- A distributed in-place re-balancing strategy for PCNs, REBAL, that reduces channel imbalances in the network and hence improves the transaction success ratio. Coupled to this, we also introduce a dynamic re-routing scheme from intermediate nodes to reduce transaction latency of successful transactions.
- A theoretical proof that REBAL indeed minimizes the imbalance in the network.
- A concrete, real implementation and comprehensive evaluation of REBAL, with real transaction data on a real topology and empirical evidence that REBAL outperforms the state-of-the-art routing schemes.

The rest of this paper is organized as follows: §II describes the background required to understand the work. §III describes the re-balancing strategy, followed by re-routing at an intermediate node. §IV describes a prototype implementation of REBAL on a simulated environment, experimental setup and the observations from the obtained results. Further we explained the related work in §V and end the paper with discussion and future direction in §VI.

II. BACKGROUND & MOTIVATION

In this section we lay out the background material to understand our work.

**Payment Channel Networks** A Payment channel network (PCN) is a set of bidirectional payment channels between users/nodes that wish to transact often with each other. Each channel is created using a funding transaction on the blockchain that locks the funds from both parties. Now, the parties can transact with each other off-chain instantly and without fees and keep a log of the latest balance. Suppose nodes $X$ and $Y$ initially contribute an amount of $a$ and $b$ to the channel $X-Y$ respectively. A transaction of amount $z$ from $X$ to $Y$ (with the constraint that $z$ does not exceed $X$’s current balance) updates the balances of $X$ and $Y$ to $a-z$ and $b+z$ respectively. Similarly $Y$ can pay $X$ any amount not exceeding its current balance, and the balances of the two parties are updated accordingly.

Since payments can take place in both directions, such channels are called “bidirectional”. In case the channel balance of one part (e.g. $X$) becomes zero, then that party ($X$) can no longer use the channel to pay the other party ($Y$), in which case we say the channel has become "unidirectional". A payment from $Y$ to $X$ would then make $X$’s balance non-zero, and the channel bidirectional again. When the channel is no longer required, an on-chain transaction signed by both parties refunds to them their latest channel balances and terminates the channel.

A PCN allows a sender and receiver to transact without sharing a direct channel. For a transaction, if the sender and receiver are connected via a direct channel, they can easily transact as described above. But in case they are not directly connected, the sender finds a (multi-hop) route to the receiver and sends the funds along the route \cite{17}, thereby using other payment channels for routing by providing a fee to the intermediate nodes.

The blockchain is involved only for creating and closing the channels and for dispute resolution. Limiting the use of blockchain for these purposes reduces the transaction confirmation latency from approximately 60 minutes to a few seconds in LN. Furthermore, sending the transaction through the private channels requires routing fees which is $\approx$500 times lower than the fees for on-chain transactions.

A party can send a payment to another provided there exists a multi-hop path from the source to destination and the payment amount is less than the minimum channel balance of all channels along the path (in the direction of the payment). Each node has complete visibility of the topology of the network, but is not aware of the latest balances of channels, except for channels with its own neighbours. Therefore, a transaction may not succeed. This is because an intermediate channel on the path may not have sufficient balance to forward the incoming transaction. In which case, it sends an error message back to the source. The source retries the transaction along alternate paths till it times-out, runs out of paths, or the transaction succeeds.

**Motivation** We have simulated the performance of the LN topology using a real-world Ripple workload \cite{6} to understand the reasons for transaction failures. Our experiments, where all the channels are balanced initially (at time 0), reveal that with time (approximately 200 seconds), for over 63% of the channels, 80% of the channel balance is along one side of a bidirectional channel. Due to channel balance getting exhausted in one direction, the success ratio for this workload drops from 71% to 29% in about 600 seconds, as depicted in figure 1.

An analysis of the state of the LN topology network with the Ripple workload reveals that when the network reaches a state of imbalance indicated above, it is possible to re-balance the network such that the imbalance of around 60% of channels...
can be cut in half without the need for an on-chain transaction. So, if we can periodically re-balance the network, it will help the PCN to complete a large number of transactions without putting any load to the main chain.

III. REBAL: DISTRIBUTED IN-PLACE RE-BALANCING

This section describes our re-balancing strategy. We start with an algorithmic approach, analyze its feasibility and then suggest a heuristic.

A. Problem formulation and optimistic algorithm

Recall that in the LN, each node has visibility of the entire LN topology but has access to the latest balances of only the channels to which it is connected. Each node can thus calculate how (im)balanced the channels to which it is connected are. It does this as follows. Let $\Delta$ be the maximum time a node takes to perform re-balancing. If $i$ and $j$ are the two parties involved in the channel $C_{ij}$, let $\Theta_i$ and $\Theta_j$ be the average incoming transaction amount for re-balancing time ($\Delta$) for channel $C_{ij}$ at node $i$ and $j$ respectively. Note that $\Theta$ is calculated from past transaction history. Let $\theta$ be the average rate of arrival of transactions amount. Then $\Theta = \theta * \Delta$. Each REBAL node maintains a queue of transactions waiting to be processed. Let $\lambda_i$ and $\lambda_j$ be the sum of transaction amounts waiting in the queue for $C_{ij}$ at $i$ and $j$ respectively. Also, $bal_{ij}$ and $bal_{ji}$ are the individual contributions of $i$ and $j$ in the channel $C_{ij}$, and we define channel capacity $Cap(C_{ij}) = bal_{ij} + bal_{ji}$. To consider incoming and outgoing $\Theta$ and $\lambda$ while making the re-balancing decision, we define the projected imbalance factor of the channel for time $\Delta$ as follows:

$$ImBal_{C_{ij}} = \left[\frac{(\Theta_i + \lambda_i + bal_{ij}) - (\Theta_j + \lambda_j + bal_{ji})}{Cap(C_{ij})}\right]$$  \hspace{1cm} (1)

Each node $i$ aims to reduce $ImBal_{C_{ij}}$ to be below a certain threshold, $Thr$. Ideally we would like $ImBal_{C_{ij}} = 0$. Node $i$ does this by transferring amount $idealBal_{ij}$ (calculated using eq 2) to $j$ if $(\Theta_i + \lambda_i + bal_{ij}) > (\Theta_j + \lambda_j + bal_{ji})$ and $bal_{ij} \geq idealBal_{ij}$.

$$idealBal_{C_{ij}} = \frac{Cap(C_{ij}) * (ImBal_{C_{ij}} - Thr)}{2}$$  \hspace{1cm} (2)

Given this, we now define an Imbalance graph ($G_{ImBal}$) that has all the nodes of the original graph representing the PCN. Each directed edge of $G_{ImBal}$ represents the amount one node should send another to reduce $ImBal_{C_{ij}}$ to $Thr$. For example, $G_{ImBal}$ of the LN topology in figure 2 for $Thr = 0$ is given in figure 3 (For simplicity of representation we have omitted the $\Theta$ and $\lambda$ from it). Note: no single node knows the entire Imbalance graph. They only have a local view of it, i.e., the channels the node is connected to. A pair of nodes $<i,j>$ do not have an edge in the imbalance graph if the $ImBal_{C_{ij}} \leq Thr$. Throughout the re-balancing process, we do not allow a node to reverse the edge direction because we are considering pending as well as future transactions while calculating $ImBal_{C_{ij}}$. From now on, unless explicitly stated, all the further discussion in this section is about the Imbalance graphs.

The chance that a transaction succeeds is directly related to the extent the network is balanced. So our aim is to reduce the imbalance factor over as many edges in $G_{ImBal}$ to the maximum possible extent, i.e., minimize $I = \sum_{i,j\in V(G_{ImBal})}ImBal_{C_{ij}}$. This is similar to finding the maximum circulation in a network [23]. The strategy is to re-balance channels by a node sending some amount to the other across the channel that is imbalanced. If $f_{ij}$ is the (re-balancing) amount that $i$ sends through channel $C_{ij}$ to $j$, then, transferring the maximum value of $f_{ij}$ ($0 \leq f_{ij} \leq idealBal_{C_{ij}}$), from $i$ to $j$ can reduce the $ImBal_{C_{ij}}$ to the maximum possible extent. At the same time $i$ should receive the same amount from its neighbours who have incoming edges to it, in imbalance graph, so that it does not touch its total amount. This can be done by transferring the amount along a cycle. If $N^-_i$, $N^+_i$ are incoming and outgoing neighbours of node $i$ respectively, then the problem can be formulated as 3.

$$\text{maximize} \sum_{<i,j> \in E(G_{ImBal})} f_{ij}$$

such that

$$0 \leq f_{ij} \leq idealBal_{C_{ij}} \text{ and}$$

$$\sum_{k \in N^-(i)} f_{ki} = \sum_{j \in N^+(i)} f_{ij} \forall i \in V(G_{ImBal})$$  \hspace{1cm} (3)

Note that to minimize $I$ we have to maximize our objective function in 3. For this, the maximum possible amount should be transferred through each cycle in $G_{ImBal}$. In other words, no further transfer is possible, i.e., $idealBal_{C_{ij}}$ for at least one channel $C_{ij}$ along the cycle should become 0. But the maximum possible amount can be transferred in many possible ways, which may or may not minimize $I$. For example, the value of maximum possible amount for the imbalance graph in figure 2 is 5 which can be transferred in 4 possible ways (assuming integer amount only for ease of representation) as depicted in figure 4a, 4b, 4c and 4d. Note that we are not
allowing any edge to reverse its direction (handled by first constrain in 3)

**Definition III.1. (Cycle saturation)** A cycle in the imbalance graph is said to be saturated if it contains at least a link $C_{ij}$ with zero $\text{idealBal}_{C_{ij}}$ in the forward direction of the cycle i.e. a node cannot transfer any further amount through that cycle.

**Definition III.2. (Flow-dependent cycle)** A set of $n$ cycles $C_1, C_2, \cdots, C_n$ in the imbalance graph are flow-dependent if they share at least an edge and a node, along the shared edge, cannot send the amounts that it receives from other nodes along all the cycles In other words, it is not possible to saturate (III.1) all $n$ cycles.

**Theorem 1.** Given $n$ flow-dependent cycles $C_1, C_2, \cdots, C_n$ such that $L(C_1) > L(C_2) > \cdots L(C_n)$, where $L(C_1)$ is the length of the cycle. If $k$ out of $n$ cycles can be saturated (following constraints in eq 3) and order of saturation is $L(C_1), L(C_2), \cdots L(C_k)$ then the objective function 3 will be maximized.

**Proof.** We will prove this by contradiction. The value of the objective function in 3 is given as follows:

$$f_{\text{cur}} = f_1 \times L(C_1) + f_2 \times L(C_2) + \cdots + f_n \times L(C_k) \quad (4)$$

Let's assume that order of saturating the cycle is $C_1$ followed by $C_2$ followed by $C_k$ and so on, and $f_{\text{cur}}$ is not maximum i.e. there exist $f_{\text{max}}$ such that $f_{\text{max}} > f_{\text{cur}}$. If $f_{\text{cur}}$ is not maximum then more amount can be transferred. So one has to reduce the previously transferred amount from either of $C_1, \cdots, C_k$ and transfer it along another cycle, say $C_2$. If $x$ is the minimum amount value that one can transfer and we will reduce that amount from $C_i$ where $1 \leq i \leq k$ then updated value of equation 3 is given as follows:

$$f_{\text{max}} = f_1 \times L(C_1) + f_2 \times L(C_2) + \cdots + f_n \times L(C_k) + (x \times L(C_2) - x \times L(C_1)) \quad (5)$$

If $f_{\text{max}} > f_{\text{cur}}$ then $x \times L(C_2) - x \times L(C_1)$ should be positive which says that $L(C_2) > L(C_1)$ which is contradicting our assumption, so $f_{\text{cur}}$ should be maximum.

**Theorem 2.** If cycles in $G_{\text{ImBal}}$ are saturated preferring the longest cycle first, then the objective function 3 will be maximized.

**Proof.** $G_{\text{ImBal}}$ contains flow-dependent cycles, defined in III.2 and/or flow-independent cycles.

Case 1: For flow-independent cycles, i.e. the cycles which don’t share any edge or don’t affect the maximum amount transferred through each other, we can saturate them in any order. So, preferring the longest cycle will also result in maximizing the objective function in 3.

Case 2: For flow-dependent cycles, we can see from theorem 1 that preferring the longest cycle will give the maximum value of equation 3.

We can see that figure 4a gives the minimum value of $I$. We can see from theorem 2 that this is achieved if the node prefers to saturate the longest cycle over other. But, because finding the longest path is an NP-Complete problem, we propose an heuristic based re-balancing approach, REBAL.

**B. REBAL Design**

The idea is that every node selects the shortest cycle for each channel it belongs to and initiates the re-balancing request. This is because a standard polynomial-time algorithm finds the shortest cycle that consumes fewer resources than finding the longest cycle. Also, shorter cycles merge to form the longer cycle. For example in figure 3, consider node $A$ and $E$ sends the re-balancing request for amount 3, 5 by selecting the shortest cycle which is $C_1$:A-B-C-D-A, $C_2$:E-D-C-E respectively. Node $D$ receives the re-balancing request through $C_1$ ($R_{C_1}$) and through $C_2$ ($R_{C_2}$) for channel CD. Note that $R_{C_2}$ will increase its imbalance by 3 while $R_{C_1}$, will decrease its imbalance by 5, i.e., the overall decrease in imbalance for channel CD is 2. So it will serve both $R_{C_1}$ and $R_{C_2}$. After processing both requests, the resulting imbalance graph will be the same if we follow either approach (i.e., longest cycle first or REBAL). The decision on serving or denial of re-balancing request depends on the time instance it arrives. If $R_{C_1}$ and $R_{C_2}$ arrives $\Delta$ (time to complete a REBAL request) apart then node $D$ will process $R_{C_2}$ and it may reject $R_{C_1}$ based on new value of $\text{idealBal}_{CD}$ calculated using eq 2. Note that each node may receive many re-balancing requests. The node prefers to serve (saturate) the one for the longest cycle first. For example in figure 3 if there exist one more cycle $C_3$:D-F-G-H-I-D then node $D$ first saturate $C_3$ and then $C_1$.

The algorithm runs in two phases: 1) find the maximum amount $\text{amt}_{\text{max}}$ that can be transferred along the cycle 2) Sends re-balancing requests with $\text{amt}_{\text{max}}$. find the maximum amount $\text{amt}_{\text{max}}$: Each node initiate re-balancing request for all its outgoing edges in $G_{\text{ImBal}}$. Let $\text{amt}$ be the amount a node $i$ needs to balance its channel, say
If finds the shortest cycle, say $C_r$, for the channel $Ch_{out}$ which has some channel $Ch_{in}$ into $i$. It then sends a re-balance request $<+amt, cycle_{id}>$ through $C_{out}$ and $<-amt, cycle_{id}>$ through $C_{in}$. Here, $amt$ is calculated using equation 2 and $cycle_{id}$ of $C_r$ is calculated by appending node’s public keys involved in the cycle.

On the arrival of a request $<+amt, cycle_{id}>$ at node $i$, it checks if it can send $amt$ through the next channel in $C_r$. Similarly, for $<-amt, cycle_{id}>$, node $i$ checks whether it can receive $amt$ through the previous channel in $C_r$. If not, $i$ rejects the request and drops it, else forwards it to the next or previous node in $C_r$ correspondingly (lines 32-44 in algorithm 1). If $amt$ requested is in the direction of edge in $C_r$, then it is added to the list of positive amounts, else to that of negative amounts (line 7). Note that many nodes along a cycle will initiate the re-balancing request while the one with the minimum $amt$ among all, which is the maximum amount that can be transferred along the cycle, will get served finally. For example, in figure 3 if nodes $B$ and $D$ initiate a re-balancing request for $amt$ 3 and 10 respectively, then only $D$’s request will go through because $B$’s request is dropped at node $A$ and/or $D$.

On arrival of re-balancing request $<\pmamt, cycle_{id}>$, a node signs and forwards it if it can be served, else drops it. When request returns back to the source node $i$, it must be signed by all the nodes along the cycle and they lock the $amt$ for the time $\Delta$. Locking the amount stop the routers from processing the new transactions which are not already considered while calculating the imbalance factor in 1. Note that node will lock the amount only after signing both $+amt$ as well as $-amt$ message for particular $cycle_{id}$.

Sends re-balancing requests with $amt_{max}$: After getting the agreement on the maximum amount that can be transferred along the cycle, $i$ will send the re-balancing transaction $tx_{rebal}$ to itself through cycle $C_r$. $tx_{rebal}$ is similar to the normal transaction, except it reveals the source (and hence destination too). Further, it includes the request message reference.

On arrival of $tx_{rebal}$, the node verifies the re-balancing amount and the signature using the request message reference included in it. If a node hasn’t verified the respective message and/or $amt$ doesn’t match, then it drops $tx_{rebal}$ by reverting to the source with an "invalid transaction" message, else forwards it. Note that nodes don’t charge a fee to forward $tx_{rebal}$ with the assumption that they are going to benefit eventually from fees coming from transactions that will succeed as a result of re-balancing.

C. Re-routing at an intermediate node

and $\lambda$ before time interval $\Delta$, as $\Theta$ is the predicted value. In which case the node holds off on processing the new transactions (not considered in $\Theta$) and send the error message insufficient balance back to the sender. To avoid this REBAL allows an intermediate node to forward the transaction to next hop through an alternative route. and $\lambda$ before time interval $\Delta$, as $\Theta$ is the predicted value. In which case the node holds off on processing the new transactions (not considered in $\Theta$) and

<table>
<thead>
<tr>
<th>Algorithm 1 CycleSelection</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: $FinalTxList \leftarrow {}$</td>
</tr>
<tr>
<td>2: On Event&lt;Time expire and/or threshold reached&gt;:</td>
</tr>
<tr>
<td>3: $\text{call SendReq}$</td>
</tr>
<tr>
<td>4:</td>
</tr>
<tr>
<td>5: On Event&lt;Receive $mList$&gt;:</td>
</tr>
<tr>
<td>6: $\text{segregate }msg \text{ in } mList \text{ to } mList_{ch} \text{ of channel } ch$</td>
</tr>
<tr>
<td>7: $\text{call OnReceive} (mList_{ch})$</td>
</tr>
<tr>
<td>8:</td>
</tr>
<tr>
<td>9: $\text{procedure FINDSHORTESTCYCLE} (ch)$</td>
</tr>
<tr>
<td>10: $\text{return shortest cycle } c \text{ involving } ch$</td>
</tr>
<tr>
<td>11: $\text{end procedure}$</td>
</tr>
<tr>
<td>12:</td>
</tr>
<tr>
<td>13: $\text{procedure SENDREQ}$</td>
</tr>
<tr>
<td>14: $\text{for each } ch \text{ in } \text{channels do}$</td>
</tr>
<tr>
<td>15: $c \leftarrow \text{call findShortestCycle} (ch)$</td>
</tr>
<tr>
<td>16: $\text{Send the re-balancing req } &lt;\pmamt, c&gt;$</td>
</tr>
<tr>
<td>17: $\text{end for}$</td>
</tr>
<tr>
<td>18: $\text{end procedure}$</td>
</tr>
<tr>
<td>19:</td>
</tr>
<tr>
<td>20: $\text{procedure ONRECEIVE} (mList)$</td>
</tr>
<tr>
<td>21: $\text{calculate idealBal } \text{ using equation } 2$</td>
</tr>
<tr>
<td>22: $\text{edgeBal } \leftarrow \text{idealBal}$</td>
</tr>
<tr>
<td>23:</td>
</tr>
<tr>
<td>24: $\text{for each } msg \text{ in } mList \text{ do}$</td>
</tr>
<tr>
<td>25: $c \leftarrow msg \rightarrow c$</td>
</tr>
<tr>
<td>26: $\text{if } msg \rightarrow c \geq \text{ next edge weight in } c \text{ then}$</td>
</tr>
<tr>
<td>27: $\text{remove } msg \text{ from } mList$</td>
</tr>
<tr>
<td>28: $\text{end if}$</td>
</tr>
<tr>
<td>29: $\text{end for}$</td>
</tr>
<tr>
<td>30:</td>
</tr>
<tr>
<td>31: $\text{Divide } mList \text{ into two parts } pList \text{ and } nList$</td>
</tr>
<tr>
<td>32: $\text{sort } nList \text{ and } pList \text{ based on cycle length}$</td>
</tr>
<tr>
<td>33: $\text{reverse } nList \text{ and } pList$</td>
</tr>
<tr>
<td>34: $\text{pSum, nSum } \leftarrow \text{sum(pList}, \text{sum(nList)}$</td>
</tr>
<tr>
<td>35:</td>
</tr>
<tr>
<td>36: $\text{while } nList \text{ or } pList \text{ is non-empty do}$</td>
</tr>
<tr>
<td>37: $\text{while idealBal } \rightarrow pList \rightarrow &gt; next.amt \text{ is } +ve \text{ do}$</td>
</tr>
<tr>
<td>38: $\text{add } pList \rightarrow next \text{ to } FinalTxList$</td>
</tr>
<tr>
<td>39: $pList \leftarrow (pList \rightarrow next)$</td>
</tr>
<tr>
<td>40: $\text{idealBal } = \text{idealBal }\rightarrow \text{next.amt}$</td>
</tr>
<tr>
<td>41: $\text{end while}$</td>
</tr>
<tr>
<td>42:</td>
</tr>
<tr>
<td>43: $\text{while idealBal } + nList \rightarrow &gt; next.amt &lt; edgeBal \text{ do}$</td>
</tr>
<tr>
<td>44: $\text{add } nList \rightarrow &gt; \text{next to } FinalTxList$</td>
</tr>
<tr>
<td>45: $\text{nList } \leftarrow (\text{nList } \rightarrow &gt; \text{next})$</td>
</tr>
<tr>
<td>46: $\text{idealBal } = \text{idealBal }+ nList \rightarrow &gt; \text{next.amt}$</td>
</tr>
<tr>
<td>47: $\text{end while}$</td>
</tr>
<tr>
<td>48:</td>
</tr>
<tr>
<td>49: $\text{return } FinalTxList$</td>
</tr>
<tr>
<td>50: $\text{end procedure}$</td>
</tr>
</tbody>
</table>
the transaction to next hop through an alternative route.

**REBAL Host:** Each sender (host) finds a route to the destination while generating the transaction. Each generated transaction is bounded with a timeout value, and it will be treated as a failure if it fails to reach the destination by then. Each receiving host confirms the receipt by sending an acknowledgment to the source.

In REBAL, the source node/host specifies the maximum timeout value (cltv) that a transaction will be valid for and the maximum amount of fee it can offer to the intermediate routers to route the transaction.

**REBAL Router:** Each routing node maintains a queue of transactions which can’t be forwarded immediately (not even through an alternative route). It drops the transaction if the queue is full and sends back a negative acknowledgement.

In case an intermediate node runs out of funds to route the tx to the next hop, unlike LND, REBAL finds an another route to the next hop and sends tx through it. There might be more than one path between a router and next hop, it selects the one through the channel with a lowest imbalance factor.

Each intermediate router along the transaction path monitors the timeout value of the incoming transaction and sends a negative acknowledgement if it violates the timeout. Also, intermediate routers on the path deduct their routing fee and forward the remaining amount. An intermediate router denies (to forward) the transaction if the total fee exceeds the maximum amount of fee offered by the source.

**IV. Evaluation**

We now present results of a comprehensive evaluation of REBAL.

**A. Experimental setup**

We simulated REBAL using OMNeT++ (v5.6.2). We used the Spider [20] simulator to evaluate the state-of-art protocols listed below. Transactions in the simulation are implemented as messages passing between the nodes with simulated real-time delay [2] representative of a real network. All the experimental results reported use five independent runs.

**Base cases** We have considered four state-of-the-art routing algorithms for comparison:

- **LND** [17]: The baseline protocol that uses the path with minimum transaction fee and minimum hop count between the sender and receiver.
- **Landmark routing** [14], [19]: This algorithm works by assigning coordinates to nodes to find paths with reduced overhead between sender-landmark and landmark-receiver.
- **Spider and Waterfilling** [20]: It splits the transaction into smaller sub-units termed mttu. It also uses price variables to perform dynamic balance-aware routing. The waterfilling heuristic uses the path that has maximum available capacity.

**Workload** We have evaluated each strategy against the real-time Ripple workload downloaded on Nov 07, 2019 [22]. Transaction size distribution in real-time ripple workload has mean, median and largest value of 2438.79, 3.3, 880000 XRP (a unit of ripple currency).

**Topology** We downloaded the LN topology data along with the respective channel capacities on Oct 03, 2020 [7]. The data consists of 5630 nodes with 34069 edges. Similar to Spider, we Snowball sampled [12] the topology data to reduce it to 206 nodes and 5468 edges to make it amenable for simulation without making it too simple. We have converted the LND payment channel capacity to XRP to make it compatible with the Ripple real-time workload [6].

In addition, we simulated two synthetic topologies with 200 nodes each: a Watts-Strogatz small world topology [8] with 800 channels, and a Barabasi-Albert scale-free topology [1] with 1536 channels which follow the same channel capacity and delay distribution as that of LND topology.

**Metrics** We report on four metrics- (1) Transaction success ratio (TSR), (2) Transaction success volume (TSV), (3) Transaction latency (of successful transactions) and (4) Throughput. We compare these metrics for different channel capacities (obtained by multiplying them with different constants termed as capacity scale factors) and transaction generation rate (transactions per sec per node).

**B. Results**

**Parameter tuning:** We observe that a single re-balancing operation takes around 2 seconds for completion ($\Delta = 2$). So ideally, it should happen once every 2 seconds. But frequent balance locking will decrease the TSR, hence it is not advisable. Figure 7c depicts that, maximum value of TSR is obtained when the re-balancing rate ($R$) lies between 1/40-1/45 per second. A lower value of the re-balancing threshold ($Thr$) will reject re-balancing requests from other nodes. On the contrary, for high $Thr$ node will only serve the re-balancing request at the cost of increasing imbalance in its own channel. Figure 7d depicts that maximum TSR is obtained when the threshold is between 0.4-0.5. So for further experiments we have set $R = 1/40$, $Thr = 0.4$, $\Delta = 2$ sec and transaction timeout value is set to 5 seconds.

**Effect of transaction rate:** We can see from figure 5a that even for 10x more transaction rate (i.e. from 2 to 20 transactions per sec per node) REBAL performs much better compared to other routing strategies for LND topology. Landmark routing has 5% more TSR than REBAL however, REBAL processes a larger transaction volume (figure 6b). This is the result of locking up a large portion of channel balances, for re-balancing, resulting in insufficient funds for further transactions in REBAL. Figure 6b shows similar behaviour as that of success ratio for transaction volume but the extent of decrease is more compared to the transaction rate. This is because of the increase in number of high-valued transaction failure for higher transaction generation rate.

From figures 5b and 5c, we can see that REBAL outperforms all the state-of-the-art protocols in terms of TSR. For the small world network, TSR of REBAL degrades because of the lower number of edges for which both REBAL and actual transaction processing compete, hence REBAL is not
able to find an alternative path to next hop. Similar behavior is observed for TSV in case of synthetic topologies (results are omitted from current imprint because of space constraint). The throughput decreases with an increase in transaction generation rate because of a decrease in TSR, but it still outperforms all other protocols, as depicted in Figure 6d. REBAL experienced a slightly higher latency, specifically for higher transaction generation rate, because of locking some balance at the intermediate nodes as a part of re-balancing, as depicted in Figure 7b. Note that transaction latency is calculated for successful transactions only. A spike in transaction latency at a transaction generation rate of 5, as depicted in figure 7b, is because of an increase in waiting time at the queue.

**Impact of channel capacity:** Figure 5d shows that REBAL outperforms all the baseline protocols. But, REBAL has 7% lower TSV than LND for higher scale factor (above 8) as depicted in figure 6a. This is because REBAL locks some funds for the re-balancing process - consequently, it processes more smaller value transactions and higher value transactions timeout at the intermediate node while re-balancing completes. We also observe similar behavior for synthetic topology (results are omitted from current imprint because of space constraint).

Figure 6c shows that REBAL can process around 70 more transactions (per sec) compared to LND for the scale factor of 1. Also, it outperforms all the baseline protocols for higher scale factors. REBAL experienced negligible increase in the latency compared to the LND with the increase in the capacity scale factor. In fact, for the higher scale factor, it decreases a bit because of the availability of the shortest path as a result of re-balancing, as depicted in figure 7a.

In summary, we observe that REBAL outperforms all the baseline protocols at the cost of negligible increase in the transaction latency compared to LND.

**V. RELATED WORK**

We broadly categorize the related work in two parts: a) Routing b) Re-balancing algorithms

**Routing strategies:** Flare [18] uses the beacon node whose role is to enhance the other nodes’ visibility of the network and help them in route computation. Landmark routing approach, like SilentWhispers [14], [19], have highly connected nodes as landmarks. All transactions are routed from source to landmark and then from landmark to destination. However, both beacon and landmark based routing create centralization and are prone to targeted attacks. In Speedy Murmurs [19], each node is associated with a vector such that the hop distance between two nodes is reflected by their distance in the vector space. Ant routing [11] obfuscates the topology of the network and treats each node equally. All the above algorithms don’t take into account the dynamic channel balances thus leading to poor performance. Spider [20] considers channels’ balance for route selection. It uses a packet switched architecture, where payments are broken down into multiple smaller payments. This leads to increase in latency and bandwidth. [10] proposes the fee mechanism and uses that to make the routing decisions which keep the channels balanced. Flash [22] focuses mainly on increasing success-volume by breaking high value transactions into multiple smaller transactions, but doesn’t provide significant improvement in success ratio.

**Re-balancing:** Revive [13] proposes a centralized re-balancing solution that compromises the privacy of channel balances. [16] tries to make an equal contribution of the node’s funds to all its channels which doesn’t work for all the scenarios (specifically the case where most of the payment are unidirectional, i.e., from payer to merchants which is the case with production Lightning Network). [9] proposes active and passive re-balancing. Active re-balancing sends loop-back transaction from its high balance channel to its low balance channel while passive re-balancing charges the transaction fee inversely proportional to channel balance in the direction of payment.

VI. CONCLUSIONS & FUTURE WORK

This paper presents a novel channel re-balancing scheme for PCNs that outperforms state-of-the-art algorithms on real workloads without any workload assumption and compromising privacy.

There are 3 possible directions for future work - i) In this paper, we haven’t evaluated any possible attacks on the REBAL scheme. Future work might include a thorough analysis of possible attacks and their impacts. ii) For REBAL, we have used current and future transaction load as a parameter to decide when to trigger re-balancing. The estimation of the future load can be improved using prediction models, which can lead to better trigger decisions. iii) Atomic Multi-path Payments (AMP) allows a transaction to be split in multiple smaller transactions, such that the original transaction
is only considered complete if all smaller transactions are complete. Performance of REBAL in combination with AMP needs to be investigated.

REFERENCES


[23] Éva Tardos, “A strongly polynomial minimum cost circulation algo-
Exact and Efficient Protective Jamming
in SINR-based Wireless Networks

Dominik Bojko
Wrocław University of Science and Technology
Wrocław, Poland
Dominik.Bojko@pwr.edu.pl

Marek Klonowski
Wrocław University of Science and Technology
Wrocław, Poland
Marek.Klonowski@pwr.edu.pl

Dariusz R. Kowalski
School of Comp. and Cyber Sci.
Augusta University
Georgia, USA
dkowalski@augusta.edu

Mateusz Marciniak
Wrocław University of Science and Technology
Wrocław, Poland
Mateusz.Marciniak@pwr.edu.pl

Abstract—A majority of research in communication in wireless networks is devoted to maximizing information flow, improving connectivity, or making the system robust against physical perturbations such as jamming. In this work we study how intentional jamming can be used for assuring privacy of wireless communication under the popular Signal-to-Interference-plus-Noise-Ratio (SINR) model. The considered problem, called Zone-restriction with Max-coverage, is as follows: how to place a number of jamming stations in order to generate interference that block the signal of given genuine stations in a specified restricted area, i.e., by making the SINR value of the genuine stations’ signal below a pre-defined threshold in that area. In the construction of algorithms, we aim at optimizing both the accuracy – by minimizing the impact of the jamming stations to the area of genuine communication and by maximizing their influence to the area that should be jammed, as well as the energy consumption of the jamming stations. We present several solutions in various settings of the network, which often lead to challenging analysis even in relatively simple cases. Among others, we show that, surprisingly, it is possible to jam arbitrarily large areas by jammers using total energy arbitrarily close to zero.

Index Terms—Wireless sensor networks, SINR, information hiding, jamming

I. INTRODUCTION

In this work we pursue a non-traditional approach to wireless communication – how to (slightly) limit the genuine communication in order to protect it from eavesdropping? More specifically, we assume that there are some restricted areas, where we expect that the genuine wireless communication signal cannot be successfully received by any entity. At the same time, we would rather not affect the genuine communication that takes place outside the restricted areas. This problem can be motivated by many natural scenarios, spanning from military communication, preventing industrial espionage to protecting personal communication against eavesdropping, or providing wireless services in selected workspaces without being overheard in another ones.

One may be tempted to think that this problem can be solved by using standard cryptographic mechanisms. Note however that in many cases it is not possible to apply predeployment of any cryptographic material, especially in the case of real-life ad hoc systems. Moreover, in some environment the network devices are computationally restricted, and thus they cannot perform cryptographic operations required by even the simplest of cryptographic protocols. Finally, in many cases, e.g., on the battlefield, one needs to hide not only the content of the message, but also the fact that a communication takes place.

The outlined problem can be considered in various wireless communication models, however, in our paper, we consider popular Signal-to-Interference-plus-Noise-Ratio (SINR) model ([1]), in which the analysis of the problem is particularly challenging. On the other hand, the SINR model assumes that the power of the signal is fading with distance from the transmitting station and there is an interference from other network devices, which gives a model close to reality and acceptable from (most of) technology perspective. The SINR model has been proven significantly complex for analysis of even seemingly simple problems, such as answering the question if a chosen point can receive the transmission from any station (cf., [1], [3]). The complexity comes, among others, from the fact that transmission of a single station impacts the reception zones of even very distant stations.

In principle, the goal of limiting the communication in SINR model can be attained in two ways. The first is based on lowering the transmission power of stations, while the second is based on adding jamming stations to the original network that selectively reduce the reception zones by the introduction of an additional interference. In our paper we concentrate on the latter approach, as it has an inevitable advantage — we do not have to modify the initial network to apply it, we only add some jamming stations. This approach, sometimes called a friendly jamming, appeared in many previous papers that assumed other models of wireless networks (cf., [4], [5]), however the way how such policy can be applied is very different from our approach in SINR model.

Our contribution and paper organization: We consider several settings of SINR networks and construct algorithms for placing jammers to block the communication in a given restricted area while simultaneously minimizing unnecessary interferences in the rest of the network. Apart from such defined accuracy, we also consider the energy cost of jamming (total power of the jamming stations). It turns out, however,
that the formal analysis of the effects of adding jamming stations leads to difficult analytical problems even in some simple settings of just a few stations.

In Section II we introduce the model of communication and formalize the Zone-restriction with Max-coverage problem, stated above. Section III presents our results for various settings in a one-dimensional model. In particular, as a warm-up, Section III-A studies the simplest case of a network with a single broadcasting station and a single jamming station. In Section III-B we present algorithm for finding an optimal solution for jamming a single station by two jamming stations. Despite very simple formulation of the problem, the analysis of the algorithm turned out to be surprisingly complex. The correctness of this algorithm is presented in Theorem 1.

In Section IV we present the idea of noisy dust – accurate positioning of multiple stations with small energy levels. Specifically, in Section IV-A we introduce an adaptive noisy dust scheme and prove somehow surprising result that the total energy of all jammers can get arbitrarily close to zero. We also utilize a similar scheme of stripes, more universal method of jamming arbitrary fragments of space in Section IV-B.

In Section V, we utilize the stripes scheme from 1D-model and hexagonal tessellation to provide a method of jamming rectangular shapes in a 2D-plane.

Due to space limitation, some technical proofs are to be presented in the full version of this paper. The related work is discussed in Section VI, while Section VII presents conclusions and most important future directions.

II. MODEL AND PROBLEM STATEMENT

A. Model of SINR network

The SINR network is a tuple\(^1\) \(\mathcal{A} = (D, S, N, \beta, P, \alpha)\), where:

- \(D \in \mathbb{N}^+\) is the dimension of the network, in realistic scenarios limited to \(D \in \{1, 2, 3\}\),
- \(S = \{s_1, \ldots, s_n\}\) is a set of positions of stations in \(\mathbb{R}^D\),
- \(N > 0\) is an ambient, background noise (fixed real number),\(^3\)
- \(\beta \geq 1\) is the reception threshold (fixed real number),
- \(P: S \to \mathbb{R}\) is a stations’ power function; by \(P_i = P(s_i)\) we denote the power of station \(s_i\),\(^3\)
- \(\alpha \geq 2\) is a path-loss parameter (fixed real number).

For a network \(\mathcal{A}\), we define the SINR function for station \(s_i \in S\) and point \(x \in \mathbb{R}^D\backslash \{s_i\}\) as:

\[
\text{SINR}_\mathcal{A}(s_i, x) = \frac{P_i \cdot d(s_i, x)^{-\alpha}}{N + \sum_{s_j \in S \backslash \{s_i\}} P_j \cdot d(s_j, x)^{-\alpha}},
\]

where \(d\) is a \(D\)-dimensional Euclidean metric. For \(x \in S \backslash \{s_i\}\) we put \(\text{SINR}(s, x) = 0\). If the network \(\mathcal{A}\) is known from context, we simplify the notation to \(\text{SINR}(s, x)\) for a station \(s\).

\(^1\)We follow the [6] definition of SINR model (different ones can be found in literature).

\(^3\)The case \(N = 0\) is also considered, but we may encounter anomalies like infinite reception zones.

The SINR function calculates the relative power of the signal transmitted by station \(s_i\) in point \(x\), taking into consideration the network parameters, distance between \(s_i\) and \(x\) and the interference of all other stations in this network. The threshold \(\beta\) is the network-wide minimal SINR value enabling communication. A reception zone of station \(s\) in network \(\mathcal{A}\) is defined as \(H^s_s = \{x \in \mathbb{R}^D : \text{SINR}_\mathcal{A}(s, x) \geq \beta\}\). In simple words, \(H^s_s\) is the space where the station \(s\) is heard. The reception zones for parameter \(\beta \geq 1\) do not overlap, thus if some station \(s\) is heard at some point \(x\), no other station is heard at that point for this network configuration. For convenience, instead of using the full name of a station \(s_i\), in lower indices, we will be using its unique order number \(i\); \(H_i^A\) will be equivalent to \(H_i^A\). Finally, we define the range of station \(s\) for the network with positive noise value \((N > 0)\) as range\((s) = (\sqrt[N]{\frac{\beta}{P}})^{-1}\), which is the radius of maximal reception zone of \(s\) (in a network consisting of a single station).

B. Formulation of the problem of restricting transmissions

For a network \(\mathcal{A}\), there is given a restricted area \(\mathcal{R}\): a subset of space, wherein no station can be heard. In other words, in all points in \(\mathcal{R}\) the SINR function of all stations in a set \(S\) has to be lowered below the threshold \(\beta\). This can be done using two techniques. First is to modify the network parameters – we can increase the threshold value \(\beta\), decrease stations’ powers, or increase the path-loss parameter \(\alpha\). Second, we can use special jamming stations added to the network in order to generate interference and change the shapes of the reception zones of the original set of stations in network \(\mathcal{A}\).

An illustration of such approaches for a single broadcasting station is presented in Figure 1. The red rectangles represent the restricted area and the blue field is the reception zone of the original station. In Figure 1a we see the initial status – the blue space overlaps with the red restricted areas, possibly compromising the transmission from the station. Using the first approach – changing some of the network parameters to reduce the reception zone – the reception zone stops overlapping with the upper restricted area, cf., Figure 1b, but it still does with the lower one. In Figure 1c we follow the second approach – adding jamming station (yellow field in the lower restricted area). It generates enough interference to prevent the original station from being heard at lower restricted area, but it is still heard at the upper one. Finally, in Figure 1d both approaches are combined – jamming station and network parameters modification solve the problem for both restricted areas.

In order to formalize our approaches, assume that there is a network \(\mathcal{A} = (D, S, N, \beta, P, \alpha)\) and some subspace \(\mathcal{R} \subset \mathbb{R}^D\) representing the restricted area to be excluded from any communication involving stations from \(S\).

The problem of Zone-restriction with Max-coverage is defined as finding a set of jamming stations \(J = \{(s^{(J)}, P^{(J)})\}\) with positions in \(S^{(J)}\) and powers defined by the function \(P^{(J)}\) in such a way that in the resulting network \(\mathcal{A}^J = \langle D, S^{(J)} \cup S, N, \beta, P \cup P^{(J)}, \alpha \rangle\) the following condi-
The problem considered in this paper is specified as follows:

\[
\text{Cover}(\mathcal{J}, \mathcal{A}) = \left\{ \bigcup_{s \in \mathcal{J}} \left( H^A_s \cap (H^A \setminus \mathcal{R}) \right) \right\} \] 

where \(|A|\) denotes the volume of a set \(A\). Note that \(\text{Cover}(\mathcal{J}, \mathcal{A})\) is always properly defined, as long as \(N > 0\), and \(0 \leq \text{Cover}(\mathcal{J}, \mathcal{A}) \leq 1\). If the coverage equals to 1, then the jamming stations do not change the genuine reception zones apart from the restricted area. Our goal is to maximize the coverage.

Finally, we define the power cost function \(\text{Cost}(\mathcal{J}) = \sum_{s \in \mathcal{S}(\mathcal{J})} P^{(J)}(s)\), which measures the total power used by the jamming stations.

The problem considered in this paper is specified as follows:

For a given network \(\mathcal{A}\) and the restricted area \(\mathcal{R}\), find a set of stations and their powers, \(\mathcal{J} = (\mathcal{S}^{(J)}, P^{(J)})\), correctly protecting \(\mathcal{R}\) and maximizing \(\text{Cover}(\mathcal{J}, \mathcal{A})\).

Optionally, we also require minimizing \(\text{Cost}(\mathcal{J})\).

This work studies selected cases of \(\mathcal{S}^{(J)}\) and \(P^{(J)}\), e.g., a limit on the number of jamming stations used for jamming.

III. JAMMING IN ONE-DIMENSIONAL NETWORKS

In this section we consider one-dimensional SINR model \((D = 1)\). The restricted area is a union of some (potentially infinite) disjoint intervals. For example, in Figure 2a there are two broadcasting stations located in \(s_a\) and \(s_b\) with the restricted area given as a union of two intervals \((b^0_l, b^1_l)\) and \((b_r, \infty)\). Such scenarios is often considered in VANET networks [7]. We say that the network is uniform if all the stations (both from \(S\) and \(S^{(J)}\)) have identical power level (that is, \(P \cup P^{(J)} \equiv 1\)).

In uniform setting all the reception zones are convex (cf., [3]), which substantially simplifies reasoning. For instance, in Figure 2a the interval \((b^0_l, b^1_l)\) from the initial restricted zone can be replaced by \((\infty, b^1_l)\). Indeed, if a given station located to the right of \(b^1_l\) is not heard in \(b^1_l\), then due to convexity, it is not heard in any point \(x < b^0_l\) as well. Hence, we can model this constraint by a single point \(b_l = b^1_l\). This configuration is presented in Figure 2b.

In the case of uniform networks the power level parameter is redundant, thus we identify the set of jamming stations \(\mathcal{J} = (\mathcal{S}^{(J)}, P^{(J)})\) with the set of their positions \(\mathcal{S}^{(J)}\), or even with a single position when we deal with a single jamming station.

A. One side jamming

As an introduction, let us consider the simplest uniform network \(\mathcal{A}_0\) with a single broadcasting station \(s\) and a single jamming station \(s_J\). W.l.o.g. we can assume that \(s = 0\) and a restricted area \(\mathcal{R}_b = (b, \infty)\) for some \(b > 0\). This problem can be solved by placing the jamming station at \(s_J = b + r\) for some \(r > 0\) in order to block communication in \(\mathcal{R}_b\). Nevertheless, this jamming station also influences the initial part of the original communication of the station \(s\), which should be taken into consideration in order to prevent a reduction of the coverage (see Figure 3b).

Lemma 1. Let \(\mathcal{A}_0\) be a network and \(s_J\) be a single jamming station placed at: \(s_J = b + \sqrt{\frac{\beta}{b - a - \beta N}}\). Then

\[b^J = b + \sqrt{\frac{\beta}{b - a - \beta N}}.\]
Fig. 3: Single side jamming example. The reception zone is indicated with darker color.

1) $s_J$ correctly protects $R_b$, 
2) guarantees coverage $\text{Cover}(\{s_J\}, A_0)$ in 
$$\left\lfloor b + (\beta(N + \text{MinI}))^{-\frac{1}{\alpha}}, b + (\beta(N + \text{MinI})^{-\frac{1}{\alpha}}) \right\rfloor,$$
where $\text{MinI} := (s_J + \text{range}(s))^{-\alpha}$ and $\text{MaxI} := (s_J + b)^{-\alpha}$.

Proof: We skip a straightforward proof of point 1., based on monotonicity of the function $\text{SNR}(s, x)$ w.r.t. argument $x$.

The point 2. of the lemma follows from the limitation of the space, where some point $b_l < b$ can have $\text{SNR}(s, b_l) = \beta$, to interval $[-\text{range}(s), -b]$. The fact that $b_l > -\text{range}(s)$ is obvious and because of the monotonicity of the jamming station interference $I(s, x) = d(s, x)^{-\alpha}$, symmetry of the $s$ energy function $E(s, x) = d(s, x)^{-\alpha}$ regarding point $s$ and the fact that $d(s, b_l) > d(s, b)$, the inequality $b_l > -b$ follows.

We define maximal interference of $s_J$, achieved at point $-b$ and denoted as MaxI and minimum interference achieved at point $-\text{range}(s)$, denoted as MinI. Let us define the following, simplified, version of $\text{SNR}$ function:

$$\text{SNR}(s, x, I) = \frac{d(s, x)^{-\alpha}}{N + I},$$
where we essentially replace the jamming interference with some constant value $I$. By using it for $\text{MinI}$ and $\text{MaxI}$, we solve the equations:

$$\text{SNR}(s, x_l, \text{MinI}) = \beta \quad \text{SNR}(s, x_r, \text{MaxI}) = \beta,$$

getting the following results:

$$x_l = (\beta(N + \text{MinI})^{-\frac{1}{\alpha}}) \quad x_r = (\beta(N + \text{MaxI})^{-\frac{1}{\alpha}}).$$

Based on the monotonicity of $I(s, x)$, we know that $-x_l \leq b_l \leq -x_r$ (Figure 4), what means that the final reception zone of $s$ can be maximally of size of the segment $[-x_l, b]$ and at least of the size of the segment $[x_r, b]$. Additionally, the maximal reception zone of $s$, excluding restricted area has the following size:

$$|H_s^A \setminus R| = \text{range}(s) + b,$$

which finalizes the proof for the $\text{Cover}$ value bounds.

Fig. 4: Bounding the possible values of $b_l$ by segment $[x_l, x_r]$.

Let us note that a careful inspection of possible choices of the place for the jamming stations $s_J$ show that the chosen place is optimal in terms of coverage.

B. Two precise jammers in uniform network

Let us again consider the network $A_0$ and restricted area of the form $R_{b_l, b_r} := (-\infty, -b_l) \cup (b_r, \infty)$ for some $0 < b_l, b_r$ and two jamming stations. Assume that $b_l, b_r \leq \text{range}(s)$ (otherwise the problem either simplifies to a single side jamming or becomes trivial). Clearly one can use directly the positions described in the previous section to jam left and right independently. However, this way network is burdened with a significant and unnecessary reduction of the coverage.

If the jamming stations $J^* := \{-x^*, y^*\}$ guarantees $H_s^{J*} = [-b_l, b_r], then we call $J^*$ an optimal arrangement.

Below, we briefly present an iterative algorithm that returns positions $-x, y$ of jamming stations that guarantee correct protection of the restricted zone and are $\delta$-close to optimal arrangement $-x^*, y^*$ (see Theorem 1 for precise formulation).

1) Short description of the algorithm: Apart from parameters which describe the SINR model and the restricted area $R_{b_l, b_r}$, the algorithm takes a precision parameter $\delta$ as an input. We use the following notation:

- $C_i = \frac{1}{b} - \frac{N b_i^\alpha}{b_i}$, for $i \in \{l, r\}$,
- for $b > 0$ and $x > b - \frac{\alpha}{\beta}$, let us define $f(a, b, x) = 1 + (b - x^{-\alpha})^{-\frac{1}{\alpha}}$,
- $h(x) = f\left(\frac{b^\alpha}{b^\alpha}, C_l; f\left(\frac{b^\alpha}{b^\alpha}, C_r; x\right)\right)$.

Algorithm 1: AssignJammingStations($\delta$)

```
Algorithm AssignJammingStations($\delta$)
    \[ t_0 = 1 + \frac{b_r}{b_l} \left(1 + C_l^{-\frac{1}{\alpha}} \right), \]
    \[ t = \text{AlignPosition}(t, \delta), \]
    \[ D_f = \left| \frac{\partial}{\partial t} f\left(\frac{b^\alpha}{b^\alpha}, C_r; t\right) \right| b_l \]
    \[ \text{if } D_f \geq 1 \text{ then } \]
    \[ \delta = \frac{\delta}{D_f}, \]
    \[ t = \text{AlignPosition}(t, \delta) \]
    \[ y = f\left(\frac{b^\alpha}{b^\alpha}, C_r; t\right) - 1 \]
    \[ b_l, x = (t - 1)b_r \]
    \[ \text{return } (-x, y) \]

Procedure AlignPosition($t, \delta$)
    \[ \zeta = h(t), \]
    \[ k = \left\lfloor \ln \left(\frac{\zeta(t - \zeta)}{\ln \zeta} \right) \right\rfloor \]
    \[ \text{for } i \in \{1, \ldots, k\} \text{ do } \]
    \[ t = h(t) \]
    \[ \text{return } t \]
```

Without going into low-level details, the function $f$ entangles positions of optimally set jamming stations. Thence we assume the same in our approach. The function $h$ adapts position of the left jamming station to be closer to optimal. Initially we carefully set admissible position $t_0$ and then the lion’s share of the execution, Algorithm 1 rectifies $t$, which is responsible for the position of the left jamming station and at the end, it returns positions $-x$ and $y$, which are $\delta$-close to the optimal ones. In fact, there are two adaptive phases, the
first one is performed in order to guarantee that \(|x - x^*| < \delta\),
and the second, to affirm dual condition \(|y - y^*| < \delta\).

2) Result and ideas:

**Theorem 1.** Consider a uniform SINR network \(A_0\) with a
single station \(s = 0\) and parameters \(N > 0, \alpha \geq 1\) and a
restricted area \(R_{b,b}^*\), such that

1) \(0 < b_l \leq b_r \leq \text{range}(s)\),
2) \(C_r^{-\frac{\alpha}{2}} < b_0 = 1 + b_r^{-\frac{\alpha}{2}} (1 + C_l^{-\frac{\alpha}{2}})\),
3) \(C_l^{-\frac{\alpha}{2}} < b_r + b_l^{-\frac{\alpha}{2}} (1 + C_r^{-\frac{\alpha}{2}})\).

Then

1) there exists a unique optimal arrangement \(J^* = \{-x^*, y^*\}\)
2) \(\text{AssignJammingStations}(\delta)\) returns
\(J = \{-x, y\}\) such that:
- \(J\) correctly protect \(R_{b,b}^*\).
- \(|x - x^*| \leq \delta\) and \(|y - y^*| \leq \delta\) (we then say that the
arrangement \(J\) is \(\delta\)-close).

The proof of Theorem 1 is technical and it is postponed
to be presented in a full version of this paper. However, we
roughly sketch its idea in here as well. First, we show that
with the assumptions of Theorem 1, \(h\) function is rising and
concave. Then, by Banach fixed point theorem, we prove that
a sequence given by the relations \(x_0 = t_0\) and \(x_{n+1} = h(x_n)\)
converges monotonically to \(1 + x^*\), and can be used in order
to provide the positions of the left jamming station. Further
we obtain that then the position of the second jamming station
can be computed in terms of \(f\) function. Next we provide how
many steps in each adaptation phase are needed before termina-
tion of \(\text{AlignPosition}(\delta)\) in order to satisfy \(\delta\)-closeness
condition, which finalizes the proof. Let us mention that it can
be proved that each of the adapting phases are executed faster
than a Newton-Raphson method for a function \(h\) (which has
very fast, quadratic rate of convergence).

**IV. JAMMING BY NOISY DUST**

In this section we introduce the idea of noisy dust —
universal strategy for utilizing numerous jamming stations
for jamming arbitrary fragments of space. Let us consider
a network \(A\), where \(S = \{s_1, s_2, \ldots, s_k\}\) and \(s_i < s_j\) for
any \(i < j\), with \(P = 1\). Moreover, for any \(s \in S\), let
us introduce border points \(b_i(s)\), for \(i \in \{1, r\}\) such that
\(s - \text{range}(s) < b_l(s) < s < b_r(s) < s + \text{range}(s)\) and \(b_r(s) <
\quad b_l(s_j)\) whenever \(i < j\). Let \(R = \mathbb{R} \setminus \bigcup_{i=1}^{r} \{b_l(s_i), b_r(s_i)\}\) be a
restricted area. By a noisy dust we understand a set of stations
\(J = (S(J), P(J))\) placed onto the restricted area, where
\(P(J) \equiv p\) and \(S(J)\) is a disjoint union \(\bigcup_{s \in S} S(J)(b_i(s))\),
where \(S(J)(b_i(s))\) is a set of positions of jamming stations,
which intuitively are close to \(b_i(s)\) and correctly protects
the nearby restricted area (e.g. \(S(J)(b_i(s_{k-1}))\) correctly protects
the segment \((s_{k-1}, s_k + \text{range}(s_{k-1}))\cap (b_i(s_{k-1}), b_i(s_k))\).

Note that \(p\) intuitively should be small and let \(F(p) = (p\beta)^{\frac{1}{2}}\).

Below we present a theorem which describes a space which
a single station with small power \(p\) can correctly protect:

**Theorem 2.** Assume a network \(A\) with a station \(s_0\), a power
\(P_0 = 1\) and some point \(b\), such that \(s_0 < b < \text{range}(s_0)\). Let
\(P_j\) be the power of a jamming station placed at \(s_j = b + r\). Let
us assume that \(d(s_j, b) = d(s_0, b)F(P_j)\). Then \(s_j\) correctly
protects the segment \((b, s_j)\).

Proof: Realize that energy functions of \(s_0\) and \(s_j\) are
monotonic for the segment \((b, s_j)\): decreasing for \(s_0\) and
increasing for \(s_j\). Hence if we will show that SINR\((s_0, b) = \beta\),
it will be enough to prove the jamming property for that
segment. We can do it by rearranging the noiseless SINR
equation to obtain the relations below:

\[
\text{SINR}(s_0, b) = \frac{d(s_0, b)^{-\alpha}}{P_j} \frac{d(s_j, b)^{-\alpha}}{d(s_0, b)^{-\alpha}} = \beta,
\]

\[
F(P_j) = (P_j)\beta^{\frac{1}{2}} = \frac{d(s_j, b)}{d(s_0, b)} F(P_j) .
\]

With the addition of the noise, the desired property of correct
protection will not be affected.

This simple equation can be used in positioning schemes for
multiple jamming stations and let us freely calculate jamming
station configuration by modifying the border point position,
jamming station power and distance of the station from the
border point \(b\). We utilize it for two jamming schemes: an
adaptive noisy dust in subsection IV-A, where we optimize
the number of required stations by taking into account how
the jamming zone of station changes with distance from the
jamming station, and special noisy dust stripes in subsection
IV-B, which let us create universal jamming networks.
Finally, in subsection IV-C we present general lower coverage
bound for the noisy dust scheme.

**A. Adaptive noisy dust**

![Fig. 5: Positioning of multiple small stations.](image_url)
number of jamming stations). Slightly abusing notation, we will denote the distance between station $s$ and point $b$ by using $d(s,b) = b$.

**Theorem 3.** Let us consider a single station network $\mathcal{A}_0$ with a restricted area $\mathcal{R}_b = (b, \infty)$. Let

$$n = \left\lceil \frac{\ln \left( \frac{\text{range}(s)}{b_0} \right)}{\ln \left( 1 + \frac{F}{b} \right)} \right\rceil, \quad s_i = \frac{b(1 + F)^i}{(1 - F)^{i-1}}.$$ 

Then, for the set $\mathcal{J}_p = (\{s_i \mid i \in [n]\}, \{s_i \to p \mid i \in [n]\})$ of jamming stations:

1) $\mathcal{J}_p$ correctly protects $\mathcal{R}_b$,
2) $\lim_{p \to 0^+} \text{Cost}(\mathcal{J}_p) = 0$.

**Proof:** Using the Theorem 2, we can easily calculate the position of a single station based on the border point, ex. for the first border point $b_0 = b$ we will get $l_1 = b_0 + F$. Now for station $s_1 = b_0 + l_1$, we have to know how far it can protect the space on the side opposite to the jammed station — indeed, we are searching for $r_1$ from Figure 5. Consider point $b_1 = s_1 + r_1$ and the equation $\text{SINR}(s_1, b_1) = \beta$. Note that $\text{SINR}(s, x)$ is continuous for $x > 0$ and $\text{SINR}(s, x) \to 0$ as $x \to 0$, so $\text{SINR}(s, x) < \beta$ for $x \in (s_1 - l_1, s_1 + r_1)$. Since $\text{SINR}(s_1, s_1 + r_1) = \beta$ is a symmetrical to the case of $\text{SINR}(s_1, s_1 - l_1)$, we conduct very similar argument and obtain $r_1 = b_1 F = (s_1 + r_1) F$. Therefore $r_1 (1 - F) = s_1 F = (b_0 + l_1) F = b_0 (1 + F) F$, so we can obtain

$$s_1 = b_0 (1 + F), \quad r_1 = \frac{F(1 + F)b_0}{1 - F} = \frac{F s_1}{1 - F}, \quad b_1 = b_0 \left( 1 + F + \frac{F(1 + F)}{1 - F} \right) = b_0 \frac{1 + F}{1 - F}. \quad (1)$$

This concludes, that by setting $s_1 = b_0 (1 + F)$, we will correctly protect the interval $(s_1 - l_1, s_1 + r_1) = (b_0, b_1)$. Realize, that this is true also for network with noise ($N > 0$).

Now we want to extend this result to multiple stations. We use similar approach, where next station $s_2$ is positioned with the assumption that point $b_1$ is its border point, and it will draw out some interval $(b_1, b_2)$. We will put recursively subsequent stations, until we reach $\text{range}(s)$ and cover interval $(b_0, \text{range}(s))$ with small jamming fields. Note that we do not need to care about points $b_i, i > 0$, since the interference of multiple stations is bigger than for a single one and in result $\text{SINR}(s, b_i) < \beta$ for $i > 0$. By using equations (1) as a base, we can extend our notation for other stations as follows:

$$b_i = b_{i-1} \frac{1 + F}{1 - F}, \quad s_i = (1 + F) b_{i-1}, \quad l_i = F b_{i-1}, \quad r_i = \frac{F(1 + F) b_{i-1}}{1 - F} = \frac{F s_i}{1 - F} = \frac{l_i}{1 - F} \quad \text{for } i \in \{1, 2, \ldots, n\}. \quad (2)$$

Thus, we instantly get the positions of the jamming stations:

$$s_i = (1 + F) b_{i-1} = (1 + F) b_0 \left( \frac{1 + F}{1 - F} \right)^{i-1} = s_1 \left( \frac{1 + F}{1 - F} \right)^{i-1},$$

with the first one given by $s_1 = (1 + F) b_0$.

We are also interested in the number $n$ of jamming stations that we need to correctly drown out the region $(b_n, \text{range}(s))$. More precisely, we search for the minimal $n$ such that $b_n > \text{range}(s)$. As $b_n = b_0 \left( 1 + \frac{F}{b} \right)^n$, we need $\left( 1 + \frac{F}{b} \right)^n > \frac{\text{range}(s)}{b_0}$ to be fulfilled, hence we attain:

$$n = \left\lceil \frac{\ln \left( \frac{\text{range}(s)}{b_0} \right)}{\ln \left( 1 + \frac{F}{b} \right)} \right\rceil. \quad (3)$$

For our convenience we denote the internal value of the ceiling function in the right hand side of Equation 3 as $\bar{n}$. It finishes the proof for statement 1. of Theorem 3.

To evaluate the energy efficiency of the algorithm, depending on the power $p$ of the jamming stations, we have to analyze the value $\text{Cost}(\mathcal{J}_p) = np$ or easier to consider value of $\text{Cost}'(\mathcal{J}_p) = \bar{n} p$. Let $C(p) = \frac{1}{p^\beta}$ and $F(p) = (C(p))^{-\frac{1}{\alpha}} = (p^\beta)^{-\frac{1}{\alpha}}$. Notice that $\frac{\text{range}(s)}{b_0} = (\bar{n} p^\beta)^{-\frac{1}{\alpha}}$ is not dependent on $p$.

We see that as $p \to 0^+$, the limits of numerator $\text{num}(p)$ and denominator $\text{den}(p)$ of the full form of $\text{Cost}'(\mathcal{J}_p)$ tend to $0$:

$$\lim_{p \to 0^+} p \ln \frac{\text{range}(s)}{b_0} = 0, \quad \lim_{p \to 0^+} \ln \frac{1 + F(p)}{1 - F(p)} = 0.$$ 

We are going to use L'Hôpital's rule to find the limit of $\text{Cost}'(\mathcal{J}_p) = \bar{n} p$ as $p$ tends to $0$. In order to proceed, we need to calculate the derivatives of $F(p)$ and both numerator and denominator of the full form of $\text{Cost}'(\mathcal{J}_p)$:

$$\frac{\partial F(p)}{\partial p} = \beta (C(p))^{\frac{1}{\alpha} - 1}, \quad \frac{\partial F(p)}{\partial p} = \alpha C(p) = \frac{F}{\alpha p},$$

$$\frac{\partial \text{num}(p)}{\partial p} = \ln \frac{\text{range}(s)}{b_0},$$

$$\frac{\partial \text{den}(p)}{\partial p} = \frac{1 - F}{1 - F} \frac{\partial F(p)}{\partial p} (1 + F) + \frac{\partial F(p)}{\partial p} (1 + F) = \frac{\partial F(p)}{1 - F} (1 + F) = \frac{2F}{\alpha p (1 - F^2)}.$$ 

From L'Hôpital's rule:

$$\lim_{p \to 0^+} \text{Cost}'(\mathcal{J}_p) = \lim_{p \to 0^+} \frac{\text{num}(p)}{\text{den}(p)} = \lim_{p \to 0^+} \frac{\partial \text{num}(p)}{\partial p} \frac{\partial \text{den}(p)}{\partial p}$$

$$= \alpha \ln \frac{\text{range}(s)}{b_0} \lim_{p \to 0^+} \frac{p (1 - (\beta p)^{\frac{1}{\alpha}})}{2 (\beta p)^{\frac{1}{\alpha}}},$$

$$= \alpha \ln \frac{\text{range}(s)}{b_0} \lim_{p \to 0^+} p^{1 - \frac{1}{\alpha}} \frac{1 - (\beta p)^{\frac{1}{\alpha}}}{2 (\beta p)^{\frac{1}{\alpha}}} = 0,$$

therefore also $\text{Cost}'(\mathcal{J}_p)$ tends to $0$ as $p \to 0^+$, because $|\text{Cost}(\mathcal{J}_p) - \text{Cost}'(\mathcal{J}_p)| < p$, what concludes the proof. }

The idea presented here can be easily used also for multiple stations jamming scenario, still keeping its property of low energy usage.
B. Noisy dust stripes

In this subsection we present a scheme of positioning jamming stations with some fixed power in a form of jamming stripes – uniformly spaced stations, which provide protection for some chosen space interval. The next theorem shows how to form a single stripe:

Theorem 4. Let \( A_0 \) be a single station network with a restricted area \( \mathcal{R}_b = (b_0, b_1) \), where \( s < b_0 < b_1 < \text{range}(s) \). Let

\[
n = \left\lfloor \frac{b_1 - b_0}{2F(p)b_0} \right\rfloor, \quad s_i = b_0(1 + F(p) + (2 - i)F(p)) \cdot
\]

Then the set \( \mathcal{J}_p = \{ \{s_i| i \in [n]\}, \{s_i \to p| i \in [n]\} \} \) of jamming stations correctly protects \( \mathcal{R}_b \).

Proof: Let us look at the jamming station closest to the \( b_0 \):

\[
s_1 = b_0 + F(p)b_0.
\]

Basing on Theorem 2, it correctly protects interval \( (b_0, s_1) \). On the other hand, due to energy function of \( s \) being monotonically decreasing for \( x > s \), we know that any point \( x > s_1 \) requires less interference than point \( b_0 \) to be jammed. However, we have already generated enough interference at range of \( F(p)b_0 \) from \( s_1 \) to jam such points, thus the effective jamming interval of \( s_1 \) is \( (b_0, b_0 + 2F(p)b_0) \). Then each of the stations \( s_i \), for \( i > 1 \), is positioned with identical spacing and power, so they will preserve that property, filling whole \( \mathcal{R}_b \) with enough interference to protect it.

The idea of Theorem 4 with single stripe set, can be easily extended for jamming arbitrary networks — it only requires to find the closest points to protect (border points for each broadcasting station) and then configuring stripes accordingly.

C. Noisy dust coverage

The noisy dust coverage value can be limited from below in generic case as presented in Theorem 5:

Theorem 5. Assume a network \( \mathcal{A} \) and a restricted area \( \mathcal{R} \) as in section IV. Let \( \mathcal{J} \) be a noisy dust for \( \mathcal{A} \) with \( \mathcal{R} \). Then

\[
\text{Cover}(\mathcal{J}, \mathcal{A}) \geq \frac{\beta^{-\frac{1}{2}} \sum_{s \in \mathcal{S}} (N + \text{MaxI}_r(s, \mathcal{J})) - \frac{1}{2}}{\sum_{s \in \mathcal{S}} b_r(s) - b_l(s)} + \frac{\beta^{-\frac{1}{2}} \sum_{s \in \mathcal{S}} (N + \text{MaxI}_r(s, \mathcal{J})) - \frac{1}{2}}{\sum_{s \in \mathcal{S}} b_r(s) - b_l(s)},
\]

where, for \( i \in \{l, r\}, \)

\[
\text{MaxI}_r(s, \mathcal{J}) = p \sum_{s' \in \mathcal{S}^{(l)}(s)} d(s', b_i(s))^{-\alpha} + p \sum_{s' \in \mathcal{S}^{(r)}(s)} d(s', b_i(s))^{-\alpha}.
\]

Proof: We utilize the similar argument to this from the proof of Lemma 1. Remark that for \( s \), there always exists \( \varepsilon(s) \) that \( s \) is heard in its vicinity \( (s - \varepsilon, s + \varepsilon) \), irrespective of \( \mathcal{J} \). From the argument in the proof of Lemma 1, each reception zone of \( s \in \mathcal{S} \) is convex, so let \( x_l(s) < s \) and \( x_r(s) > s \) be such the points that \( \text{SNR}(s, x_l(s)) = \text{SNR}(s, x_r(s)) = \beta \). Therefore \( d(s', s) < d(s', x_r) \) and \( d(s', b_i(s)) < d(s', x_r) \), for any \( s' \in \mathcal{S}^{(l)}(s) \) and \( i \in \{l, r\} \). Therefore, for every \( s \in \mathcal{S} \), there exist \( y_l(s) \) and \( y_r(s) \) such that \( x_l(s) < y_l(s) < s < y_r(s) < x_r(s) \) and \( \text{SNR}(s, y_l(s), \text{MaxI}(s, \mathcal{J})) = \text{SNR}(s, y_r(s), \text{MaxI}(s, \mathcal{J})) = \beta \). Therefore \( y_l(s) = (\beta(N + \text{MaxI}(s, \mathcal{J})) - \frac{1}{2}) \) for any \( s \in \mathcal{S} \) and \( i \in \{l, r\} \), just like in Lemma 1. By summing all \( d(y_l(s), y_r(s)) \) we attain the numerator of inequality (4). The denominator is just the (finite) volume of \( R \setminus \mathcal{R} \).

V. Jamming in 2D

An extension of our problem to a two-dimensional case bears some new interesting challenges. For instance, reception zones are no longer reduced to intervals and can form complex shapes on a plane, which results in even more complicated analysis. In this section we propose a simple and natural generalization our ideas from section IV. Let us consider a simple network \( \mathcal{A} \) with a single broadcasting station and \( D = 2 \). On the plane, there is a broad class of reasonable restricted areas. For a sneak peak, let us examine a simple case of rectangular restricted zone with sides parallel to the axes: \( \mathcal{R} = \{(x, y) \in \mathbb{R}^2 | x_1 < x < x_r, y_b < y < y_l\} \), where parameters \( x_1 < x_r \) and \( y_b < y_l \) define the rectangle.

![Image](image_url)

(a) A blue area is a full effective jamming zone of a single station and a red one is a simplified jamming area reduced to a hexagon.

(b) An example of 2D jamming — a red rectangle is \( \mathcal{R} \) and small hexagons are the simplified jamming zones of added stations.

In order to provide 2D noisy dust, let us reuse the jamming stripes idea from subsection IV-B. First, we select a point \( b = (b_x, b_y) \in \mathcal{R} \), which is the closest from the broadcasting station \( s = (0, 0) \). Note that two-dimensional undisturbed reception zone is a circle. Instead of covering the rectangle with such the figures, we utilize a hexagonal tessellation, which covers \( \mathcal{R} \). For simplification we approximate a circle drawn out by a jamming station via a regular hexagon (see Figure 6a). This approach has a slight impact on the jammed space, however it still covers the area properly. If we assume that each jamming station has the same arbitrary chosen small power \( p \), then we can calculate a range of each jamming station as \( r = d(b, s)F(p) \) (with \( F(p) = (p\beta)^{\frac{1}{2}} \)). Finally, we cover \( \mathcal{R} \) with regular hexagons of side \( r \), with a first hexagon centered in \( b \). An argumentation from subsection IV-B together
with triangle inequality shows that such the approach correctly covers \( R \). An exemplary result is presented in Figure 6b.

The aforementioned technique can be easily extended to cases with restricted zones defined as arbitrary unions of the rectangles (also rotated). Nevertheless, methods of restriction of areas of more complicated shape require extensive research.

VI. RELATED WORK

The SINR model is well established in wireless networks, including older and newer technologies such as 5G mobile networks [8]. It was used as a measurement of connection quality [9], [10]. SINR is also widely used in theoretical models of wireless communication. Its geometrical properties were studied by Avin et al. [3], who analyzed the properties of reception zones under uniform SINR model, showing, among others, their convexity (the result heavily utilized in this paper). Non-uniform network properties were analyzed in [11], along with new point location algorithm, and in [12], where non-uniform SINR network model, combined with Voronoi Diagrams, proved to retain some of the useful properties of the uniform setting. There is also a large amount of work considering the basic problems under the SINR model, such as broadcasting [13], link scheduling [14] or power control [15]. Quickly evolving and growing wireless communication technology is prone to many security threats (ex. [16], [17]) and more then ever require effective and efficient solutions to protect users privacy. Most of such protective measures are based on cryptographic solutions ([18], [19]). The approach taken in this paper, using jamming stations as a part of the security mechanism, has been considered in [20]–[22] in the context of simpler models and the idea was proved to be practically feasible ([23]). Regarding the SINR model, in [24], the authors considered a model similar to ours, but focusing on a practical 2D scenario, where the space is divided into a storage, in which the legitimate communication is supposed to take place, a jamming space, where jammers can be placed, delimited by a fence, and the rest of the space, where the adversary can eavesdrop. In such settings, the optimization problems of jammers’ positioning and power assignment were presented with approximation algorithms working for continuous space. This work has been further extended in [4], where SINR model is used as a connection quality measurement, the solution is based on performing temporal jamming, and the channel quality is modeled by the bit-error probability. In our paper we extensively use some of previous (e.g. convexity of some SINR diagrams proved in [3]). Nevertheless, to the best of our knowledge, the general results of the current paper cannot be reduced to techniques used in previous literature. This is because the small changes in the problem formulation or the assumed model lead to significantly changed analysis in SINR-based networks.

VII. CONCLUSIONS AND OPEN PROBLEMS

We introduced the problem of Zone-restriction with Max-coverage in SINR wireless networks and considered three scenarios. Finding a general solution for this problem seems to be very challenging due to the complexity of the model. The examples of important open directions are: to consider two thresholds \( \beta' < \beta \), which models the case where the eavesdropping adversary may have a more sensitive receivers than regular users, cf. [4], and jamming only selected stations; add mobility and directional antennas to the model.

REFERENCES

Diffusion Analysis Improves Scalability of IoT Networks to Mitigate the Massive Access Problem

Erol Gelenbe and Mert Nakıp
Inst. Theoretical & Applied Informatics
Polish Academy of Sciences
Gliwice, Poland 44–100
Email: {seg, mnakip}@iitis.pl
& Yaşar University, Turkey
1 CNRS Labs: I3S Université Côte d’Azur
& Abraham de Moivre, Imperial College

Abstract—A significant challenge of IoT networks is to offer Quality of Service (QoS) and meet deadline requirements when packets from a massive number of IoT devices are forwarded to an IoT gateway. Many IoT devices tend to report their data to their wired or wireless network gateways at closely correlated instants of time, leading to congestion known as the Massive Access Problem (MAP), which increases the probability that the IoT data will not meet its required deadlines. Since IoT data loses much of its value if it arrives to destination beyond a required deadline, MAP has been extensively studied in the literature. Thus we first take a queuing theoretic view of the problem, and also use a Diffusion Approximation to gain insight into the IoT traffic statistics that affect MAP. Then we introduce the Quasi-Deterministic Transmission Policy (QDTP) which significantly alleviates MAP when the average traffic rate grows beyond a given level and substantially reduces the probability that IoT data deadlines are missed. The results are validated using real IoT data which has been placed in IP packets for transmission.

Index Terms—Internet of Things (IoT), Scheduling, Massive Access Problem, Queueing Theory, Quasi-Deterministic Transmission Policy, Diffusion Approximations

I. INTRODUCTION

Sensors for health applications, monitoring of areas which are of difficult access such as remote areas or large civil engineering structures, and geophysical characteristics [1]–[4] are among the numerous motivations for deploying sensor networks, giving rise to the Internet of Things (IoT) where the number of connected devices is rapidly increasing with the needs of autonomous systems, smart cities and smart vehicles [5], [6]. This increase in connectivity results in high traffic rates, causing congestion at the Physical Random Access Channels (PRACH) that service these systems [7], [8] which is known as the Massive Access Problem (MAP). When IoT data is used to control a complex distributed system, the needs for synchronization of the data to present a coherent view of the system can lead to further constraints regarding packet delays [9].

MAP was addressed in early research [10]–[12] through adaptive routing to reduce congestion in networks with multiple paths and gateways, and via information theoretic techniques to reduce the amount of traffic that is sent [13]. More recent work [7], [8], [14]–[29] commonly assumes that IoT traffic arrives at random, leading to solutions to MAP that include Access Class Barring (ACB) [8], [14], [16], [23]–[26], Cognitive Machine-to-Machine (M2M) communication [17], [19], game theory [18], clustering of devices [20], [27], data rate adaptation [21], Spread-spectrum, Non-Orthogonal Multiple Access (NOMA) [22], Interference Cancellation (SIC) [7], the use of CSMA/CA or slotted-ALOHA [15], [28] and collision awareness [29]. Other work [30]–[35] has suggested proactive network solutions for MAP, which can include techniques such as adaptive traffic offloading to storage areas or less congested gateways [36].

On the other hand it was empirically shown that the IoT traffic at the MAC-layer is predictable via specific machine-learning techniques that can help to identify distinct IoT traffic classes in [37]. Thus in [38], the predictability of IoT traffic is used to address the MAP in a “predictive network” framework with a Joint Forecasting-Scheduling (JFS) system that allocates the time slots of a single frequency channel for IoT devices based on a forecast of traffic generation. In [39] a Multi-Scale Algorithm is suggested to improve the performance of the JFS system while [40] has suggested the extension of JFS for multi-frequency channels. The results of this work have shown that predictive networks, in particular JFS, are promising for the solution to the MAP and that the lightweight scheduling heuristics are crucial since optimal schedules are difficult to achieve in practice due to their high computational requirements.

In recent work [41] we show that, for any given and fixed IoT packet traffic rate, the statistics of the actual instants at which IoT packets are sent has a significant impact on the number of packets which are received before the packets’ deadlines expire. We showed that “smoothing” the traffic by setting the individual packet transmission times in a uniformly distributed manner will considerably improve the system’s...
performance. We used queueing analysis to select the randomization instants, and to predict which packets would not meet their deadlines and should be eliminated before transmission to improve the chances that other packets meet their deadlines.

In the present paper, we seek the optimum instants of packet transmission which will maximize the probability that deadlines are met. Since exact analytical solutions for queueing systems with general arrival and service processes are not readily available, we first use Diffusion Approximations [42]–[44] to compute the probability that an IoT packet will meet its deadline. Based on this analysis we show that the probability that the packet does not meet its deadline is an increasing function of the Squared Coefficient of Variation (SCV) of the interarrival times. Thus we observe that for any given fixed average arrival rate, and any service time, the distribution of the IoT data, the best possible schedule (in the sense of minimizing the IoT date that misses its deadline) is deterministic, with data being sent at fixed time intervals whose value is identical to the inverse of the arrival rate.

However such a deterministic schedule cannot be applied exactly: indeed, at low traffic rates deterministic schedules may unnecessarily delay some of the data, while not providing any significant improvement in the amount of data that misses its deadline. Thus in the sequel we develop a Quasi-Deterministic Transmission Policy (QDTP) which is used when traffic rates are moderate or high, and is turned off for low traffic rates.

DAs also provide us with mathematically based insight with regard to the randomization result obtained earlier in [41], since the uniformly distributed randomization reduces the SCV of the interarrival times for the publicly available empirical data set which is in the range 1.6 to 2.18, down to approximately 0.33. Furthermore the QDTP results in an SCV close to zero, hence it also minimizes the probability of missing deadlines.

The rest of this paper is organized as follows. Section II describes the problem we wish to solve, while Section III introduces the diffusion process based analysis and introduces our main result concerning the Quasi-Deterministic Transmission Policy which aims to reduce the probability that deadlines are not respected. Section IV details the QDTP and evaluates it on the experimental data that we use from [45]. Finally section V summarizes our main conclusions and suggests further research.

II. The Probability of Meeting Deadlines

We model the channel that offers access to the IoT gateway as a single server queue with a service time that is related the length of each individual IoT packet that is being sent, with queuing theory techniques that are widely used [46]–[50].

Each IoT device that uses this gateway forwards packets structured from data in the form of “bursts” of multiple bits seen in the data set [45]. These bits would normally be packetized in some form, for instance based on the relatively recent LoRa-WAN standard which is used for IoT devices [51]. However we simplify matters and assume that the packetizations is based on IP packets with 21–Byte headers, which is not too different from LoRa-WAN, followed by payload bytes that contain the IoT device bit Bursts. The data transmission times are then selected in a manner that is compatible with the measurements reported in [51] were and effective data transmission rate in free space of 5400 bits/sec was measured; beyond that rate, it was found that bit error rates become significant. Based on the measured average packet length of 22.47 Bytes, we see that the average packet transmission time would be 33.33 msec, which will be used in the sequel.

The IoT traffic data set [45] that we have used contains the traffic patterns of 10,000 IoT devices, whose traffic generation patterns belong to one of the following classes: Fixed Bit Aperiodic, Fixed Bit Periodic, Variable Bit Aperiodic and Variable Bit Periodic. For a device $j$ which sends a burst of Bits at time $a_{j,n}$, the deadline beyond which the burst is of “no value” is denoted by $\Delta_{j,n} > 0$: thus the bits must arrive at the gateway and be served and transferred out of the gateway by time $a_{j,n} + \Delta_{j,n}$. The evaluations we will conduct are based on different fixed values of the deadline.

In turn, the gateway will then process the successive packets to extract the required information, or to pass on the packet to some other system such as a Cloud server.

Let $a_1 \geq 0, a_2 \geq a_1, \ldots, a_{n+1} \geq a_n, \ldots$ be the successive transmission times to the gateway of data packets by the various IoT devices that are connected, which is the First-In-First-Out ordered set of all data transmission instants from all IoT devices $\{a_{j,n}\}$.

On the other hand $S_1, S_2, \ldots, S_n, \ldots$ are the successive durations of occupancy of the channel to the gateway by these successive packets which we call the “service times”, so that in the numerical examples considered later in the paper we will take $E[S_n] = 0.33$ msec. Each packet will also have a deadline that is denoted $\Delta_n \geq 0$ for the $n$–th packet and which we discuss below.

The waiting $W_n$ experienced by the $n$–th packet is the delay that separates its arrival instant from its departure instant, and it is given by the well known Lindley’s recursive equation [46], [49]:

$$W_{n+1} = [W_n + S_n - a_{n+1} + a_n]^+, \quad n = 0, 1, 2, \ldots \quad (1)$$

where we take $a_0 = 0$ and $[X]^+ = 0$ when $X < 0$, and $[X]^+ = X$ if $X \geq 0$. The response time $R_n$ experienced by the $n$–th packet is then defined as:

$$R_n = W_n + S_n, \quad (2)$$

i.e. the waiting time plus its service time. Note that $A(t)$ the number of arrivals by time $t$ and the arrival rate of packets (when the arrival rate does not vary with time and the corresponding limit exists) are defined as:

$$A(t) = \sum_{n=1}^{\infty} 1[a_n \leq t], \quad \lambda = \lim_{t \to \infty} \frac{A(t)}{t}. \quad (3)$$
When the arrival instants \(\{a_n\}_{n\geq 0}\) constitute a random process, and the \(\{S_n\}_{n\geq 0}\) are random variables, we define the probability \(\Pi_n\) that the \(n-th\) packet misses its deadline, i.e.:

\[ \Pi_n = \text{Prob}[R_n > \Delta_n], \text{ and } \Pi = \lim_{n \to \infty} \Pi_n, \quad (4) \]

which is the probability that by the time a packet exits the gateway, its deadline has expired. Note that the “clock instant” when the \(n-th\) packet’s deadline expires is \(a_n + \Delta_n\).

In the sequel we will assume that the transmission channel characteristics are fixed, and that all packet transmission times are proportional to their sizes and are drawn from the same distribution, so that the service times are samples of the same random variable \(S\) with mean value \(E[S] = 0.33\) ms. On the other hand, we will assume that the arrival rate of IoT packets \(\lambda\) depends on the number of active IoT devices in the system, which we denote by \(M\).

The problem we now address is how to take appropriate scheduling decisions for each value of the workload \(\lambda\), and in particular we will consider how to optimize the instants at which the packets are actually transmitted, which we call \(t_n\), so as to minimize \(\Pi\). Note that \(t_n \geq a_n\), while \(a_n\) is the instant at which the IoT device data is available for transmission.

Thus in effect, the simple scheduling algorithm we describe, which we call Quasi-Deterministic Traffic Policy (QDTP), will transform each \(a_n\) into a new value \(t_n\), and \(W_n\) will be transformed into \(W^*_n\) where:

\[ W^*_{n+1} = [W^*_n + S_n - (t_{n+1} - t_n)]^+, \quad (5) \]

so that the new response time becomes:

\[ R^*_n = [t_n - a_n] + W^*_n + S_n, \quad (6) \]

since we need to include the delay introduced by the QDTP algorithm itself into the response time of the \(n-th\) packet that is being sent. Note that the new probability that the deadline is not met, also becomes:

\[ \Pi^*_n = \text{Prob}[R^*_n > \Delta_n], \text{ and } \Pi^* = \lim_{n \to \infty} \Pi^*_n, \quad (7) \]

A. Interarrival and Service Time Statistics

In the sequel we will assume that the channel characteristics between the IoT devices and the gateway are fixed, and that all packets are drawn from the same population having given packet length characteristics. Thus we assume that all \(S_n\) are independent random variables with the same probability distribution (i.e. they are i.i.d. or independent and identically distributed), with given mean \(E[S]\) and SCV:

\[ C_A^2 = \frac{E[S^2]}{(E[S])^2} - 1. \quad (8) \]

We also assume that we will operate under variable load \(\lambda\) (the arrival rate) but under stable conditions, i.e. \(AE[S] < 1\).

For a given \(\lambda = (E[a_{n+1} - a_n])^{-1}\), we will assume that we restrict ourselves to the i.i.d. case but we have the freedom of choosing the probability distribution function that best fits our needs to minimize \(\Pi\), and we define:

\[ C_A^2 = \frac{E[(a_{n+1} - a_n)^2]}{(E[a_{n+1} - a_n])^2} - 1. \quad (9) \]

In fact, our analysis will show that by setting \(C_A^2\) as small as possible, we can minimize \(\Pi\).

1) An Example of IoT Data Statistics: To illustrate some of the statistics obtained from real IoT data available from the Open Source Repository [45], in Figure 1 we show the distribution of the amount of data in Bits from data Bursts emanating from the IoT devices in the data set. We observe that the average number of Bits per Burst is 6.8167 and the SCV of the number of Bits per Burst is 1.9506. There are only 2000 Bursts with more than 50 Bits per Burst. For those Bursts with less than 50 Bits, the average number of Bits per Burst is 6.778 and the SCV of the number of Bits per Burst is 1.7633.

In Figure 2, we have used the same data set, but assumed that each burst from an IoT device is sent in the form of an IP packet with a 21–Byte header, and that the Bits belonging to the burst are stored in 8–Bit Bytes inside each packet, so that we exhibit the histogram of the length of the packets in Bytes.

In this data set, only 2000 packets are longer than 30–Bytes. We note that the overall average packet length is 22.7444–Bytes with an SCV of 0.0031. If we only consider just those packets whose length is less than 30–Bytes the statistics are hardly different, since the average packet length is 22.4695–Bytes with an SCV of 0.0028 and the amount of payload data transmitted per burst is on average around 12–Bits.

In the numerical examples of the next Sections III and IV, we normalize the average arrival rate \(\lambda\) based on the LoRaWAN bit-rate mentioned at the beginning of Section II, so that the average packet length of 22.4744 Bytes is transmitted in 33.33 msec, and the maximum traffic rate \(\lambda = 1\) corresponds to 30 packets/sec.
III. USING THE DIFFUSION APPROXIMATION

While the probability density of the response time in steady-state:

$$F_R(t) = \lim_{n \to \infty} Prob[R_n \leq t], \quad (10)$$

is known in the case when at least one of the interarrival or service times are exponential (G/M/1, M/G/1 systems) [49], there is no easy way to obtain it exactly for both arbitrary interarrival and service time distributions. Therefore, we will use a diffusion approximation [52], [53] to determine the probability $F_R(\Delta)$ that the response time of the station is shorter than the deadline $\Delta$, where we assume that the deadline is identical for all the IoT packets:

$$\Pi = \lim_{n \to \infty} \Pi = 1 - F_R(\Delta). \quad (11)$$

In the diffusion approximation, the number of customers $N(t)$ in a single server queue is modeled by the diffusion process $X(t)$ on the interval $[0, +\infty)$ with probability density function $f(x, t; x_0)$ which approximates the queue length probability $p(n, t; x_0)$ for the initial condition $x_0$ at $t = 0$. In steady-state, when $f(x) = \lim_{n \to \infty} f(x, t; x_0)$ the diffusion model yields [52]:

$$f(x) = \begin{cases} \frac{\lambda_0}{\alpha} (1 - e^{-\alpha x}), & \text{for } 0 < x \leq 1 \\ \frac{\lambda_0}{\alpha} (e^{-\alpha z} - 1) e^{\alpha x}, & \text{for } x \geq 1, \quad z = \frac{\beta}{\alpha}, \quad (12) \end{cases}$$

where:

$$\lambda = \frac{1}{E[a_{n+1} - a_n]}, \quad \mu = \frac{1}{E[S_n]}, \quad (13)$$

$$\beta = \lambda - \mu, \quad \alpha = \lambda C_A^2 + \mu C_B^2. \quad (14)$$

Note that $p_0$ is obtained by the relation $p_0 + \int_0^{\infty} f(x)dx = 1$, and its value is also known from queueing theory. On the basis of (12) we can also calculate the mean number of customers in the system:

$$E[N] \approx \int_0^{\infty} x f(x)dx$$

$$\approx \frac{\lambda_0}{\alpha} \int_0^1 x(1 - e^{-\alpha x})dx + \int_1^{\infty} x(e^{-\alpha z} - 1) e^{\alpha x}dx,$$

$$\approx \frac{\lambda_0}{\alpha} \left[0.5 - \frac{1}{z}\right] = \left[0.5 + \frac{C_A^2 \theta + C_B^2}{2(1 - \theta)}\right] \theta. \quad (15)$$

Finally, using Little’s law, the mean response time in steady-state can also be obtained:

$$E[R] = \frac{E[N]}{\lambda}. \quad (16)$$

We note that $E[R]$ is monotone increasing in $C_A^2$ and $C_B^2$ but we need the probability distribution function of $R$ to determine the probability of respecting the deadline. Therefore, we compute the response time with the use of the diffusion process [44]. It is can be obtained as a first passage time, since the time $X(t)$ needs to travel from the point $x = x_0$, corresponding to the queue length at the moment when a new packet joins it, to the first following instant when $x = 0$ when the packet has left the queue.

The probability density function $\gamma_{x_0,0}(t)$ of the distribution of first passage time from $x = x_0$ to $x = 0$, i.e. probability density that the process hits $x = 0$ the barrier at time $t$ after starting at $x = x_0$ for time $t = 0$, is given by [47]:

$$\gamma_{x_0,0}(t) = \frac{x_0}{\sqrt{2\pi \alpha t^3}} e^{-\frac{(x_0 - \theta t)^2}{2\alpha t}}. \quad (17)$$

For an arrival that finds the diffusion at level $x_0$ on arrival, this is simply the probability density function of the response time, because when this customer’s service is complete it will leave the queue in the empty state, and the diffusion at level $x = 0$.

Considering that the customer arrives when the system is at steady-state, the density of $x_0$ is simply $f(x_0)$, i.e. the stationary distribution of the diffusion process given in the expression (12).

We therefore obtain the probability density function of the response time as:

$$f_R(t) = \int_0^{\infty} x e^{-\frac{(x + \theta t)^2}{2\alpha t}} f(x)dx, \quad (17)$$

and the probability that the deadline is missed is simply:

$$\Pi = 1 - \int_0^\Delta f_R(\tau)d\tau. \quad (18)$$

In order to illustrate these results, we provide two figures that show how $\Pi$ varies with $C_A^2$, $\lambda$ and $\Delta$. Both Figures 3 and 4 also show, for comparison purposes, the actual values of the SCV of inter-arrival times $C_A^2$ of the real data set [45] with vertical bars, marked as ranging from approximately 1.6 to 2.18. The vertical bar for 0.33 which is close to the the value for the randomization policy of arrival instants developed in a recent paper [41], which reduces the value of $C_A^2$ by selecting the instants at which data is sent from individual devices using...
a uniform distribution over a deterministic interval of length \( \Delta - E[R] \), when \( E[R] \leq \Delta \).

In Figure 3, we show the probability of missing the deadline (y-axis) in logarithmic scale (to the base ten) for a given value of \( \Delta \), different values of \( \lambda \), and with \( C_A^2 \) varying over a wide range. Here the average service rate and the SCV of service time are both fixed to 1. We see that as \( C_A^2 \) and \( \lambda \) increase, \( \Pi \) increases significantly.

Similar results are shown in Figure 4 for different values of \( \Delta \) and a fixed value of \( \lambda \), and \( C_A^2 \) varying over a wide range, showing that as \( \Delta \) decreases and \( C_A^2 \) increases, \( \Pi \) increases significantly.

In Figure 5, we detail the probability of missing the deadline for the real data set of [45], plotted against the number of IoT devices \( M \) being used. For each value of \( M \), we also give the arrival rate \( \lambda \) normalized against the measured average bit rate from the \( M \) sources and the corresponding real values of the SCV of interarrival times \( C_A^2 \). We see that the value of \( M \) or of the corresponding \( \lambda \) has the principal effect in determining the measured fraction \( \Pi \) of the data transfers which do not meet the deadline.

![Fig. 3. The probability of missing the deadline (y-axis) logarithmic scale (to the base ten), estimated using the diffusion approximation, increases significantly as \( C_A^2 \) increases and the deadline \( \Delta \) measured in slots decreases, for a fixed but high value of the arrival rate \( \lambda = 0.8 \). The average service rate \( \mu \) and the SCV of service time \( C_B^2 \) are both fixed to 1.](image)

![Fig. 4. The probability of missing the deadline (y-axis) logarithmic scale (to the base ten), estimated with the diffusion approximation, increases significantly as \( C_A^2 \) increases and the deadline \( \Delta \) measured in slots decreases, for a fixed but high value of the arrival rate \( \lambda = 0.8 \). The average service rate \( \mu \) and the SCV of service time \( C_B^2 \) are both fixed to 1.](image)

![Fig. 5. The probability of missing the deadline (y-axis) logarithmic scale (to the base ten), estimated with the diffusion approximation, using the traffic statistics of the real data set of [45], is plotted against the number of IoT devices \( M \) (y-axis) that are being used. Note that each value of \( M \) corresponds to specific measured values of \( \lambda \) and \( C_A^2 \) shown along the x-axis. The corresponding arrival rate \( \lambda \) normalized against the measured average bit rate from sources is also shown. The real values of the SCV of interarrival times \( C_A^2 \) are also indicated.](image)

IV. QUASI-DETERMINISTIC TRANSMISSION POLICY (QDTP)

For \( 0 \leq a_1 \leq a_2 \leq \ldots \leq a_n \leq a_n+1 \leq \ldots \), the list of successive Burst dates of all the IoT devices, and \( \lambda \) the overall average arrival rate of the bursts, let \( t_n = a_n, n = 1, 2, \ldots \) be the instants at which the Bursts are actually sent from the IoT devices, and define the deterministic quantity \( D \leq \frac{1}{\lambda} \), so as to provide a throughput that can accomodate the average interarrival time. The “Quasi-Deterministic Transmission Policy” (QDTP) which is meant to reduce the value of \( C_A^2 \) and hence reduce \( \Pi \) the probability of missing deadlines, is defined as follows:

1) Set \( n = 1 \),
2) Send Packet 1 at \( t_1 = a_1 \),
3) Set \( n \leftarrow n + 1 \),
4) If \( a_n \leq t_{n-1} + D \), Send Packet \( n \) at \( t_n = a_{n-1} + D \),
5) Else if \( a_n > t_{n-1} + D \), Send Packet \( n \) at \( a_n \).
6) Go to (3).

To implement QDTP in practice, we would need to know \( \lambda \) in advance, which is possible when we have a fixed set of IoT devices, each of which sends data at pre-determined instants. Also, we either need the receiver to know the Packet generation times in advance, and to request all the senders to send their Packets at the instants defined by the QDTP, or the senders can know in advance their sequence number \( n \), “listen” to the successive sending instants and apply the
QDTP to determine when they need to send their own data. Thus the communication channel would need to be two-way or the individual IoT devices should also have the ability to sense the channel.

To evaluate the effectiveness of QDTP, we first conducted measurements of the SCV of interarrival times for the data set [45], both for the raw IoT data from [45], and for the same data using QDTP, for a varying number of active IoT devices $M$ as shown in Figure 6. We see that QDTP substantially reduces the SCV $C_A^2$ for all values of $M$.

Then in Figure 7, we show the relative frequency (empirical probability) of missing the deadline – for a very small value of the deadline $\Delta = 2$ – for both the raw data set of [45] and for the case where the QDTP is used with the same data set. QDTP obviously succeeds in considerably reducing the probability that the deadline is missed. Finally, in Figure 8 the data set in [45] with varying numbers $M$ of active IoT devices, and different values of $\Delta$, is used. We see that for all values of $\Delta$ above 2, QDTP reduces the empirically measured $\Pi$ to practically zero. However when the heuristic is not used, $\Pi$ tends to one as $M$ increases beyond a few hundred devices.

**V. CONCLUSIONS**

The MAP is one of the biggest challenges for the future of IoT networks and occurs when large numbers of devices access a single channel to reach their gateway, causing congestion at the entry points and leading to deadlines being missed for the data sent from IoT devices. In order to address this problem, predictive network designs have been used, where scheduling modifies priorities between devices and selects the instants at which IoT packets are transmitted.

In this paper, we use insight from queueing theory and Diffusion Approximations to design the QDTP scheduling heuristic that improves the scalability of IoT networks. Using a real data set of outputs from a large number of IoT devices, we have examined the relevant statistics and used them to study the effect of the interarrival time statistics on the probability that the IoT packets meet (or do not meet) their deadlines.

The performance of QDTP has then been evaluated extensively using the data set in [45] for a widely varying range of numbers $M$ of IoT devices, resulting in widely varying average arrival rates and many different deadline values. In particular, we have compared the performance of QDTP against the case where the original packet transmission dates (found in the experimental data set) are used.

These evaluations which use real data have demonstrated that QDTP can provide a very large reduction in the empir-
cally measured fraction of packets that miss their deadlines. Future work will combine QDTP with priority policies to attempt to obtain further improvements in IoT network performance, resulting in further alleviation of the MAP problem.

Furthermore, since queueing systems with deterministic arrivals have been studied by different authors [54]–[56] we expect that the insight provided by our work may lead to further useful interactions between classical queueing theory and the study and optimization of the Internet of Things.

ACKNOWLEDGEMENTS

The authors thank the anonymous referees for their very helpful recommendations and corrections, which have served to significantly improve this paper.

REFERENCES


Capri: Achieving Predictable Performance in Cloud Spot Markets

Bogdan Ghit
Databricks Inc.
Amsterdam, the Netherlands
bogdan.ghit@databricks.com

Asse Tantawi
IBM T.J. Watson Research Center
Yorktown Heights, NY, USA
tantawi@us.ibm.com

Abstract—Large cloud providers offer spot instances at attractive prices to improve resource utilization, resulting in a spot market where users bid for resources and providers alter prices dynamically. As prices surpass bid values, resources may be relinquished from users with low bids. Achieving predictable performance on spot markets is challenging for data analytics workloads because they are very sensitive to preemptions due to the excessive cost of recomputations.

We introduce Capri, a scheduling system for running cloud data analytics in spot markets in which users may experience periods of degraded performance. Capri dynamically predicts the functional relationship between bid and performance, thus helping with managing expectations and bid advice. We propose a new spot market abstraction called the bribe scheduler which delivers differentiated service levels based on bids. Capri uses a prediction mechanism built on a queueing approximation of the bribe scheduler. Capri dynamically estimates parameters to adapt the queueing model and provide accurate performance predictions in the face of time-varying workloads.

We collect measurements using Capri running two realistic workloads, IMDB and TPCDS, and demonstrate the accuracy of our approximation and parameter estimation methodology. We show that Capri achieves a median prediction error below 3% in bursty workloads. We find that Capri’s service level prediction is pessimistic as users are likely to experience better performance than they should receive for their bids.

I. INTRODUCTION

Data analytics applications are powered by a diverse array of computing platforms from clusters to clouds, and more recently, to spot markets [10], [17], [31]. While attractive for offering inexpensive servers, spot markets often fail to deliver predictable performance [4]. As spot prices fluctuate, applications are exposed to interruptions which may result in long delays, forcing the user to wait either for the price to drop or to recover lost state [5], [22], [29]. Because such delays are difficult to anticipate, users may feel discouraged and abandon the spot market [19]. In this paper we present Capri, a spot market scheduling system that adaptively determines the functional relationship between bid and performance for data analytics applications, thus enabling users to anticipate with good accuracy their service levels on the spot market.

As depicted in Figure 1 Amazon EC2 offers spot resources with interruption rates that are usually higher than 20% irrespective the instance type and availability zone. Two common solutions for running data analytics on a spot market are either to place large enough bids and reduce the risk of preemptions [23], [33], or to periodically save the application state to avoid losing work already done [22], [29]. Not only these solutions are cost ineffective, but they also don’t provide any performance expectation to users.

We seek to provide differentiated service levels to users based on their bids by means of a new spot market abstraction called the bribe scheduler. The concept of bribing for compute resources has been previously studied in the context of a single-server queue with users that buy their relative priority by means of a bribe or bid to gain access to a server [15]. Whereas in this model users are granted exclusive access to the server, in our cloud setting we aim at sharing resources across multiple applications at the same time. Therefore, we encapsulate the application runtime system on a set of spot containers that are prioritized by the bribe scheduler based on the user bids. To discriminate users based on their bids, the bribe scheduler employs preemption so that only the containers that have the highest bids are in service at any time. Unlike existing cloud spot markets, our bribe scheduler re-queues preempted containers instead of completely aborting them. A queued container may be deployed again whenever resources become available.

We want to be able to anticipate the relationship between bid and performance for data analytics applications. To achieve this goal, we design a spot advisor that is motivated by first principles analysis within the context of a theoretical model.
for studying bribe scheduling. To assess the performance of a job in our analysis, we use the average job slowdown defined as the ratio between the turn-around time of the job and its uninterrupted runtime. We generalize the closed-form formula of the job slowdown as a function of the bid from a single-server system with rigid jobs to a multi-server system with jobs consisting of multiple inter-dependent tasks. To do so, we incorporate model parameters and we take an adaptive approach for dynamically estimating those parameters by employing an extended Kalman filter on the slowdown and bid values measured over a period of time.

The main contributions of this paper are as follows:

1) We design CAPRI, a scheduling system for cloud data analytics that achieves differentiated service levels and dynamically estimates the functional relationship between bid and performance.
2) We deploy CAPRI on a public cloud and show that it anticipates with high accuracy the job slowdown as a function of bid and delivers better performance than what users should expect for their bids.

II. SYSTEM MODEL

In this section we describe the scheduling system and its model which provides differentiated service levels in a cloud spot market based on bidding.

We consider data analytics applications running in a multi-resource environment. Such applications are typically decomposed into multiple tasks that run on workers that are deployed on units of the available capacity called compute slots. A worker runs in a container and is responsible for the execution of the tasks assigned to it by the task scheduler. Figure 2 depicts the scheduling of jobs in this environment. An incoming job requests a given number of containers and is placed in a queue where jobs are ordered based on their bid values, so that lower bid values are in the back of the queue. Jobs wait in the queue until they are allocated at least one container before they start receiving service. While in service, the number of allocated containers may grow and shrink depending on the container availability and system load. A partially executing job has only a fraction of requested containers allocated and is progressing with degraded performance. When all containers of a job are revoked, the job goes back to the waiting queue.

Let job arrivals constitute a Poisson process with rate \( \lambda \) and let \( X \) be the random variable representing the job bid value. Without loss of generality, we assume that the bid value is in the set \( \mathcal{X} = [0, 1] \). The probability distribution function of \( X \) is denoted by \( B(x) = P[X \leq x], x \in \mathcal{X} \). We assume that \( B(x) \) is continuous and differentiable. Let \( S(x) \) denote the job slowdown defined as the ratio between the average response time and the average service time. An approximate expression for the job slowdown \( S(x) \) is given by [11]:

\[
S(x; B, \Theta) = \frac{1}{[1 - \theta_0(1 - B(x))^{\theta_1}]^2}, \tag{1}
\]

where \( \Theta = [\theta_0, \theta_1] \), acting as scale and shape parameters, respectively, \( 0 \leq \theta_0 < 1 \) and \( \theta_1 > 0 \). First, \( \theta_0 \) acts as a (virtual) replacement for the server utilization, which may not be available to an external observer. Second, \( \theta_1 \) captures additional model features, when compared to a simple M/M/1 bribe queue [15].

III. CAPRI SPOT MARKET

We want to provision transient spot resources to data analytics applications while providing performance expectations to these applications. To achieve this, we assign to each application a relative priority in the scheduler queue equal to the bid value placed by the user at submission time. Our spot market scheduling system called CAPRI consists of a bribe scheduler that provides differentiated service levels based on bids and a spot advisor that incorporates the relationship between the job slowdown and bid value described in Section II. Figure 3 depicts at a high-level the functional components and interfaces needed by CAPRI to operate in a real environment.

A. Job Management

We present the typical interaction between users and the CAPRI spot market through the job management framework. The data analytics job model assumed by CAPRI requires configuring a specialized runtime system which consists of a driver process that coordinates a set of workers used to execute the job. The driver and worker processes run in isolation inside containers which are allocated on compute and memory resources as specified by the user.

The driver generates an abstract representation of the user program that we call the logical plan of the job. In particular, the driver decomposes the job into multiple tasks some of which are independent and may run in parallel as they operate on a different data partition of the same dataset (map tasks) and others which may have data dependencies among them (map tasks before reduce tasks). In this way, we obtain a direct-acyclic graph of tasks (DAG) which is the logical plan of the job. The driver has an internal DAG-aware scheduler.
that further generates a physical plan of the job which is a mapping of tasks to compute slots.

CAPRI enables users to compete for the available resources through bidding. The bid is set upfront by the user and represents the cost per compute slot that the user will pay as long as its job is running excluding any waiting time that the job may experience. Because the amount of spot resources is limited, CAPRI may revoke a fraction of the resource quota allocated to a given user if there are higher bidders in the system. This is a key difference from existing spot markets where the user is charged based on a spot price that is completely under the control of the cloud provider.

**B. Bribe Scheduler**

To provide differentiated service levels to multiple users, CAPRI defines a new abstraction of a cloud spot market called the bribe scheduler. The bribe scheduler is a materialization of the scheduling policy analyzed in Section II. The bribe scheduler registers with a cluster-wide resource manager from which it receives a spot offer – a list of free resources on multiple (virtual) machines that CAPRI can use to serve incoming user requests.

We employ two complementary mechanisms that target a subset of the resource requests of existing spot users, and operate at different timescales. On the arrival of a job, CAPRI places the individual resource requests for containers in a waiting queue which keeps these requests ordered by the user bid with the driver always placed ahead of its workers so that it is scheduled first. As long as there are idle spot resources, CAPRI schedules the request at the head of the queue and allocates an isolated bundle of its resource request. The job can use this bundle to deploy either a driver if it doesn’t already have a running one or a worker otherwise.

We ensure that no waiting request has a higher bid than any other user request in service by means of preemption. When the spot offer is fully utilized and an arriving request places a higher bid than some of the user requests that are currently serviced, CAPRI may preempt multiple requests with lower bids in order to make room for the new request. Preempted requests are placed back in the waiting queue where they will wait for their turn as the higher bidders complete their work and leave the system.

CAPRI seeks to guide its scheduling decisions only on the user bid, and so it may not be able to service all resource requests of a given job at the same time. Because the job model assumed by CAPRI is elastic, it can run on as many resources as it can get. The job cannot run without a driver and it will immediately release all its workers when the driver is revoked. However, a job may continue running even if (a subset of) its workers are revoked by CAPRI. As a consequence, the work already done and lost due to preemptions needs to be rescheduled by the driver and restarted from scratch on the remaining set of workers.

CAPRI may fully or partially preempt the resource requests of a job multiple times until it completes. However, in order to limit their costs, users can control the lifetime of their jobs by setting a maximum number of driver restarts. When that limit is reached, the user will abandon the spot market even though its job is incomplete. In addition, users may guard their jobs from failures through a periodic checkpointing mechanism that can be incorporated into the runtime system of the job [29]. Deciding how and when to checkpoint their jobs based on the volatility of the spot market is however orthogonal to the work presented in this paper.

**C. Spot Advisor**

The key feature we propose is the spot advisor which is a mechanism that allows users to get insights into their expected level of performance on the spot market. CAPRI incorporates the relationship between the job slowdown and bid value which we have obtained in our analysis of the bribing queue in Equation 1. In order to adapt the model parameters to workload changes, CAPRI collects samples of the user bid, job runtime, and response time. While the bid is determined at submission time, the latter two samples are collected once the job completes. Using these samples, CAPRI dynamically estimates and updates the model parameters over time using an extended Kalman filter as described in [11].

Independent of collecting samples and updating the model parameters, CAPRI allows users to query the spot advisor. In particular, CAPRI can answer two types of queries. When the user provides a bid value, we can predict the average slowdown by replacing in Equation 1 the fraction of previous users that had a lower bid than the current user. Furthermore, we can provide a bid advice to a user that sets an expectation for the desired level of service. To so so, we employ Equation 1 from which we can obtain the user bid as a function of the job slowdown by simply inverting the function.

**D. Resource Allocator**

To operate the spot market, CAPRI assumes ownership over a set of machines offered by a cluster-wide resource manager. CAPRI uses those machines to confine the runtime systems of the incoming jobs in isolated bundles of resources such as containers. Such resource bundles are specified in terms of a
quota of compute slots and memory. In this section we present the job isolation properties of CAPRI.

CAPRI provides performance isolation between different jobs by leveraging container isolation mechanisms. Using containers enables fine-grained resource-sharing, and so we can confine the resource usage of a process tree to any amount of compute slots and/or memory. Whereas other isolation mechanisms exist, containers are attractive for CAPRI because they simplify the deployment of a job runtime system. In particular, the user only needs to set the instructions that CAPRI can employ to generate a container image which is a portable file that can be further used to instantiate containers.

CAPRI delegates the task of allocating resources at the granularity of containers to a container management platform. Besides creating and preempting containers, we also want CAPRI to be able to monitor the status of its containers in order to get accurate measurements of the job performance. However, because providing isolation is platform dependent, we make the resource allocation module pluggable. Therefore, we maintain a low-level interface that enables CAPRI to delegate the (de-)allocation and monitoring requests to the underlying container management platform.

IV. EXPERIMENTAL SETUP

In this section, we present the configuration of our cloud deployment and the data analytics workloads we use in our evaluation of CAPRI. Our experimental setup consists of a Kubernetes cluster with six Amazon EC2 t3.2xlarge virtual machines each of which is configured with 8 vCPU slots, 32 GiB memory, and a network performance of up to 5 Gbps.

We use two workloads which consist of mixes of queries from the IMDB [16] and TPCDS [20] benchmarks. The IMDB benchmark includes 111 different queries with input data from 21 tables with a total size of 3.6 GB stored in CSV format. Similarly, the TPCDS benchmark has 104 queries with input data from 24 tables with a total size of 1 GB stored in Parquet format. Each workload consists of a stream of 1,000 jobs with a Poisson arrival process and an imposed average load of 0.9. A job request consists of five containers with 1 vCPU and 1 GiB each and may be preempted at most four times before abandonment. The bid values in both workloads are sampled from a synthetic geometric distribution with $p = 0.1$ which is dominated by relatively low bids with a median value of 0.3 as shown in Figure 4a.

Figure 4b depicts the job size distribution in the IMDB and TPCDS workloads. On average, the jobs sizes in the TPCDS workload are twice as large as in the IMDB workload. The IMDB workload is dominated by short-interactive queries with more than 40% of all jobs taking less than 5 minutes and very few long-running jobs. In particular, roughly 20% of the jobs in the IMDB workload account for almost 50% of the total load. In contrast, the TPCDS workload is less variable than IMDB, with more than half of the jobs having sizes between 20 and 140 minutes. Running the IMDB and TPCDS workloads on CAPRI in our setup took 9.66 h and 15.91 h, respectively.

We implemented our CAPRI scheduling system using Spark on Kubernetes. To run Spark on Kubernetes we employ a job submission framework which bundles the Spark runtime system inside Kubernetes containers [2]. The job submission framework receives the user request for running an application with a certain bid and number of containers. The framework creates a Spark driver which in turn launches multiple workers all of which run inside Kubernetes containers. The driver and worker container scheduling are handled by our CAPRI scheduling system which maintains a waiting queue ordered by the container bid. In order to control the life-cycle of containers, CAPRI uses the Kubernetes Java Client [1], which provides access to the Kubernetes API for creating and deleting bindings between containers and cluster resources.

We packaged the spot advisor in an external library that dynamically tracks the model parameters as it learns of new data collected when jobs complete, namely the total queueing time and the wall-clock time during which the job receives
In this section we analyze different aspects of CAPRI’s operation in the face of time-varying workloads such as the prediction accuracy, cost savings, and preemption sensitivity.

A. Prediction Accuracy

We first evaluate CAPRI’s ability to deliver differentiated service levels based on the submitted bids and to anticipate the job slowdown attained given a bid.

Figure 5 compares the observed and predicted job slowdowns for the complete range of bid values with both workloads. Jobs that bid higher than 0.5 experience close to ideal slowdowns. As expected, jobs that bid below 0.25 are more likely to experience large slowdowns because they face a higher risk of preemption. However, we observe that the range of slowdowns is much wider in IMDB than in TPCDS. The IMDB workload is dominated by short-interactive jobs that take in the order of minutes and so, jobs that experience delays due to other jobs with higher bids typically have slowdowns much higher than 2. In contrast, the job sizes in the TPCDS workload are larger and more homogeneous, which means their slowdowns are less sensitive to delays. Jobs with relatively low bids may still be lucky and experience good service levels during periods of lower utilization. This is the case for the cluster of jobs in TPCDS that bid below 0.25 and experience slowdowns between 1 and 2. In contrast, such lucky jobs are less frequent in IMDB because the system has a higher utilization when running this workload.

Figure 6 shows the relationship between the predicted and observed job slowdowns. We use the R-squared value, also called coefficient of determination, to assess how close CAPRI’s job slowdown prediction is to an ideal prediction. The R-squared value ranges between 0 and 1 and provides a good measure of how well the observed job slowdowns are replicated by CAPRI’s prediction model. High R-squared values that are close to 1 denote a strong correlation between the observed and predicted values. An important observation is that job slowdowns in IMDB are more predictable than in TPCDS. The system load is more stable and less sensitive to job preemptions in IMDB. These short jobs are hit by preemptions relatively late in their execution and so, the amount of work they waste and need to recompute compensates for the decrease in utilization caused by jobs that reach the preemption threshold and get abandoned. In contrast, job preemptions in TPCDS result in larger fractions of work that is abandoned, thus decreasing more drastically the utilization of the system.

In Figure 7 we depict the distribution of the relative prediction error delivered by CAPRI with each workload. CAPRI anticipates the service levels of half of the jobs with less than 3% prediction accuracy for both workloads. Moreover, CAPRI’s prediction error is very dense in the range between 0 and 10% and very rarely exceeds 30%, which confirms our model is accurate and robust. We also see that the distribution of relative errors is more dense on the positive side, which means that CAPRI tends to overestimate the job slowdown. Thus, on average jobs may expect to experience lower slowdowns than their anticipated levels of performance.

B. Cost Savings

To analyze the cost of using CAPRI we investigate the evolution of the spot price over time. Unlike AWS spot markets, CAPRI is a free market in which users are charged based on their bids. Thus, in our experiments with CAPRI,
the spot market price is determined by the current system load and the user bids, rather than a hidden process that dynamically changes it periodically as in the case of AWS spot markets [4]. As we see in Figure 8, the spot price on CAPRI mostly fluctuates within the range between 0 and 0.3 and rarely increases above 0.3, which is the median bid value in our workloads, even though we operate the market under high system loads. When compared to the cost of the on-demand uninterrupted execution, CAPRI achieves an 80% and 65% cost reduction for IMDB and TPCDS, respectively.

C. Sensitivity Analysis

In all our previous experiments we have set the maximum number of restarts before a job is abandoned by CAPRI to 4. To understand the impact of the preemption-restart mechanism, we perform a sensitivity analysis of the number of retries a job may attempt after preemptions. To this end, we run a set of micro-experiments using 10% of all jobs in each workload. We set the number of job restarts between 1 and 16. Any job that is preempted more times than the predefined number of job restarts is completely abandoned by CAPRI.

Figure 9 depicts the slowdown achieved by jobs at different percentiles in the job slowdown distribution when the number of job restarts increases from 1 to 16 for each workload. We find that CAPRI is not overly sensitive to the number of restarts a job may attempt. We can see that in both workloads only 10% of all jobs have slowdowns that are higher than 2.5. More importantly, roughly 80% of all jobs experience close to ideal slowdowns when the number of retries is less than 8. Setting a lower number of retries per job increases the chance of job abandonment. Without any retries like on AWS spot markets, we have roughly 25% of all jobs that are completely abandoned in both workloads. However, setting maximum 16 retries per job reduces the number of abandoned jobs in CAPRI to only 4.

Allowing jobs to retry multiple times increases the amount of work that is lost due to preemptions and needs to be recomputed when jobs get restarted. Having extra work to recompute because of the preempt-restart mechanism adds more pressure on CAPRI which may lead to congestion and starvation of jobs that bid relatively low. When we set the number of retries to 4, the samples of the IMDB and TPCDS
workloads impose an average system utilization of 0.86 and 0.77, respectively. Because the system is almost never idle for the duration of the experiments the likelihood of having queueing delays is relatively high, hence the few jobs that have slowdowns above 2.

VI. RELATED WORK

In this section we summarize the related work from three aspects: auction mechanisms for resource allocation on the spot market, bidding strategies for spot resources, and dynamic parameter estimation techniques.

Auction Mechanisms. Running auctions to sell unused cloud resources may improve system efficiency and/or the provider revenue. To this end, efficient resource provisioning policies have been extensively studied in prior work from a theoretical perspective. In order to discriminate users accessing their services, cloud providers can employ two main mechanisms: priority-based allocation and preemption based on user bids [3], [15], [18]. Departing from the single-server queues, other spot market designs have advocated for modeling auctions with heterogeneous instances rather than the previous type agnostic approaches [25]. The analysis of the online auction problem in this setting resulted into a mechanism that is optimal with respect to system efficiency across the temporal domain and is also able to dynamically provision heterogeneous resources.

Another practical approach to auctions that allows users to bid for their resources and bundles heterogeneous machines while taking into account operational costs of servers, has been proposed to optimize the social welfare and the provider’s net profit [32]. Cloud providers may also operate proportional allocation schemes in which a user receives a fraction of resources that is proportional to the bid. In this area, game theoretic techniques have been used for service provisioning [6], price anticipation [9], and the introduction of resource allocation methods [21]. Merkat [8] is a resource manager that employs weighted proportional allocation [21] to dynamically allocate cloud resources among applications.

Bidding the spot market. Since Amazon EC2 released its spot markets in 2009, a sizable body of research analyzed the operation of such systems in the cloud. The characterization and prediction of spot prices of the AWS spot markets [4] inspired the design of user bidding strategies that optimize cost while also achieving uninterrupted service. Such strategies can be derived either by means of statistical analysis of historical spot prices [14], [33] or through more advanced modelling techniques such as Markov chains [7], [30]. However, recent
work has shown that complex bidding strategies [12], [26] are often ineffective in practice because cloud providers allow users to place maximum bids while charging them for a much smaller price that constitutes the spot price [23].

**Dynamic parameter estimation.** Modeling and predicting the performance of MapReduce-based applications has been studied in various settings [13], [28]. Common approaches build an estimator by choosing a relationship between an output variable that needs to be predicted and several system properties that can be measured and used for prediction. Most of these solutions build the estimator with machine learning techniques that use large amounts of training data based on low-level application performance characteristics [24], [27].

**VII. Conclusion**

**CAPRI** is an alternative spot market to existing public clouds for containerized data analytics that employs bribe scheduling to provide differentiated service levels to its users. **CAPRI** is designed from first principles in the context of an analytic performance model that estimates the functional relationship between performance and bid. With a control theoretical approach based on Kalman filtering, we dynamically tune this model using runtime performance and bid measurements. We incorporate our adaptive model in a spot advisor that **CAPRI** employs either to set an expectation for the performance of a job given a particular bid value or to suggest a minimum bid value required to attain a given service level.

Our evaluation shows that **CAPRI** rapidly adapts the performance model to workload changes and exhibits a strong relationship between the predicted and measured job slowdown performance. In particular, **CAPRI** achieves a median prediction error below 3\% which in most cases is pessimistic. In **CAPRI** jobs are likely to experience better service levels than what they should expect given their bids, which results in an average cost reduction of 65\% or higher. We also show that **CAPRI** is rather insensitive to the number of job restarts caused by preemptions.

**References**


Characterizing Machine Learning I/O Workloads on Leadership Scale HPC Systems

Arnab K. Paul†, Ahmad Maroof Karimi†, Feiyi Wang
Oak Ridge National Laboratory, USA
{paula, karimiahmad, fwang2}@ornl.gov

Abstract—High performance computing (HPC) is no longer solely limited to traditional workloads such as simulation and modeling. With the increase in the popularity of machine learning (ML) and deep learning (DL) technologies, we are observing that an increasing number of HPC users are incorporating ML methods into their workflow and scientific discovery processes, across a wide spectrum of science domains such as biology, earth science, and physics. This gives rise to a diverse set of I/O patterns than the traditional checkpoint/restart-based HPC I/O behavior. The details of the I/O characteristics of such ML I/O workloads have not been studied extensively for large-scale leadership HPC systems. This paper aims to fill that gap by providing an in-depth analysis to gain an understanding of the I/O behavior of ML I/O workloads using darshan - an I/O characterization tool designed for lightweight tracing and profiling. We study the darshan logs of more than 23,000 HPC ML I/O jobs over a time period of one year running on Summit - the second-fastest supercomputer in the world. This paper provides a systematic I/O characterization of ML I/O jobs running on a leadership scale supercomputer to understand how the I/O behavior differs across science domains and the scale of workloads, and analyze the usage of parallel file system and burst buffer by ML I/O workloads.

Keywords—Burst Buffer, Darshan, High Performance Computing, HPC Storage, IBM Spectrum Scale, I/O Characterization, Machine Learning, Parallel File System

I. INTRODUCTION

The I/O needs of high performance computing (HPC) workloads have been historically dominated by write-intensive modeling and simulation workflows. However, the increasing popularity of machine learning (ML) methods to solve complex problems in various areas, such as biology, astrophysics, and chemistry, has given rise to varied I/O patterns in large-scale data analysis and ML/deep learning (DL) workloads [1]. This diversity of I/O needs by HPC workloads warrants the need to characterize the requirements of I/O in modern HPC centers.

There has been a multitude of studies [2]–[5] dedicated to the characterization of write-heavy checkpoint/restart simula-

This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

† Made equal contribution to this work.

lation workloads. Recent studies [6]–[9] have also focused on analyzing the I/O requirements of popular ML methods, like deep learning. It has been typically seen that ML workloads have small read and write access patterns. However, there lacks a holistic understanding of the I/O characteristics of typical ML workloads based on different domain sciences and the scale of ML jobs on leadership scale HPC systems.

Application users typically use various I/O profiling tools to characterize their I/O workloads. These tools are valuable to provide insights into potential performance-tuning efforts and enable evaluation of I/O trends for the entire HPC cluster. Darshan [10]–[12] is one of the most popular HPC I/O characterization tools. It is designed to capture an accurate picture of application I/O behavior, including properties such as patterns of access within files, with minimum overhead. Darshan logs have been used in many studies to characterize the I/O needs of HPC workloads. However, darshan does not annotate the logs into ML and non-ML workloads. Therefore, it is a non-trivial task to classify the two kinds of HPC workloads (ML and non-ML), which would help in characterizing the different HPC I/O behavior.

Most ML jobs are perceived to be read-intensive with a lot of small reads while a few ML jobs also perform small writes. This kind of I/O behavior suggests that an in-system storage, like a burst buffer [13] will provide better I/O performance for ML workloads than a parallel file system, like IBM Spectrum Scale (GPFS) [14], where the performance is limited by a large number of metadata requests. On the other hand, a burst buffer is a fast and intermediate storage layer between the non-persistent memory of the compute nodes and persistent storage – the parallel file system. The burst buffer layer is configured to take a burst of read or write I/O at a very high rate. However, there has been no prior study on the usage of burst buffer and parallel file system by large-scale ML I/O workloads.

In this paper, we aim to fill the gap in literature by characterizing the I/O behavior of ML workloads based on different science domains, the scale of ML I/O job runs, and temporal trends over one year. We also analyze the usage of the parallel file system and burst buffer by ML I/O workloads. To this end, we study the darshan logs of 23,389 HPC ML I/O jobs spanning over one year (January 2020 - December 2020) on Summit [15] - the second-fastest supercomputer in the world according to the latest top-500 list [16]. IBM Spectrum Scale (previously known as
GPFS) [14] forms the parallel file system in Summit. In the remainder of the paper, we use the term ‘GPFS’ for the parallel file system and ‘BB’ for burst buffer.

Specifically, we make the following contributions with regards to the I/O behavior of ML workloads on a leadership scale HPC systems.

- Develop a technique to annotate ML workloads from darshan logs.
- Analyze the I/O behavior of large-scale ML workloads classified by different science domains and the scale of job runs.
- Study the usage of the parallel file system and burst buffer by ML I/O jobs and understand the scope of improvement in I/O performance.

From our study, we observe that ML workloads generate a large number of small file reads and writes, which is ideal for burst buffer. However only few science domains use burst buffer, and out of those, only some use the burst buffer efficiently. The temporal trend of ML workloads on Summit also indicates an exponential increase in the I/O activity from ML workloads by different science domains which is indicative of the future which will be dominated by ML.

### II. Background

#### A. Summit Supercomputer

The Summit supercomputer is based upon IBM AC922 system and deployed at the Oak Ridge Leadership Computing Facility (OLCF). It consists of 4,608 compute nodes. Each node is equipped with 2 IBM POWER9 (P9) processors and 6 NVIDIA Tesla V100 (Volta) GPUs. Also, each node has 512 GB of DDR4 CPU memory, and each GPU has 16 GB of HBM2 memory. An NVLink 2.0 bus connects each P9 CPU to 3 V100 GPUs. An InfiniBand EDR network with a fat-tree topology connects the nodes. A 1.6 TB NVMe device is present on each compute node to be used as node-local storage – burst buffer (BB). Summit is connected to Alpine, a 250 PB IBM Spectrum Scale (GPFS) file system. Summit can access Alpine at 2.5 TB/s in aggregate under a large, and sequential write I/O access pattern. Alpine is a center-wide file system and directly accessed by all other OLCF resources.

#### B. Darshan - HPC I/O Characterization Tool

Figure 1 provides an overview of the darshan’s architecture. As an application executes, the darshan instrumentation module for MPI, POSIX, and STDIO generates data records characterizing the application’s I/O workload within different components of the I/O stack. The instrumentation modules for the various components are registered with the darshan core library. During application shutdown, each module organizes its records, compresses it and writes those collectively to the log file. MPI-IO is recorded for every `MPI_File_read()` and `MPI_File_write()` calls. POSIX module records each `read()` and `write()` call. Many applications rely on text-based I/O in leadership class computing facilities. The STDIO module characterizes the `stdio.h` family of functions, such as `fopen()`, `fprintf()`, and `fclose()`.

### III. Methodology: Annotating ML Jobs

Darshan provides `darshan – util`, a collection of tools for parsing and summarizing log files produced by the darshan instrumentation module. We use one of its tools – `darshan – parser`, to parse the raw darshan logs of one year (January to December 2020) and get the filewise I/O statistics in each module (MPI-IO, POSIX, or STDIO) accessed by the application. The metadata of each parsed darshan log consists of `jobid`, `userid`, `start_time`, `end_time`, `executable`, `no_of_processes`, and `runtime`. These parsed logs are used to annotate ML jobs and study their I/O behavior.

To identify ML I/O workloads from the parsed darshan logs, we create a list of ML keywords from the most common ML libraries used on leadership scale HPC systems. By browsing through the executable and file names present in the darshan logs from the first two months, we find that most HPC ML workloads use libraries from either R or Python. Combining the knowledge of ML terminologies found in the darshan logs of the first two months and searching for the top ML/DL libraries used in R and Python, a list containing 42 ML keywords is created, which is shown in Table I. The list of keywords may not be comprehensive, but it is

---

**Table I:** List of 42 ML keywords used to annotate ML jobs from darshan logs.
IV. ANALYSIS OF ML I/O WORKLOADS

A. Classification Based on Science Domains

1) Distribution of ML I/O Jobs: ML I/O jobs are analyzed to identify the science domains that make the most use of ML techniques in their HPC applications. Figure 2 shows the distribution of ML jobs in different science domains along with the number of users and unique applications which make use of ML techniques in their HPC jobs. As explained in Section III, a total of 23,389 ML jobs were analyzed.

![Figure 2: Classification of 23,389 ML jobs by science domains. (Note: #users specifies the number of unique users submitting ML jobs, and #apps is the number of unique applications in each science domain).](image)

Observation: Biology constitutes the maximum proportion of ML jobs on Summit over a year. However, Computer Science has the maximum number of users that use ML approaches in their jobs.

2) Jobs using GPFS and BB: Figure 3 shows the number of ML jobs in each science domain that either has at least one file access on BB or all file accesses exclusively on GPFS.

![Figure 3: The number of ML jobs using burst buffer and GPFS classified by different science domains on Summit. (Note: BB jobs access at least one file from the BB. GPFS jobs access all the files exclusively from GPFS).](image)

3) Types of ML I/O Jobs: For every science domain that uses BB, we classify each job into one of the three categories: Read-Intensive (RI), Write-Intensive (WI), and Read-Write (RW).

\[ \text{result} = \frac{\text{ReadBytes} - \text{WriteBytes}}{\text{ReadBytes} + \text{WriteBytes}} \] (1)

Equation 1 specifies the job type based on the following:
- \(-1 \leq \text{result} \leq -0.5\): Write-Intensive (WI)
- \(0.5 \leq \text{result} \leq 1\): Read-Intensive (RI)
- \(-0.5 < \text{result} < 0.5\): Read-Write (RW)

Table II shows the percentage of ML jobs classified into the job type (RI, WI, RW) that either use exclusively GPFS or at least one of the file is in BB.

![Table II: Comparison of the percentage of read-intensive (RI) vs write-intensive (WI) vs read-write (RW) ML jobs using GPFS or Burst Buffer classified by the four science domains that use BB.](image)

Observation: Computer Science and Chemistry have a high percentage of RI ML jobs which have all files in GPFS. Therefore, a large percentage of read-heavy files from Computer Science and Chemistry can be migrated from GPFS to BB to improve the I/O performance of the ML workloads.

4) I/O Activity of ML I/O Jobs in GPFS and BB: Figure 4 shows the density distribution of I/O behavior (x-axis: bytes written, y-axis: bytes read) by ML jobs in both GPFS and BB classified by the four science domains that use BB. Based on the job types (RI, WI, RW) discussed above, the plot suggests that jobs which are nearer to the x-axis and further away from zero are WI, the jobs which are close to y-axis and far away from zero are RI, and the jobs in the middle are RW. Figure 4 is consistent with Table II which shows that Computer Science and Chemistry ML jobs which use BB...
Table III: The mean number of read and write calls per job made in each group of file access sizes classified by science domains. The file access sizes are grouped into <1MB, 1MB - 10MB, 10MB - 100MB, 100MB - 1GB, and >1GB bins. ~99% of the read and write calls are less than 10MB calls.

are mostly WI, while the ML jobs using BB from Materials and Biology are mostly RI.

Figure 4: Density distribution plots of I/O activity from ML jobs using either GPFS or BB classified by science domains.

Observation 1: ML jobs that use BB can either be classified as read-intensive or write-intensive, while a large majority of jobs use exclusively GPFS are read-write. This warrants further investigation into the read and write access sizes of the ML workloads, which is discussed in Section IV-A5.

Observation 2: A large number of ML jobs performing fewer reads and writes (closer to zero along the y-axis) exclusively use GPFS. This shows that many ML users believe jobs performing less I/O will incur a much higher overhead in copying files from GPFS to BB than the gain in I/O performance by doing I/O on BB.

5) Read and Write Access Sizes: Table III shows the mean number of read and write calls used per ML job in different access sizes classified by different science domains. The various file access sizes are grouped into five bins: <1MB, 1MB - 10MB, 10MB - 100MB, 100MB - 1GB, and >1GB. This analysis is important because large sequential read and write (higher sized bins) gives a higher performance from GPFS, while small reads and writes (lower sized bins) are more efficient in BB.

Observation: Almost 99% of the read and write calls for ML workloads are less than 10MB. This implies that burst buffer is an excellent candidate to improve I/O performance as a large number of small read and write requests overloads the parallel file system. There is a massive scope in I/O performance improvement, especially in the Physics domain, which comprises a healthy portion of ML I/O jobs on Summit as shown in Figure 2. Physics has a large number of small file accesses but does not utilize burst buffer as previously seen in Figure 3.

B. Classification Based on Scale of Jobs

1) Distribution of ML I/O Jobs: Based on Summit’s scheduling policy [17], the ML jobs are classified into three categories: small, medium, and flagship. The description for each category is shown in Table IV.

![Table IV](image-url)

Table IV: Summit scheduling groups by job node count.

Figure 5 shows the classification of small, medium and flagship scale ML jobs by science domains.

Observation: More than 78% of ML jobs run on less than 45 nodes with Biology and Computer Science having the maximum proportion of jobs. Biology has more than two-thirds share of the medium-scale ML jobs while Computer Science and Physics have the maximum share among the 77 ML jobs which run on flagship scale.

2) Jobs using GPFS and BB: Figure 6 shows the number of ML jobs that either has at least one file access on BB or all file accesses on GPFS.

Observation 1: Only ML users from Computer Science use BB for flagship jobs, while the ML users from other domains using BB; Biology, Materials, and Chemistry use it for jobs running on less than 922 nodes. The reason might be the multiple legacy codebases from the three science domains are in the process of being scaled up to include BB and improve I/O performance.

Observation 2: Typically, the number of jobs having at least one file on BB is lesser than the number of jobs using GPFS exclusively. Contrary to the pattern, the number of Computer Science jobs at the flagship scale, which uses BB, surpasses the number of jobs only using GPFS.

3) I/O Activity for Different Types of Jobs in GPFS and BB: The different job types: RI, WI, and RW, are described above in Section IV-A3. Table V shows how the different
types of jobs use GPFS and BB based on the scale of the ML jobs.

<table>
<thead>
<tr>
<th>Job Size</th>
<th>GPFS RI</th>
<th>GPFS WI</th>
<th>GPFS RW</th>
<th>BB RI</th>
<th>BB WI</th>
<th>BB RW</th>
</tr>
</thead>
<tbody>
<tr>
<td>Flagship</td>
<td>78.57</td>
<td>21.43</td>
<td>0</td>
<td>88.89</td>
<td>11.11</td>
<td>0</td>
</tr>
<tr>
<td>Medium</td>
<td>56.43</td>
<td>30.71</td>
<td>12.86</td>
<td>52.24</td>
<td>46.76</td>
<td>0.99</td>
</tr>
<tr>
<td>Small</td>
<td>43.57</td>
<td>24.31</td>
<td>32.12</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table V: Comparison of percentage of read-intensive (RI), write-intensive (WI), and read-write (RW) jobs for GPFS and Burst Buffer classified by the scale of job runs.

Observation 1: Flagship jobs which use exclusively GPFS are more read-intensive. Therefore, there is a scope of improvement in I/O performance if these RI ML jobs can migrate the read-heavy files from GPFS to BB.

Observation 2: Medium scale jobs which have at least one of their files on BB, use it more for WI jobs. This implies that ML users might not be confident of using BB for improved read performance and still prefer to use BB in a traditional manner, that is, for capturing periodic write bursts.

C. Temporal Trend of ML I/O Jobs

1) High-Level I/O Trend: The evolution of the I/O characteristics of ML jobs over a period of one year is shown by Cumulative distribution function (CDF) plots for read and write bytes in Figure 7.

Observation: More than 50% of the I/O happened in the later part of the year (starting mid-August). This suggests an exponential growth in the use of ML technologies in the HPC workloads and the importance of such a study to build better technologies to meet the future I/O needs of such ML applications.

2) ML I/O Behavior Trend of Different Science Domains: Figures 8a and 8b show the temporal trend of the percentage of bytes read and written by ML jobs on GPFS and BB over a period of one year classified by the four science domains that use BB.

Observation 1: Users from Computer Science started adopting BB for their ML jobs earlier than the other science domains. The steep jump in the usage of BB in Chemistry, Materials, and Biology suggests that only a few users used BB in a small time period. Figure 8b follows the same trend as Figure 7, where most of the ML jobs were run in the last few months of the year. Therefore, domain sciences outside Computer Science are starting to make use of BB in a more prevalent manner for their ML workloads, which will continually be on the rise. This means that better I/O optimization methods need to be developed to use burst buffer more efficiently.

Observation 2: The temporal trend for reads and writes is similar on both GPFS and BB, except for Computer Science for reads on BB, which suggests that there might have been benchmark runs from the users in Computer Science to test the read performance of BB at the start of the year, before using BB in the ML applications.

D. Usage of Burst Buffer by ML I/O Jobs

As seen in Section IV-A5, a large number of small reads and writes constitute the I/O behavior of ML workloads which is more suitable for BB than GPFS. Therefore, in
In this section, we analyze the usage of BB by ML jobs. Out of a total of 23,389 ML jobs, only 1046 jobs make use of BB.

1) Jobs having Common Files in BB and GPFS: Write-intensive ML Jobs typically use BB for temporary writes and then persist those to GPFS. On the other hand, read-intensive ML jobs copy files from GPFS to BB, and then the files are read from BB to improve the overall read performance of the ML job. Therefore, from all the 1046 ML jobs that use BB, we analyze only the jobs with common files in both GPFS and BB and observe the distribution of bytes read and written by these common files.

Figures 9a and 9b show the percentage of RI, WI, and RW jobs classified based on the science domains and the scale of job runs that have common files on both GPFS and BB. We found a total of 396 jobs that have common files on GPFS and BB.

Observation 1: Figure 9a shows that ML jobs from Biology performed persistent writes while the other types of jobs do not have any common files. This is completely opposite to the behavior exhibited by ML jobs from Chemistry, where the WI jobs do not have any files which are persisted. Jobs coming from Materials are only RI jobs, some of which use BB to improve read performance. ML users from Computer Science try to make optimal use of BB by having all three categories of jobs; RI, WI, and RW, use BB to either improve read performance or persist write-heavy files from BB to GPFS.

Observation 2: Figure 9b shows that the trend of common files across all the scales of jobs is dominated by Computer Science shown in Figure 9a. However, as more ML technologies are adopted by science domains other than Computer Science, the usage of burst buffer will be skewed towards the sub-optimal BB usage. Therefore, for an optimal system-wide I/O performance, ML users need to be well educated on the benefits of BB as well as novel I/O optimization techniques similar to [18], [19] should be developed which can transparently make use of BB without making changes to legacy code bases.

2) I/O Activity of Files Persisted from BB to GPFS: Figure 10 shows the I/O distribution of ML jobs which use BB to gauge the data size which are persisted from BB to GPFS. Total bytes read/written are the total bytes by the ML jobs which use BB. Burst buffer bytes read/written are the total bytes read or written from BB. Persisted bytes read/written shows the size of files which are persisted from BB to GPFS after they are read or written.

Observation: Files which perform read bytes from BB are not persisted back to the GPFS. Also, as expected almost all the write bytes on BB from Computer Science are persisted to GPFS. However, this behavior does not hold true for Chemistry which do not persist the write bytes that were performed on BB. This might imply that ML jobs from Chemistry might write into a lot of temporary files which need not be persisted.

E. ML I/O Job Performance on GPFS and Burst Buffer
In this section, we analyze the performance benefit that can be observed by using the BB more efficiently by ML I/O jobs.

1) Read vs Write Performance: Table VI compares the mean, median and standard deviation of read and write performance by files in the ML jobs on GPFS and BB.

<table>
<thead>
<tr>
<th>I/O Rate</th>
<th>GPFS</th>
<th>Burst Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>(MBps)</td>
<td>Mean</td>
<td>Median</td>
</tr>
<tr>
<td>Read</td>
<td>721</td>
<td>390</td>
</tr>
<tr>
<td>Write</td>
<td>782</td>
<td>257</td>
</tr>
</tbody>
</table>

Table VI: Comparison of I/O performance for files in the ML jobs on GPFS and Burst Buffer.

Observation 1: Both read and write performance on BB outperforms GPFS. BB gives a better improvement on read performance (4.95x) when compared to the write performance (3.48x) than GPFS.

Observation 2: Based on the mean and standard deviation on BB, we can conclude that 66% of the reads and writes on BB get I/O performance between 1GBps and 6GBps. This is consistent with the theoretical peak that can be obtained on BB from a single node on Summit – 2.1 GBps for writing and 5.5 GBps for reading [20].
The temporal trend of ML workloads shows that there is an exponential increase in the I/O activity from ML jobs from Computer Science, capturing burst of writes for write-heavy files. This suggests that there should be better optimization technologies that can transparently migrate files between GPFS and BB for the domain sciences who are less informed about BB.

Observation 2: ML jobs from Computer Science have a better mean performance combining file accesses on both GPFS and BB, compared to other domains. This implies that Computer Science makes better use of BB and GPFS, for example, not using BB for ML jobs doing small overall I/O, copying read-heavy files from GPFS to BB before doing the reads on BB, capturing burst of writes for write-heavy files on BB before persisting it on GPFS. This again suggests that there should be better optimization technologies that can transparently migrate files between GPFS and BB for the domain sciences who are less informed about BB.

V. DISCUSSION

The analysis of I/O behavior of 23,389 ML jobs provides valuable insights into the future of HPC storage systems. This study is focused on Summit – the world’s second-fastest supercomputer. However, the ML jobs studied in this paper represent other leadership scale HPC systems as well.

A. Lessons for domain scientists

- ML workloads from all science domains generate a large number of small file reads and writes, which is better suited for BB. However only few science domains use BB for their ML workloads. Therefore, domain scientists need to be trained to use BB for their ML workloads.
- HPC ML users from Computer Science makes use of BB more efficiently which results in the much better I/O performance compared to other science domains which also use BB. It is observed that ML workloads from Computer Science copies read-heavy files from GPFS to the BB and performs most reads from BB, and burst of small writes also happen on BB which is later persisted to the GPFS. This means that other domain scientists should also be trained to use BB more efficiently to yield better I/O performance.

B. Lessons for storage architects

- The temporal trend of ML workloads shows that there is an exponential increase in the I/O activity from ML jobs.
workloads which is indicative of the future which will be dominated by ML. Therefore better storage solutions need to be designed that can handle the diverse I/O patterns from future HPC ML I/O workloads.

- It can sometimes be difficult to modify legacy code bases from various science domains to include the usage of burst buffer. I/O optimization techniques must be developed that can help use the burst buffer transparently without modifying the application code.
- This study also provides guidance on the capacity of future HPC storage systems which will be dominated by ML workloads.

VI. CONCLUSION

There is an exponential increase in the use of ML technologies in HPC I/O workloads by various science domains. Therefore, understanding the I/O characteristics of ML workloads is very important for system architects, file system developers and HPC users. This paper analyzed the darshan logs of more than 23,000 ML workloads running on Summit. Our study showed that ML workloads generate a large number of small file reads and writes which is ideal for a burst buffer compared to a parallel file system. However not many science domains use burst buffer efficiently for their ML I/O workloads. Even the different scale of jobs affected the I/O usage of ML workloads, where only applications from Computer Science used burst buffer for running jobs on more than 921 nodes. This is a lesson for both domain scientists to use burst buffer more efficiently for their ML workloads, and also for storage architects to build better storage solutions for future large-scale HPC storage systems. In future, we will use this study to build a tool that can characterize ML workloads from darshan without the need of ML keywords.

ACKNOWLEDGMENT

We would like to thank Hyogi Sim for his suggestions and inputs for the paper. This research used resources of the Oak Ridge Leadership Computing Facility, located in the National Center for Computational Sciences at the Oak Ridge National Laboratory, which is supported by the Office of Science of the DOE under Contract DE-AC05-00OR22725.

REFERENCES


