BenchCouncil Transactions on Benchmarks, Standards and Evaluations

TBench
Volume 1, Issue 1
2021

Editorial

Original Articles

ISSN: 2772-4859

Original Articles

Conference

BenchCouncil Transactions on Benchmarks, Standards and Evaluations (TBench) is an open-access multi-disciplinary journal dedicated to benchmarks, standards, evaluations, optimizations, and data sets. This journal is a peer-reviewed, subsidized open access journal where The International Open Benchmark Council pays the OA fee. Authors do not have to pay any open access publication fee. However, at least one of the authors must register BenchCouncil International Symposium on Benchmarking, Measuring and Optimizing (Bench) (https://www.benchcouncil.org/bench/) and present their work. It seeks a fast-track publication with an average turnaround time of one month.

The Preface to Special Issue of 2021 BenchCouncil International Symposium on Benchmarking, Measuring and Optimizing

This volume contains the papers presented at Bench 2021: the BenchCouncil International Symposium on Benchmarking, Measuring and Optimizing, held virtually in November 2021. The Bench conference has three defining characteristics. First, it provides a high-quality, single-track forum for presenting results and discussing ideas that further the knowledge and understanding of the benchmark community. Second, it is a multi-disciplinary conference. This conference edition attracted researchers and practitioners from different communities, including architecture, systems, algorithms, and applications. Third, the program features both invited and contributed talks. The Bench symposium solicits papers that address pressing problems in benchmarking, measuring, and optimizing systems.

The call for papers for the Bench 2021 conference attracted a large number of high-quality submissions. At least four experts reviewed each paper during a rigorous review process, and the program committee selected 11 papers for the Bench 2021 conference. The papers in this volume include revisions requested by program committee members. Bench 2021 had four keynote lectures. Jack Dongarra, professor of the University of Tennessee, presented "High-Performance Computing: Where We Are Today And A Look Into The Future". Dr. Peter Mattson, the Senior engineer of Google, presented "Building what ML needs". Dr. Wanling Gao, associate professor of the Chinese Academy of Sciences, presented "AI Scenario, Training, and HPC AI Benchmarks". Dr. Vijay Janapa Reddi, associate professor of Harvard University, introduced "AI Tax: Motivating the Need for End-to-end Performance Analysis of ML Tasks". There are four Tutorials in Bench 2021, which are "Automated Benchmarking of cloud-hosted DBMS with the benchANT", "High Frequency Performance Monitoring via Architectural Event Measurement", "Advanced MPI Programming", and "DataBench Toolbox for Pipeline-based selection of Big Data and AI Benchmarks". There are four BenchCouncil Distinguished Doctoral Dissertation Award Finalists. Dr. Romain Jacob from ETH Zurich, Dr. Pei Guo from the University of Maryland, Baltimore County, Dr. Belen Bermejo from the University of Balearic Islands, and Dr. Kai Shu from Illinois Institute of Technology.

During the conference, the International Open Benchmark Council (BenchCouncil) sponsored four different types of awards to recognize significant contributions to the area of benchmarking, measuring, and optimizing. The BenchCouncil Achievement Award recognizes a senior member who has made long-standing contributions to the field. Jack Dongarra was named the 2021 recipient of the achievement award. The BenchCouncil Rising Star Award recognizes a young researcher who demonstrates outstanding research and practice related to the conference's theme. Dr. Peter Mattson, Dr. Wanling Gao, and Dr. Vijay Janapa Reddi were named the 2021 recipients of the rising star award. The BenchCouncil Best Paper Award is to recognize a paper presented at our conference with high potential impact. And this year Prof. Tony Hey generously donated to the BenchCouncil Award committee to spin off the best student paper award. And this award is to a student as the first author who publishes a paper that has a potential impact. In 2021, we had three best paper finalists. Junqi Yin, Aristeidis Tsaris, Sajal Dash, Ross Miller, Feiyi Wang, and Arjun Shankar from Oak Ridge National Laboratory received the Bench 2021 best paper award for their paper "Comparative Evaluation of Deep Learning Workload for Leadership-class Systems". Jiaqiang Liu, Jingwei Sun, Zhongtian Xu, and GuangZhong Sun from the University of Science and Technology of China received the Bench 2021 Tony Hey Best Student Paper award for their paper "Latency-Aware Automatic CNN Channel Pruning with GPU Runtime Analysis".

We are very grateful to all the authors for contributing such excellent papers to the Bench 2021 conference. We appreciate the indispensable support of the Bench 2021 Program Committee and thank its members for the time and effort they invested in maintaining the high standards of the Bench symposium.

Resit Sendag
University of Rhode Island, USA

Arne J. Berre
SINTEF Digital, Norway

Lei Wang
ICT, Chinese Academy of Sciences, China

Chen Liu
Clarkson University, USA

Axel Ngonga
Leipzig University, Germany

Contents

BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100012

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Call for establishing benchmark science and engineering

Jianfeng Zhan

Institute of Computing Technology, Chinese Academy of Sciences, China

ARTICLE INFO

Keywords:
Benchmark science and engineering
Origin and evolution
Measurement standard
Standardized data set
Standard benchmark hierarchy
Consistent benchmarking
Meta-benchmark

ABSTRACT

Currently, there is no consistent benchmarking across multi-disciplines. Even no previous work tries to relate different categories of benchmarks in multi-disciplines. This article investigates the origin and evolution of the benchmark term. Five categories of benchmarks are summarized, including measurement standards, standardized data sets with defined properties, representative workloads, representative data sets, and best practices, which widely exist in multi-disciplines. I believe there are two pressing challenges in growing this discipline: establishing consistent benchmarking across multi-disciplines and developing meta-benchmark to measure the benchmarks themselves. I propose establishing benchmark science and engineering; one of the primary goals is to set up a standard benchmark hierarchy across multi-disciplines. It is the right time to launch a multi-disciplinary benchmark, standard, and evaluation journal, TBench, to communicate the state-of-the-art and state-of-the-practice of benchmark science and engineering.

1. The origin and evolution of the benchmark term

Benchmarking is common practice in all industries, and indeed in many areas of life [1]. For example, an Olympic sprinter or fund manager or IT product manager may compare themselves against a benchmark or a close competitor to evaluate their performance. Unfortunately, the benchmark term independently evolves in multi-disciplines and has related but different implications. This section investigates the origin and evolution of the benchmark concept.

I find that the modern benchmark concept (close to its current definition) first appeared in measurement science [2] in the form of bench mark (two words separated by a space). For example, in geodesy, a bench mark is a mark whose height, relative to datum, has been determined by leveling—the operation to measure differences in height between established points relative to a datum [3]. Later, this concept is extended into multi-disciplines.

In the computer discipline, one of the earliest benchmarking effort [4] dated back to 1962 in the Auerbach Corporation's Standard EDP Reports. Joslin defined this benchmarking effort as "a routine used to determine the speed performance of a computer system" [4]. The reports included reporting performance data using typical benchmark tasks – many basic functions – but based on the vendor's published data without stipulating that the benchmark must run on the system under test. Around 1965, Joslin [5] stated that the most important question in computer evaluation should be "how long will it take this system to process my workload (my computer application)?". This exploring methodology produced the concepts of workload modeling, application benchmark, synthetic benchmarks, and standard benchmark, which are still used nowadays [4]. These concepts seem abstract, not directly related to the bench mark concept, though having some connections. The primary reason may be that the computer is a new thing at that time.

The followings are simple explanations of these concepts. Workload modeling is selecting a representative sample set of programs from the entire real workloads [4], which is a critical factor ensuring the benchmark quality. An application benchmark is a mix of programs to be run on several different computer configurations to obtain comparative performance in terms of handling the specific applications [5]. Because of the difficulty (cost) of porting real applications across different systems, in 1969, Bucholz [6] argued a greater degree of abstraction – a synthetic benchmark to imitate the real application – is necessary to make comparisons across different systems practical. The rising costs of synthetic benchmarks motivated the standardization of benchmarks. In 1976, a group of government and industry personals was formed to ascertain the possibility of a standard benchmark library [7], which was the first try in this regard.

As a general term, in the 1987 edition of the Oxford Reference Dictionary, the benchmark is defined as a surveyor's mark indicating a point in a line of levels, a standard or point of reference [3]. The editors obviously did not consider the benchmark concept that appeared in the computer discipline, but their benchmark definition is similar to that in geodesy we referred at the beginning of this section; Zairi et al. [3] thought this definition is the beginning of today's use of the word benchmark in the management discipline.

E-mail address: zhanjianfeng@ict.ac.cn.
URL: https://www.benchcouncil.org/zjf.html.

https://doi.org/10.1016/j.tbench.2021.100012

Available online 21 December 2021

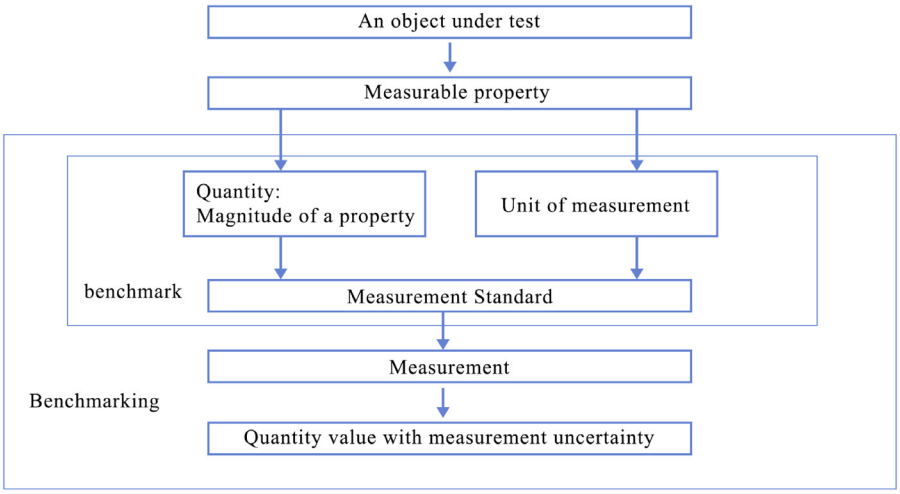


Fig. 1. The interpretation of the first category of the benchmark from the perspective of metrology [8,9].

In the management discipline, the Xerox Corporation was the pioneer of benchmarking [3]: its roots began in 1979, evaluated itself externally through this process which became known as competitive benchmarking. This benchmarking research and practice [3] encompassed an in-depth, ongoing study of best competitors, including detailed reverse engineering of competitor products, technology processes, what they achieved and how they did it, and a tear-down analysis of operating capabilities and features of competing products. This benchmarking practice is very similar to the computer discipline's benchmark-driven performance engineering in terms of the principle. The latter tries to disclose the root causes of the performance bottlenecks of and optimize the computer systems considering the specific workloads.

Gradually, benchmarking was extended as a strategic quality tool to all aspects of the business and progressively integrated into the management process [3]. In this context, Zairi et al. [3] defined it as the continuous process of measuring products, services, and processes against the industry best practices that lead to superior performance.

2. Five categories of benchmarks

This section investigates five categories of benchmarks in multi-disciplines. My intention is not to provide a consistent or unified benchmark definition. Instead, I try to reveal the essence of the benchmarks in five different scenarios. I leave the discussion of consistent benchmarking in the following two sections.

The first category of the benchmark is a measurement standard. In the computer discipline, the Linpack benchmark is of this category, which is widely used to report the performance of a high-performance computer. I provide an interpretation of this category from the perspective of metrology. The Joint Committee for Guides in Metrology (JCGM) [8] defines a measurement standard as a realization of the definition of a quantity, with stated value and associated measurement uncertainty, used as a reference. As shown in Fig. 1, a benchmark realizes the definition of a quantity, the unit of measurement, the measurement methodology, and the reference implementation with stated measurement uncertainty. A quantity is a measurable property of the object under measurement, like length, energy, etc. Benchmarking covers two phases: the design and implementation of the benchmark and measuring the object's properties with the benchmark.

The second one is the representative workloads that run on the systems under measurement. The application benchmarks or synthetic benchmarks in the computer discipline, discussed in Section 1, are of this category. They provide the design input to the system design and implementations. They do not necessarily meet the stringent definition of measurement standards, but they are also used to evaluate systems. For example, in the computer discipline, many deep learning workloads (algorithms) are random with poor repeatability [10,11]. Deep learning is a kind of artificial intelligence (AI) workload. However, they are representative workloads that cannot be overlooked in the system design and implementation.

Generally speaking, the first category of the benchmarks is selected from the second category according to more strict criteria. Fig. 2 explores how to define the representative workloads in the computer discipline. There is increasing freedom from a mathematical problem definition to an algorithm, an intermediate representation, An ISA-specific representation (ISA is short for instruction set architecture), and a micro-architecture representation. Section 3 will further discuss this challenge.

The third is a standardized data set that represents real-world data science problem [12], with defined properties, some of which have ground truth. ImageNet [13] (deep learning benchmark) and MIMIC-III [14] (critical care benchmark) are typical examples. The benchmark of this category is often used to measure against different algorithms. The state-of-the-art algorithm implementation plus the data set usually constitutes the benchmark of the second category.

The fourth is a representative data set, used as a reference. For example, a financial benchmark is an index (statistical measure), calculated from a representative set of underlying data, is used as a reference for financial instruments or contracts [15]. Well-known financial benchmarks include the London Interbank Offered Rate (Libor) and the Euro Interbank Offered Rate [15].

The fifth is the industry best practices in different domains. Benchmarking is the continuous process of searching the industry best practices that lead to superior performance and measuring products, services, and processes against them [3]. The Xerox Corporation pioneered and enhanced this benchmarking process.

3. The challenges

As I elaborate in Section 2, the five categories of benchmarks have a closely connected relationship. However, currently, there is no consistent benchmarking across multi-disciplines. Even no previous work tries to relate those five categories of benchmarks in multi-disciplines. The metrology science paves a foundation for this direction. However, they mainly focus on classical quantities like length, time, and power. Significantly different from those classical quantities, the properties of the objects in the computer, management, or finance disciplines are

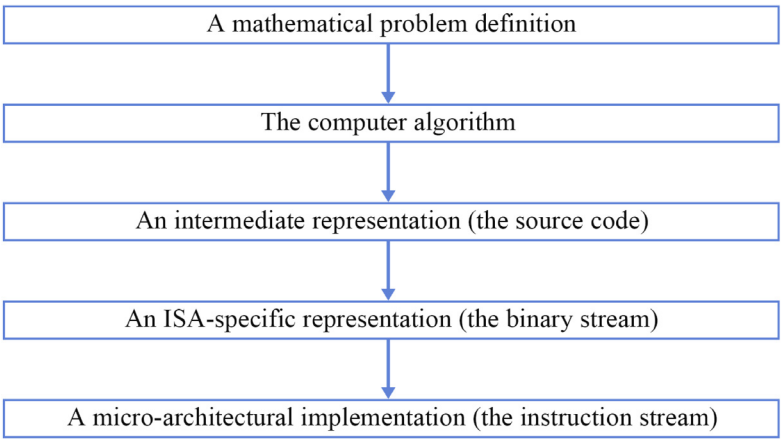


Fig. 2. In the computer discipline, a representative workload, the second category of the benchmarks, is hierarchically defined. From top to down is a mathematical problem definition, an algorithm, an intermediate representation, an ISA-specific representation, a micro-architectural representation. The lower level has more state space. State-of-the practice only analyzes a micro-architectural representation, which is only a subspace or even a point at a high-dimension space [16]. This hierarchy definition can be extended to other disciplines.

greatly affected by its mathematical problem definition and concrete implementation, which raises a serious challenge.

Different observation angles may distort the observable properties. For example, shown in Fig. 2, the quantity value of a computer workload is greatly affected by mathematical problem definitions, concrete algorithms, different ISA and micro-architecture implementations.

I further take the first category of benchmarks as an example to demonstrate the importance of tackling this challenge. Measuring "Quantum Supremacy" against the classical supercomputer is a fundamental issue. Google's "Quantum Supremacy" declaration in 2019 [17] stated that the Sycamore superconductive quantum computer (200 s) is over a billion times faster than the state-of-the-practice Summit system in 2016 [18] (10,000 years) in the task of measuring and simulating one million samples. However, in 2021, a group of scientists and engineers declared, on the Sunway Supercomputer [19], they reduced the classical simulation sampling time of Google Sycamore to 304 s, from the previously claimed 10,000 years through both algorithmic and architecture innovations.

The speed up – the ratio of the quantity values of two different kinds of systems – definitely will change wildly in the future. Understanding the benchmark very well under a hierarchy like that defined in Fig. 2 is a priority before correctly interpreting the implication of the speed up, or else it will mislead the scientific community. The situation may become much complex in the other disciplines, as a clear hierarchy definition is also a luxury. Establishing consistent benchmarking across multi-disciplines is very challenging.

The other challenge is how to measure the benchmarks themselves. Previous work has a preliminary discussion on this issue. For example, in the computer discipline, the characteristics of a (good) benchmark, i.e., representative [4,20], relevance, reproducible, fair, verifiable, repeatable, and economical are discussed in [21,22]. However, most of those properties are subjective. We need a meta-benchmark to evaluate those benchmarks.

I take the representative characteristic as an example; the current theory and practice cannot convince the community that this topic is seriously treated. From the perspective of mathematics, it is necessary to establish a mathematical foundation and consider the meaning of representative in a high dimension space. Unfortunately, in practice, the benchmarking methodology seems ad-hoc. For example, it is reported that there are 6.8 million apps in the leading app stores [23]. How does the community infer the mobile phone market's representative workloads (and benchmarks)?

4. The proposal

I believe that it is necessary to establish benchmark science and engineering; one of the goals is to set up standard benchmark hierarchy across multi-disciplines. There are two reasons. First, there is a natural hierarchy in different categories of benchmarks. As we discussed in Section 2, the first benchmark category is selected from the second category according to more strict criteria. Second, through this hierarchy, we can tackle the challenge of the rising cost of benchmarking. For example, we can put more resources on the primary benchmarks while relating the other benchmarks to the primary benchmarks through traceability.

Fig. 3 is my proposal. The most important is to keep benchmarking consistently, and the following measures will help achieve the target: (1) the unified definition of base quantity and units of measurement; (2) the realization of quantities and units of measurement with different accuracy (and hence cost) levels; (3) the traceability and calibration across the standard benchmark hierarchy. Traceability [8] is a property of a measurement result whereby the result can be related to a reference through a documented unbroken chain of calibrations, each contributing to the measurement uncertainty.

At the first tier, the international community needs to define the fundamental benchmarking principle and realize the base quantity, unit of measurement, primary measurement standard, which is the reference of all other benchmarks. The second tier is the first and second categories of the benchmarks. They will reuse the definitions and realizations of base quantity and unit of measurement from the first tier. Meanwhile, the definition and realization of derived quantity and unit of measurement are necessary.

The third tier is the second and fourth categories of the benchmarks. The community often needs to revisit and ponder the mathematical or data problem definitions to provide state-of-the-art and state-of-the-practice implementations. The fourth tier is the fifth category of the benchmarks. As it searches for the best practice, keeping an eye on the advancement of all hierarchies is necessary.

5. TBench: the venue for benchmark science and engineering

I think it is the right time to launch a new journal, BenchCouncil Transactions on Benchmarks, Standards, and Evaluations (in short, TBench). It will provide a venue to communicate and tackle the challenges mentioned above as there is no multidisciplinary and interdisciplinary journal on this area. I only noticed in the management discipline a closely related journal named Benchmarking: An International Journal.

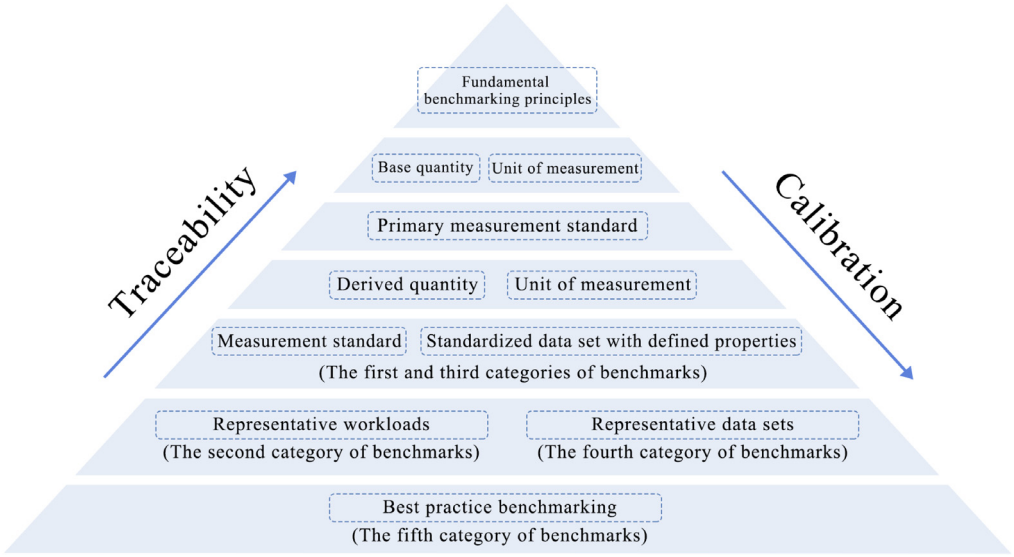


Fig. 3. The standard benchmark hierarchy proposal.

The vital importance of a new journal is to guarantee that high-quality submissions receive high-quality reviews promptly. According to the past experiences in the other reputable journals and conferences in the computer discipline, which is my primary background, I have some considerations.

In the computer discipline, a journal paper often cannot receive consistent and timely reviews compared with other top-tier conferences. For example, different associate editors invite reviewers from uncertain sources to handle papers with large deviations. Instead, a program committee meeting provides comparatively consistent reviews at a top-tier conference.

Another issue is the significant delay. Overall, the average turnaround of handling a paper is from three months to a year. Some journals reject most submissions at the disposal of a staff who does not understand its content to speed up the process and reduce the external review load. That will harm our community for two reasons. First, the value of peer review is to provide constructive feedback, which is the stone of our scientific community. Second, it will result in the abuse of editor rights. The last issue is most journals adopt a single-blind review, which prevents fair review.

To resolve the above issues, I enact the following plans. (1) Consistent and reliable reviews. In addition to about thirty founding editors or editors, similar to the program committee member of a conference, we will invite approximately 30 associate editors (Junior researchers with Ph.D. degrees). The associate editor is similar to the external review committee member of a conference. A team of founding editors, editors, and associate editors will provide the basis for consistent and reliable reviews.

(2) Fast-track peer review. The editor-in-chief (EIC) will read each paper's abstract and introduction. Suppose the team thinks this is a high-quality paper with high impact potential. In that case, they will invite three editors to have a timely review, including possible remote discussion and make a final decision within three weeks. The team will ask one editor and two associate editors to review the other papers. Overall, the team will finish one round of decisions within one month.

(3) A double-blind review process. One member of the EIC team without conflict of interest (COI) is responsible for checking COIs, while the other EIC and editor, who do not know the authors' identities, make a final decision. Each published article is reviewed by a minimum of three independent reviewers using a double-blind peer-review process. The identities of the reviewers are not known to the authors, and the reviewers also do not know the identities of the authors.

Acknowledgments

I am very grateful to many persons' contributions to TBench, especially Prof. Dr. Tony Hey for discussing the TBench plan, Dr. Lei Wang for discussing and proofreading this article, Mr. Shaopeng Dai for compiling the references, Mr. Qian He for drawing the figures, Mr. Zhengxin Yang for discussing the metrology related work, Ms. Chitra Krishnamoorthy, Ms. Divyaa Veluswamy, and other KeAI and Elsevier staffs for publishing TBench. Without all of you, launching TBench is impossible.

References

[1] A. Clare, Performance evaluation, in: The CFA Institute Investment Foundations, 2014, pp. 173–205.
[2] S.S. Stevens, et al., On the Theory of Scales of Measurement, Bobbs-Merrill, College Division, 1946.
[3] M. Zairi, P. Leonard, Origins of benchmarking and its meaning, in: Practical Benchmarking: The Complete Guide, Springer, 1996, pp. 22–27.
[4] B.C. Lewis, A.E. Crews, The evolution of benchmarking as a computer performance evaluation technique, MIS Q. (1985) 7–16.
[5] E.O. Joslin, Evaluation and performance of computers: application benchmarks: the key to meaningful computer evaluations, in: Proceedings of the 1965 20th National Conference, 1965, pp. 27–37.
[6] W. Buchholz, A synthetic job for measuring system performance, IBM Syst. J. 8 (4) (1969) 309–318.
[7] D.M. Conti, Findings of the Standard Benchmark Library Study Group, (500–538) Sept. of Commerce, National Bureau of Standards, Institute for Computer Sciences and Technology, 1978.
[8] I. BIPM, I. IFCC, I. IUPAC, O. ISO, The international vocabulary of metrology—basic and general concepts and associated terms (VIM), 3rd edn. JCGM 200: 2012, in: JCGM (Joint Committee for Guides in Metrology), 2012.
[9] R.N. Kacker, On quantity, value, unit, and other terms in the JCGM international vocabulary of metrology, Meas. Sci. Technol. 32 (12) (2021) 125015.
[10] F. Tang, W. Gao, J. Zhan, C. Lan, X. Wen, L. Wang, C. Luo, Z. Cao, X. Xiong, Z. Jiang, et al., Aibench training: balanced industry-standard AI training benchmarking, in: 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, 2021, pp. 24–35.
[11] Z. Jiang, W. Gao, F. Tang, L. Wang, X. Xiong, C. Luo, C. Lan, H. Li, J. Zhan, HPC AI500 V2. 0: The methodology, tools, and metrics for benchmarking HPC AI systems, in: 2021 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2021, pp. 47–58.
[12] MIT, Automl benchmark datasets, 2021, https://openml.github.io/automlbenchmark/benchmark_datasets.html Accessed December 2, 2021.
[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, L. Fei-Fei, Imagenet: A large-scale hierarchical image database, in: 2009 IEEE Conference on Computer Vision and Pattern Recognition, IEEE, 2009, pp. 248–255.

[14] A.E. Johnson, T.J. Pollard, L. Shen, H.L. Li-Wei, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L.A. Celi, R.G. Mark, MIMIC-III, a freely accessible critical care database, Sci. Data 3 (1) (2016) 1–9.
[15] IOSCO, Financial Benchmarks, Technical Report, 2013.
[16] L. Wang, X. Xiong, J. Zhan, W. Gao, X. Wen, G. Kang, F. Tang, Wpc: Whole-picture workload characterization across intermediate representation, isa, and microarchitecture, IEEE Comp. Archit. Lett. (2021).
[17] F. Arute, K. Arya, R. Babbush, D. Bacon, J.C. Bardin, R. Barends, R. Biswas, S. Boixo, F.G. Brandao, D.A. Buell, et al., Quantum supremacy using a programmable superconducting processor, Nature 574 (7779) (2019) 505–510.
[18] J. Wells, B. Bland, J. Nichols, J. Hack, F. Foertter, G. Hagen, T. Maier, M. Ashfaq, B. Messer, S. Parete-Koon, Announcing Supercomputer Summit, Technical Report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States), 2016.
[19] Y. Liu, X. Liu, F. Li, H. Fu, Y. Yang, J. Song, P. Zhao, Z. Wang, D. Peng, H. Chen, et al., Closing the "quantum supremacy" gap: achieving real-time simulation of a random quantum circuit using a new Sunway supercomputer, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2021, pp. 1–12.
[20] F. Pan, W. Wang, A.K. Tung, J. Yang, Finding representative set from massive data, in: Fifth IEEE International Conference on Data Mining (ICDM'05), IEEE, 2005, pp. 8–pp.
[21] J. v. Kistowski, J.A. Arnold, K. Huppler, K.-D. Lange, J.L. Henning, P. Cao, How to build a benchmark, in: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering, 2015, pp. 333–336.
[22] K. Huppler, The art of building a good benchmark, in: Technology Conference on Performance Evaluation and Benchmarking, Springer, 2009, pp. 18–30.
[23] Statista, Number of apps available in leading app stores, 2021, https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, accessed at Dec 2, 2021.

Dr. Jianfeng Zhan is a Full Professor at Institute of Computing Technology (ICT), Chinese Academy of Sciences (CAS), and University of Chinese Academy of Sciences (UCAS), and director of the Software Systems Labs, ICT, CAS. He received his B.E. in Civil Engineering and MSc in Solid Mechanics from Southwest Jiaotong University in 1996 and 1999, and his Ph.D. in Computer Science from Institute of Software, CAS, and UCAS in 2002. His research areas span from Chips, Systems to Benchmarks. A common thread is benchmarking, designing, implementing, and optimizing a diversity of systems. He has made substantial and effective efforts to transfer his academic research into advanced technology to impact general-purpose production systems. Several technical innovations and research results, including 35 patents, from his team, have been adopted in benchmarks, operating systems, and cluster and cloud system software with direct contributions to advancing the parallel and distributed systems in China or even in the world. He has supervised over ninety graduate students, post-doctors, and engineers in the past two decades. Dr. Jianfeng Zhan founds and chairs BenchCouncil and serves as the Co-EIC of TBench with Prof. Tony Hey. He has served as IEEE TPDS Associate Editor since 2018. He received the second-class Chinese National Technology Promotion Prize in 2006, the Distinguished Achievement Award of the Chinese Academy of Sciences in 2005, and the IISWC Best paper award in 2013, respectively.

BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100001

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Workflow Critical Path: A data-oriented critical path metric for Holistic HPC Workflows

Daniel D. Nguyen *, Karen L. Karavanic

Department of Computer Science, Portland State University, Portland, OR, United States of America

ARTICLE INFO

Keywords:
Critical path analysis
Holistic HPC Workflows
Parallel performance tools
Workflow critical path

ABSTRACT

Current trends in HPC, such as the push to exascale, convergence with Big Data, and growing complexity of HPC applications, have created gaps that traditional performance tools do not cover. One example is Holistic HPC Workflows — HPC workflows comprising multiple codes, paradigms, or platforms that are not developed using a workflow management system. To diagnose the performance of these applications, we define a new metric called Workflow Critical Path (WCP), a data-oriented metric for Holistic HPC Workflows. WCP constructs graphs that span across the workflow codes and platforms, using data states as vertices and data mutations as edges. Using cloud-based technologies, we implement a prototype called Crux, a distributed analysis tool for calculating and visualizing WCP. Our experiments with a workflow simulator on Amazon Web Services show Crux is scalable and capable of correctly calculating WCP for common Holistic HPC workflow patterns. We explore the use of WCP and discuss how Crux could be used in a production HPC environment.

1. Introduction

The term *workflow* is used throughout scientific computing with different contexts and meanings. For example, some scientific applications are developed with a workflow management system such as Pegasus [1] or Kepler [2], that schedules, runs, adapts, and summarizes a large number of lightweight tasks. Yet many computational science applications are implemented outside of any structured workflow management system. They comprise multiple steps, where each is a distinct library, script, or application with a specific functionality and design. For example, a science code might call an existing modeling code that is treated as a black box. These *Holistic HPC Workflows* are the focus of this work. Holistic HPC Workflows are an increasingly important paradigm with the potential for performance bottlenecks caused by movement and copying of large datasets, and inefficient interfaces between the separate components and applications. Today these workflows often include analysis and visualization of very large data sets, using methods developed for Big Data such as machine learning, analytics, and visualization. This growing complexity requires new ways of characterizing performance at the workflow level [3]. Workflow management systems (WMS) like Pegasus offer researchers a way to organize, execute, and analyze their scientific jobs. However, the performance analysis is tightly coupled to the WMS, thus these systems do not solve the problem of holistic performance analysis for workflows designed outside of such a system.

Analyzing the performance of Holistic HPC workflows presents a challenge for many existing performance tools, that are able to accurately and efficiently diagnose the performance of each individual component, but not to diagnose problems that span across them [4,5]. For instance, tools such as HPCToolkit [6] and TAU [7] use profiling and tracing techniques to detect performance bottlenecks in parallel applications. Some tools such as Darshan [8] and IPM [9] use I/O tracing to characterize I/O behavior of parallel applications. These tools were designed to analyze the performance of a single parallel code using the common approaches of message passing interface (MPI), multithreading (OpenMP), acceleration (CUDA), or a hybrid approach. However, diagnosing Holistic HPC Workflows requires integrated analysis across the separate components. A recent U.S. Department of Energy report on the future of scientific workflows called out this need for research "extending single-application performance validation tools to workflows of applications" [10].

One motivating example for our work is the Groningen Machine for Chemical Simulations (GROMACS) [11]. GROMACS is a scientific framework for simulating molecular dynamics of biochemical modules such as proteins, lipids, and nucleic acids. It models these molecular dynamics by solving Newtonian equations of motion for systems with hundreds to millions of particles. A common workflow pattern in GROMACS involves setting up a simulation environment, adding a solvent medium, generating an initial molecular model, calculating energy minimization, calculating initial equilibrium, and calculating actual molecular dynamics [12]. Each step can correspond to a single application using a shared file system, managed by a job scheduler like SLURM. Analyzing the workflow performance of GROMACS proved difficult; attempts included using a top-down approach by deconstructing

* Corresponding author.
E-mail addresses: ddn2@pdx.edu (D.D. Nguyen), karavan@pdx.edu (K.L. Karavanic).

https://doi.org/10.1016/j.tbench.2021.100001
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 23 December 2021

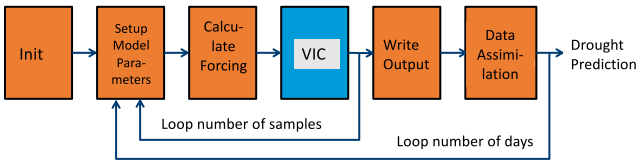


Fig. 1. Application layer of DroughtHPC workflow.

workflow into I/O, communication, and computation components and subsequently instrumenting the workflow applications to record these metrics [13].

A second motivating example is *DroughtHPC* [14]. This application, developed at Portland State University, predicts drought for a target geographical area. It utilizes the Variable Infiltration Capacity model (VIC) [15] to simulate meteorological samples over a given time period. A python script is used to perform data assimilation and call VIC in a loop (see Fig. 1). Every call to VIC inputs and outputs 25 files. The number of calls equals the number of samples needed multiplied by the number of days needed. Locating workflow bottlenecks for DroughtHPC, particularly due to dataflow and the control flow of the entire workflow, was challenging [16]. To investigate performance bottlenecks, researchers manually ran a variety of measurement tools to focus attention to the key bottlenecks. The overhead of calls to the VIC hydrologic model from within a python loop and significant file creation, reads, and writes, represented main performance bottlenecks. The DroughtHPC study shows a need for one performance tool that can detect common dataflow patterns and diagnose runtime bottlenecks across different phases in a scientific workflow.

Holistic HPC Workflow Diagnosis is also an important aspect of the procurement process for major new systems at large science labs. Describing the workload accurately is essential to matching the capabilities of the future systems to the needs of the lab. Fig. 2 shows an example of the phases associated with common large-scale scientific simulation workflows, with data retention timescales divided into temporary, campaign, and forever. The temporary timescale describes application data that is typically discarded at completion of a phase or run. The campaign timescale includes data used throughout the execution or set of executions of the entire scientific workflow. The archive timescale describes data stored for longer archival purposes. This type of diagram is modeled after those developed by The Alliance for Application Performance at Extreme Scale (APEX) [17].

In this paper we present our initial work to address this need. *Workflow Critical Path* (WCP) is a data-oriented critical path metric for Holistic HPC Workflows. Building on earlier work on Critical Path Analysis for individual MPI applications [18–20] we have developed a technique for determining the critical path across an entire Holistic HPC Workflow. This has the potential to help researchers better understand data movement patterns and potential bottlenecks occurring across the complex memory hierarchy and storage systems in a large-scale HPC cluster. Our approach is designed to focus developers' optimization efforts, avoiding the need to separately analyze each participating application and manually determine where to focus. It also allows the detection of performance bottlenecks related to moving from one stage of the workflow to the next, for example, copying and transforming simulation output data for analysis with a visualization tool.

The key contributions of this paper are:

1. **We define Workflow Critical Path (WCP), a novel performance metric for Holistic HPC Workflows.** WCP describes the critical path for an entire HPC workflow by defining a program activity graph (PAG) where vertices represent *data state* and edges represent *data mutations*.
2. **We present Crux,** a distributed, runtime tool that calculates WCP. Crux follows a service-oriented architecture and deploys on a target number of nodes in an HPC cluster. Crux provides an API for building workflow PAGs and computing WCP. It also provides a user interface (UI) for visualizing WCP data. Our Crux prototype can be deployed in the Cloud using Amazon Web Services or locally using Docker.
3. **We developed a configurable HPC workflow simulator framework, and used it for a detailed capability, scaling and performance study.** The configurable workflow simulators allow users to simulate representative workloads.

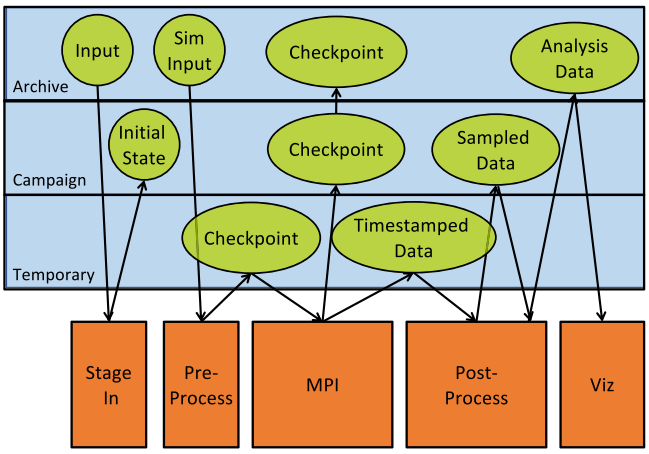


Fig. 2. APEX style workflow diagram. The brown boxes along the bottom show the major steps in the workflow, and the blue rectangles show the levels of memory and storage for the data at each step.

This represents a first step towards *Holistic HPC Workflow* performance diagnosis [21].

2. Related work

There are a number of workflow management systems in use today, for example Kepler and Pegasus. Such systems generally include performance monitoring infrastructure, however the applications must be specifically implemented for the specific workflow management system. Crux on the other hand targets Holistic HPC Workflows, that comprise separately developed components to solve a single problem.

Annotation-based (also referred to as application-instrumented) distributed monitoring schemes developed for commercial server environments rely on applications to explicitly tag every record with a global identifier that links these message records back to the originating request. These systems tend to be very accurate but potentially slow, as all system components must be instrumented. One example is Dapper, developed by Google [22]. It has been used for a large, production distributed systems tracing framework. In a Dapper trace tree, the tree nodes are basic units of work which are referred to as spans. The edges indicate a causal relationship between a span and its parent span. A span is a simple log of timestamped records which encode the span's start and end time, any RPC timing data, and zero or more application-specific annotations. A span can contain information from multiple hosts and in fact every RPC span contains annotations from both the client and server processes. Crux follows an annotation-based approach, however it greatly reduces the overhead by only creating nodes for data operations.

Facebook's end-to-end performance tracing infrastructure, Canopy, is another example of a large-scale, runtime performance tool that can record and process over 1 billion traces per day [23]. Canopy has several features similar to WCP. Canopy models trace data as DAGs with nodes representing events in time, with events defined more broadly and at a lower level than WCP. Canopy authors noted how infeasible

it was to expose traces at that particular level of granularity since end-users, i.e. Facebook engineering teams, would not understand the mappings to higher-level concepts. To address this, Canopy constructs a modeled trace of events, which are higher-level representations of lower-level performance data. WCP focuses on the end-to-end movement and transformation of data across an entire HPC workflow instead of performance within any particular workflow component. For example, WCP is not intended to diagnose one MPI application. Like WCP, Canopy also derives the critical path of its trace data and visualizes the critical path to the end user.

Research into CPA for parallel programs started in the 1980s with work such as Yang & Miller [20]. Their approach involved constructing a directed, weighted graph, called program activity graph (PAG), whose vertices represent events (e.g. send/receive and process creation/termination events) in a program and whose edges represent the duration of the event. They were able to return the longest path on a scale of tens of thousands of nodes. Critical path analysis evolved in the 1990s with techniques such as using piggybacking critical path data on MPI messages to compute the critical path profile during runtime [19] Hollingsworth demonstrated that using this technique, most programs can tolerate a 5%–10% level instrumentation overhead without suffering significant change of the critical path length. Critical path for individual MPI applications has continued to be improved and scaled up with increasing numbers of MPI ranks [24–27]. Overall, critical path analysis is useful in identifying the cause of a program's total execution time, diagnosing bottlenecks to application scalability, and predicting overall performance [24].

Our work targets holistic HPC workflows, thus requiring a novel approach to monitor separate components and merge their graphs. In preliminary work towards this same goal, Herold & Williams introduced a top-down performance analysis approach to monitor workflow applications [18]. They implemented a tracing infrastructure that interfaces with the resource manager to provide summarized performance metrics for workflow, jobs, and job steps. In contrast, we focus on defining a specific metric, WCP, and a runtime approach to its calculation and visualization.

3. Workflow critical path (WCP)

Workflow Critical Path is calculated by constructing a program activity graph (PAG) spanning all components of a holistic workflow, representing *data state* as vertices and *data mutations* as edges. The result is a PAG that can be analyzed for *data state patterns* through an entire HPC workflow (Fig. 3).

A *data state* comprises:

- Size — the size of the data, for example 10 MB;
- Time — the creation timestamp;
- Origin — the original application that produced the state;
- Location is the current storage location, for example "node1 disk1" ; and
- Label - a meaningful descriptor for the data state (e.g. file.csv, byte_stream).

An edge represents a *data mutation*, an operation that changes a data state. An edge comprises:

- cost — the elapsed time between two connected data states; and
- mutation — the operation performed on a data state resulting in a state change.

The resulting graph allows WCP to describe a data set evolving over time. This focus on data generalizes time: unlike profilers, the "cost" captured in each edge includes all computation and I/O activity between each two data states. Reducing I/O activity cost thus potentially changes or improves the critical path. Reducing computation time also potentially changes or improves the critical path, just as in computation-oriented approaches.

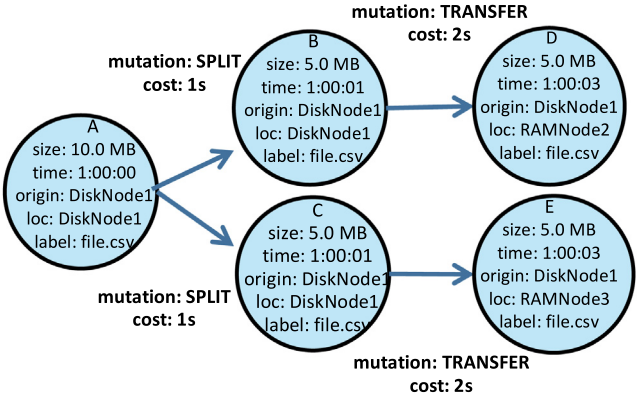


Fig. 3. Trivial example of a data state undergoing different mutations. Vertex A represents a 10.0MB file called file.csv on a disk belonging to Node1. Vertex A undergoes a SPLIT mutation that divides file.csv into file1.csv and file2.csv. The result is two new data states, B and C. Vertices B and C undergo a TRANSFER mutation that transfers file1.csv and file2.csv from disk on Node 1 to memory on nodes 2 and 3 respectively.

3.1. Critical path algorithm

The critical path represents the *longest* path through the graph of data state mutations based on execution time. Thus, critical path algorithms are typically shortest path algorithms modified to find the path with the longest execution time [28]. A well-known algorithm that solves the single source shortest path (SSSP) problem is Dijkstra's algorithm which has a worst-case performance of O(|E|+|V| log |V|) where |V| is the number of vertices and |E| is the number of edges. Delta-stepping [29] is a distributed variant that divides Dijkstra's algorithm into phases that can be executed in parallel on distributed memory architectures for an average-case time of $O(\log^3 n/\log\log n)$. For our WCP prototype we use a version of Delta-stepping implemented for shared memory architectures described by Kranjčević et al. [30]. The input of the Δ-stepping algorithm is a graph given by its vertices V, edges E, and the cost function c, a source node s, and an optional parameter $\Delta > 0$ used to divide all the outgoing edges of each vertex into two categories, called light and heavy edges, based on whether the cost of that edge is smaller or larger than Δ. The Kranjčević et al. implementation of Delta-stepping performs $O(|V|^{1+\frac{1}{d}})$ operations total for graphs representing d-dimensional square lattices. Their testing showed an average parallel efficiency of at least 50% over Dijkstra. The pseudocode for this algorithm is shown in Fig. 4.

Traditional PAGs where nodes represent computations and edges represent computational activities typically store the duration of computational activities as the edge weight and employ a longest path algorithm to return the critical path [28]. Since WCP represents data state as vertices and data mutations as edges, we use the elapsed time between data states as the edge weight.

We store a weight property, called cost, for edge E such that the cost equals the inverse of elapsed time, i.e. difference between timestamped values of vertex A and vertex B.

$$cost_E = \frac{1}{time_B - time_A}$$

The inverse elapsed time means that edges between data state vertices with large time differences will receive a small cost value and vertices with small time differences will receive a large cost value.

4. The crux prototype

To enable further study of WCP, we implemented a prototype, Crux, along with the tooling needed to build and deploy. Crux comprises the following modules:

```
function Δ-Stepping(V,E,c,s,Δ):
    for each vertex v in V:
        heavy[v] ← {(v,w) ∈ E : c(v,w) > Δ}
        light[v] ← {(v,w) ∈ E : c(v,w) <= Δ}
        tent[v] ← ∞
    end for
    relax(s,0)
    i ← 0

    while B ≠ ∅:
        S ← ∅
        while B[i] ≠ ∅:
            Req ← {(w,tent(v)+c(v,w)) : v ∈ B[i] and (v,w) ∈light[v]}
            S ← S ∪ B[i]
            B[i] ← ∅
            for each (w,d) ∈ Req: relax(w,d)
        end while
        Req ← {(w,tent(v)+c(v,w)) : v ∈ S and (v,w) ∈ heavy[v]}
        for each (w,d) ∈ Req: relax(w,d)
        i ← i+1
    end while
    return tent[]
end function

function relax(w,d):
    if d<tent[w]
        tent[w] ← d
        B[⌊tent[w]/Δ⌋] ← B[⌊tent[w]/Δ⌋] \ {w}
        B[⌊d/Δ⌋] ← B[⌊d/Δ⌋] ∪ {w}
    end if
end function
```

Fig. 4. Pseudocode for the Delta-Stepping Algorithm.

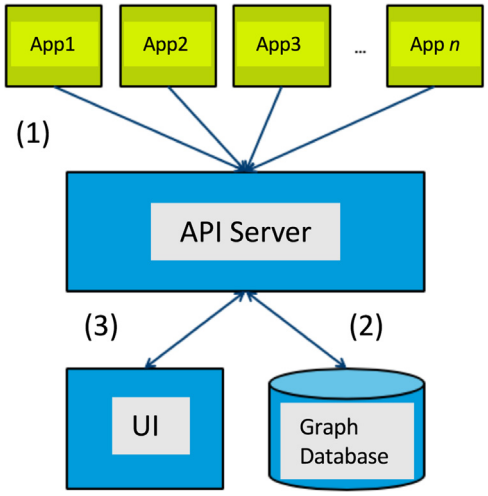


Fig. 5. Crux API interaction with HPC clients. (1) HPC applications (green) make API calls to Crux (blue) through HTTP requests to the API server. (2) The API server communicates to the graph database using a compatible protocol. (3) The UI is a standalone application making HTTP requests to the API server for CRUD (create, read, update, and delete) actions on the graph database.

Crux API: An HTTP, application programming interface (API) that exposes representational state transfer (REST) endpoints to workflow HPC applications. The Crux API server implements routines to build workflow PAGs; interfaces with the Crux Database; performs data integrity checks; and manages Crux's performance metadata;

Crux Database: A database that stores a workflow PAG and executes Crux's critical path algorithm for finding the WCP;

Crux UI: A user interface (UI) to visualize workflow PAGs and WCP.

In the remainder of this section we describe each of these modules, Crux deployment, and examples of Crux.

4.1. Crux API

The Crux API is an HTTP API that follows a representational state transfer (REST) architecture, chosen for benefits such as scalability and portability. The Crux API must follow several constraints. First, it must define stateful objects as API resources for clients to access. The API resources should map to Crux's data state schema. For example, a client should be allowed to query a specific data state vertex in the database by sending an HTTP GET request to an API. Second, the Crux API must be a manager of the Crux database. It must implement logic that tells the database how to perform simple CRUD actions such as creating a vertex or updating an edge, or more complicated actions like submitting queries needed to calculate the critical path from two data state vertices in the PAG. Third, it must enforce the Crux data state schema so that clients cannot send malformed requests. Fourth, the API must provide support to the Crux UI for any backend requests and must provide common application features such as user login and access token management (See Fig. 5 and Table 1).

We identified the following properties as most important when comparing different backend tools and languages for the Crux API prototype: rapid development, high performance and asynchrony. Thus we chose to implement the Crux API using Python's Asynchronous Server Gateway Interface (ASGI), a core library used by a popular Python backend framework, Django. We chose FastAPI [31], a framework built around Starlette, which is a lightweight ASGI framework. FastAPI is a fast Python framework that integrates with standards for OpenAPI and the JSON schema.

REST-based web services are typically organized into resources, logical objects we want to expose to the user. The Crux API includes two resources: states and mutations. A resource is identified by a Uniform Resource Identifier (URI). Clients can access that resource by sending an HTTP request method to that URI. RESTful web APIs typically deploy URIs following the pattern `scheme://host:port/version/resource`. Parameters can also be used in URIs. For example, an API might have a path parameter `/users/{ID}` which lets the client specify a certain user with a specific ID. Parameters can also exist in the form of query parameters which lets a client sort or filter on a particular resource. For example, `/states?location=disk01` returns all data states located on disk01. For Crux, we use a combination of path and query parameters for clients to access resources.

Our Crux prototype includes six types of data mutations, based on common data operations observed in scientific applications:

TRANSFER — Transfer of data between one physical location to another (e.g. staging in data from storage to compute node)

CONVERT — Conversion of data format or schema (e.g. JSON to CSV)

APPEND — Appending data to existing data (e.g. adding timestamps to data points in a file)

SPLIT — Splitting of data into multiple locations (e.g. mpi_scatter())

MERGE — Merging data from different sources (e.g. mpi_gather())

DELETE — Permanent deletion of data

4.2. Crux database

The Crux Database is the backend storage for the Crux API. The database must support concurrent control to manage write operations from multiple API instances, and scaling to accommodate collected PAG data, representing vertices, i.e. data state, and edges, i.e. data mutations, of an entire HPC workflow. For an example workflow of 5 applications, each generating 100 data states and performing 100 data mutations, Crux's database must hold 50,000 entries.

For the Crux prototype, we wanted a solution that was well documented, showed strong concurrency use cases, and came with graph

Table 1
The Crux API.

HTTP Method	Path State Info	Description
GET	/states	Returns a list of data states
GET	/states/{ID}	Returns a data state with matching ID
POST	/states	Creates a new data state
GET	/mutations	Returns a list of data mutations
POST	/mutations/transfer start state, end state	Creates a TRANSFER data mutation between a start data state vertex and an end data state vertex.
POST	/mutations/convert start state, end state	Creates a CONVERT data mutation between a start data state vertex and an end data state vertex.
POST	/mutations/split start state, end states	Creates a SPLIT data mutation between a start data state vertex and ending at all end data state vertices
POST	/mutations/merge start states, end states	Creates a MERGE data mutation between all start data state vertices and ending at an end data state vertex.
POST	/mutations/append start state, end state	Creates an APPEND data mutation between a start data state vertex and an end data state vertex
POST	/mutations/delete start state, end state	Creates a DELETE data mutation between a start data state vertex and an end data state vertex
POST	/wcp start state: id, end state:id	Returns a list of data state vertices representing the workflow critical path between a start data state vertex and an end data state vertex

algorithm support optimized for that database. To this end, we chose a graph database, Neo4j [32], with a large ecosystem of tools and support. Neo4j uses a query language called Cypher and uses a convention of referring to vertices as *nodes* and edges as *relationships*. Cypher can be used to describe patterns of nodes and relationships and filter those patterns based on labels and properties. For example, the following Cypher query returns all data state nodes with matching property values:

```
MATCH (n)
WHERE n.size = {size} and
            n.location = {location} and
            n.time = {time} and
            n.origin = {origin}
RETURN n
```

To integrate Neo4j into Crux, we use a containerized version. We developed a custom Python library to express Crux data state, and data mutation schemas as proper Cypher queries to create nodes and relationships. We use a Python Neo4j client to execute write transactions between Crux API server and Neo4j. We calculate WCP using Neo4j's `algo.shortestPath.deltaStepping()` routine which implements delta-stepping for shared memory architectures described by Kranjčević et al. The Cypher query:

```
MATCH (start)
WHERE id(start) = {start_id}
CALL algo.shortestPath.deltaStepping.stream
    (start, "cost", 3.0)
YIELD nodeId, distance
RETURN algo.getNodeById(nodeId)
    AS destination, distance
ORDER BY distance
```

4.3. Crux UI

The Crux UI is a user interface to visualize critical path data in the Crux Database. This includes visualizing program activity graphs, critical paths, and various metadata like workflow runtime. In addition, the Crux UI provides features such as user authentication and profiles. The Crux UI runs as a standalone application and communicates to the Crux database via the API server. For users to access the Crux UI, the UI application must be properly exposed so authenticated end users can reach it from their location. For example, if end users are outside of the HPC cluster environment, the Crux UI can sit behind a public load balancer which routes public traffic to the UI instance.

We implemented the Crux UI for our prototype with the Neo4j Browser. The Neo4j Browser is a general-purpose UI that lets users query, visualize, administrate and monitor a Neo4j database. With this simpler approach, users can view a workflow PAG being constructed during runtime and submit Cypher queries against the graph database.

To visualize WCP in the Neo4j browser, we use the `algo.shortestPath.stream()` routine in the following Cypher query:

```
MATCH (start), (end)
WHERE id(start) = {start_id} and
    id{end} = {end_id}
CALL algo.shortestPath.stream (start, end,
    "cost")
YIELD nodeId, cost
RETURN algo.asNode(nodeId), cost
```

We needed to make one adjustment to the default behavior to ensure correctness for MERGE data mutations. A MERGE data mutation signifies the combining of two or more data states, such as combining of data from files to create a new file. When this occurs in Crux, a new data state vertex gets created and edges from each of the pre-merge data state vertices get added. At this point, each edge receives an elapsed time calculated from the parent vertex's timestamp and the timestamp of the new data state vertex. However, the critical path should be the path that includes the pre-merged date state vertex or vertices with the *smallest* elapsed time to the new data state vertex For Neo4j's shortest path algorithm to correctly return this path, we assign a large integer value as the cost for the other non-critical paths (see Fig. 6).

We were able to accomplish most needed functionality for Crux with Neo4j, however, the browser falls short of our particular needs. A production version of Crux would require a different approach for the Crux UI.

Fig. 9 shows a diagram of Crux installed in an HPC cluster. A basic installation of Crux requires the following:

- Minimum of 3 nodes located in the HPC cluster. These allocated nodes shall be on the same network as other compute nodes and accessible via HTTP.
- Crux UI installed on 1 node behind a load balancer or reverse proxy. This allows end users outside the HPC cluster network to reach Crux. The UI shall target HTTP requests to the Crux API server via another load balancer.
- At least one instance of the Crux API server installed on at least 1 node. Depending on workflow size, it may be appropriate to install multiple instances over multiple nodes. We expect private load balancer(s) to distribute API calls from client HPC applications efficiently to an API instance.

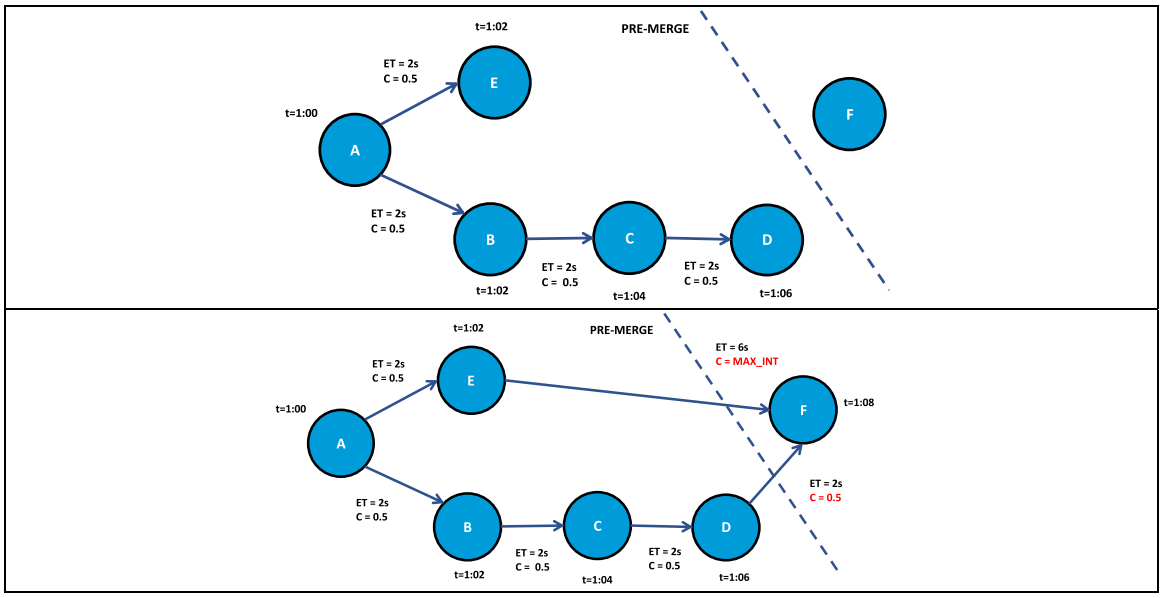


Fig. 6. Handling MERGE data mutations to use with Neo4j's shortest path algorithm. Top image shows a PAG with timestamped values for data state vertices A, B, C, D, and E along with data mutation edges with elapsed time (ET) and cost (inverse ET) shown. The intention is to merge vertex D and E. Vertex D took 6s to create from A whereas E took only 2s to create. Vertex F represents the new data state vertex from merging E and D. Bottom image shows Vertex F created at t=1:08 resulting in an elapsed time of 2s between D and F and 6s between E and F. Since the critical path must be ABCDF, we assign edge EF a high integer value in order for Neo4j's shortest path algorithm to return ABCDF as the critical path between A and F.

- Crux database installed on 1 node.

The Crux API is designed as a stateless server. It does not track or store data from clients. Clients of Crux must make appropriate API calls to Crux. This means that client HPC applications must support the same protocol (e.g. HTTP) to communicate with Crux. Furthermore, clients must know how to create data state information defined by Crux's schema. The following pseudocode shows an application loads in a data file `input.txt`. In order to model this in Crux, a total of 3 Crux API calls are needed.

4.4. Example

To illustrate WCP and Crux in practice, we instrumented two applications written in Python and C that perform similar I/O operations. Both programs stage in data, perform computation on the data, and write the results out to a new file. We inserted a total of 6 Crux API calls in each program. In the Python application, we included our custom Python library called Crux to access utility functions that help create and manage Crux data states. In C we also include utility functions that wrap around the `libcurl` library to help execute HTTP requests (see Figs. 7–9).

```
// loadFile() implies a TRANSFER data mutation where data is read into memory We
// use one API call to create the new data state vertex, another call to fetch the
// previous data state vertex, and a final call to create the data mutation edge
// Total Crux API calls: 3

data_file = loadFile("input01.txt")

// API call to create new data state vertex
httpPost(cruxServerURL + '/states', { label: 'stagein', size: sizeof(data_file),
                                      time: getTime(), location: 'memory', origin: 'myapp'
                                      })


// API call to fetch previous state when input01.txt was created
prev_state = httpGet(cruxServerURL + '/states?location=input01.txt')

// API call to create TRANSFER data mutation
httpPost(cruxServerURL + '/mutations/TRANSFER', {
    start_state: prev_state,
    end_state: { label: 'stagein', size: sizeof(data_file),
                 time: getTime(), location: 'memory', origin: 'myapp'
                 })
```

Fig. 7. Crux API calls for capturing a load from input.txt.

5. Crux workflow simulator

In order to effectively test WCP, we developed a workflow simulator for Crux, with a system of distributed applications to simulate representative scientific workflows. The simulator system serves as a lightweight, local testbed to examine Crux's performance (see Fig. 10).

We designed five representative workflows, each of which exhibits a characteristic element or pattern we have observed in HPC workflows, motivated in particular by the APEX report and DroughtHPC. The 5 workflows are:

1. Generic. These jobs include staging in data, preprocessing data, MPI, postprocessing data, and visualizing data. We consider this the simplest of workflows in that there is only one data source and the critical path will depend on the MPI rank that takes the longest. Simulators used: 'Stagein', 'Preprocess', 'MPI', 'Postprocess', and 'Viz'. TOTAL_MPI_RANKS = 4 (see Fig. 11).
2. *Data Splits*. MPI-based workflow with data splitting across a number of physical nodes. These jobs include staging in data, preprocessing data, MPI, postprocessing data, and visualizing data. Simulators used: 'Stagein', 'Preprocess', 'MPI', 'Postprocess', and 'Viz'.
3. *Checkpoint*. A workflow that includes more than one run of a parallel codebase with a checkpoint file created in between runs. The time duration to transfer the file is configurable. We simulate checkpoint files being written to storage between runs of parallel tasks representing the scientific simulation. Simulators: 'Stagein', 'Preprocess', 'MPI', 'Checkpointout', 'Checkpointin', 'MPI2', 'Postprocess2', 'Viz'.
4. *Multiple Sources*. Simulates a workflow that involves loading more than one source of data. The loading occurs between workflow jobs. Simulators: 'Stagein', 'Preprocess', 'MPI', 'Postprocess', 'Load', 'MPI2', 'Postprocess', 'Viz'.
5. *Create Delete*. Simulates a workflow that involves creating temporary files and deleting them between runs of a scientific simulation. Simulators: 'Stagein', 'Preprocess', 'MPI', 'Filecreate', 'Postprocess', 'MPI2', 'Postprocess', 'Viz'.

```
int main(int argc, char *argv[])
{
   char *startVertex, *inputVertex, *endVertex,
        *dataVertex;
   char *relationshipData;
   char *url, *data;

   // Init Crux and curl library
   curl_global_init(CURL_GLOBAL_ALL);
   startVertex = connectCrux();

   data = readFile("input.txt");

   // API calls to create new data state for
   // loading an input file into memory and to
   // create new data mutation TRANSFER to
   // represent loading a file
   inputVertex = newDataState("input.txt", 0);
   post(newURL (HOST, "/nodes/myapp"), inputVertex);
   post(newURL(HOST, "/relationships/transfer"),
   newRelationship(startVertex, inputVertex));

   sampleComputation(data);

   // API call to create new data state shows
   // data converted after computation in memory
   // API call to create new data mutation CONVERT
   dataVertex = newDataState("", strlen(data));
   post(newURL(HOST, "/nodes/myapp"), dataVertex);
   post(newURL(HOST, "relationship/convert"),
        newRelationship(inputVertex, dataVertex));

   writeFile("output.txt", data);

   // API call to create new data state for writing
   // to the new file, output.txt
   // API call to create new data mutation TRANSFER
   // to represent creating a file
   endVertex = newDataState("output.txt", 0);
   post(newURL(HOST, "/nodes/myapp"), endVertex);
   post(newURL(HOST, "relationships/transfer"),
   newRelationship(dataVertex, endVertex));
   curl_global_cleanup();
   return 0;
}
```

Fig. 8. Example of a simple C code with the inserted Crux API calls shown in boldface.

To make the simulators extensible, we create a common configuration file from which each simulator loads. This file parameterizes values such as maximum wait time between jobs or input dataset size. Each simulator starts an HTTP server and implements run_simulation()which takes a list of previous data states and returns a list of new states once all simulated jobs have completed. The pseudocode below highlights the basic logic in each simulator (we use the terms "nodes" and "relationships" to refer to vertices and edges respectively in order to follow Neo4j's naming convention).

```
// Input:   Previous data state vertex, prev_state
//             URL string to Crux API, url
// Output:     New data state vertex, new_state
run_simulator(prev_state, url)

// Simulate new preprocess data state
new_state = simulateDataState('preprocess')

// Call Crux API to create new data state  vertex
postRequest(url + '/nodes/preprocess', new_state)

// Call Crux API to create new data mutation edge,
// APPEND,
// between new data state and prev data state
postRequest(url + '/relationships/append',
    prev_state, new_state)
return new_state
```

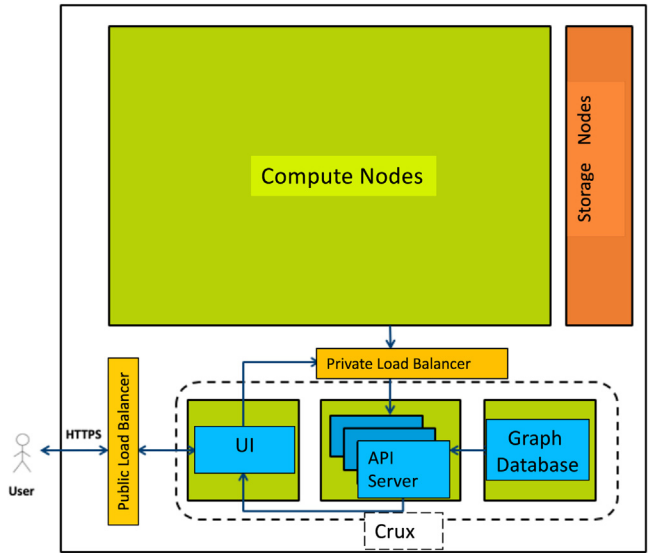


Fig. 9. Deployment of Crux in an HPC cluster. Crux components (blue) are deployed on 3 dedicated nodes. Two load balancers (yellow) are used to route traffic: a public load balancer which securely handles HTTPS requests from a user outside the HPC cluster network, and a private load balancer which routes traffic from compute nodes to Crux API server instance (in this case 3 running instances). The public load balancer can also be a reverse proxy. Storage nodes (orange) are displayed for reference.

In order to orchestrate simulator applications at runtime, we design a controller application called simulator manager. Simulator manager knows when to schedule each simulator's main routine. It also facilitates the passing of data between applications and performs health checks on each before starting. Simulator applications therefore only need to communicate with the simulator manager and not each other. The simulator manager's main routine receives an ordered list of URLs to each simulator. It initiates a null data state and enters a loop to call the first simulator with the null data state. The return value is a new data state which gets assigned to the previous state variable. The loop is then continued with the second simulator being called and so on. The pseudocode below outlines the basic algorithm.

```
// Input: ordered list of urls to each simulator, simulator_urls
// Output: void
run(simulator_urls)
    prev_state = null
    for url in simulator_urls
        prev_state = startSimulator(url, prev_state)
    return void
```

We use Docker to package Crux components as container images. Containers are isolated environments by OS-level virtualization. A container shares a host's kernel with other containers, but each container will only the see contents assigned to it. Docker is a set of tools for building and deploying containers. We choose to implement Crux with Docker for a variety of reasons. First, using containers for development offers benefits such as isolation, reproducibility, portability, and version control. Second, container images are lightweight compared to most virtual machine images. This is important when deploying Crux with workflow simulators since all Crux components and simulators run as individual containers (the largest being the API container at ~900MB and the smallest being a simulator container at ~115MB). Third, containers make it easy to deploy to cloud environments, which we leverage for testing purposes. (See Fig. 10).

Workflow Simulator components are implemented as standalone applications:

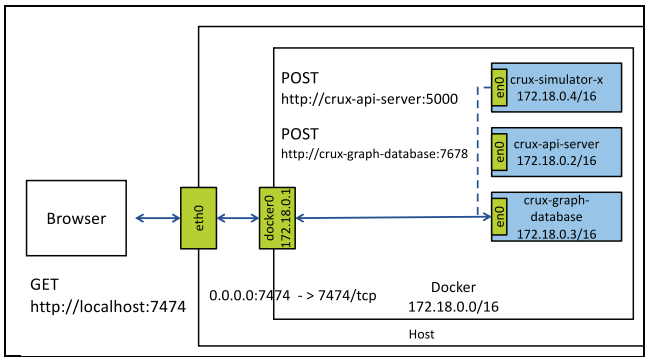


Fig. 10. Example of local deployment of Crux with simulators using Docker. Crux components and workflow simulators (blue) running locally as Docker containers. Network interfaces (green) shown to illustrate how containers run on separate virtual network than the host.

- The Simulator Manager is a controller and communicates with all simulator applications via HTTP. It knows when to launch a certain simulator app and send data between apps when needed.
- Simulator apps are configured to represent common HPC workflow jobs such as pre-processing data, running an MPI job, or performing post analysis. Each simulator app makes API calls to the Crux API server.
- The Crux API server, implemented in Python, sends Cypher queries via the Bolt protocol at bolt://graphdatabase.crux:7687.
- The web browser opens the Neo4j Browser at http://localhost:7474/browser/.

6. Evaluation

We conducted a series of experiments to demonstrate Crux capabilities and performance. For these tests we used Docker 19.03.5, Docker-compose version 1.24.1, Terraform v0.12.5, Python 3.7.6, Neo4j 3.5, Neo4j Graph 3.5.4.0, and AWS ECS Agent 1.32.0. The AWS EC2 Image used was Amazon Linux AMI 2018.03.y x86_64 ECS HVM GP2 t2.medium (2 vCPU, 4 GB RAM), and the local system was macOS Mojave (10.14.6) running on a 2.9 GHz Intel Core i9 (12 core) with 32 GB 2400 MHz DDR4.

6.1. Capability and WCP correctness

To explore WCP correctness, we configured the Crux workflow simulator to generate and execute the five characteristic workflows detailed in Section 6. The workflow simulator is deployed locally and makes API calls to a Crux instance on AWS. We kept runtimes short with modest total vertices in order to better provide screenshots of the entire PAG and workflow critical path.

For each of the 5 workflows, we configured a Crux simulator, then deployed it locally using a tool called Docker-Compose which launches all simulator components as containers. The simulator containers make API calls to a remote deployment of Crux on AWS. The workflow simulation is complete when we receive a JSON string from the Crux API which contains the workflow critical path. At this point, we collected screenshots of the visualized workflow PAG and workflow critical path via Crux's Neo4j Browser. Results are shown in Table 2 and Fig. 11. WCP was correctly computed in all cases. While the five cases do not cover every possible workflow pattern, we believe they cover key workflow patterns and thus are representative. The next step will be to explore WCP with full applications.

Table 2
Results of 5 simulator studies.

Workflow	*WCP Correct?*	*Cost*	*Elapsed time* (s)
Generic	Yes	9.492	6.584
Data Splits	Yes	22.159	57.590
Checkpoint	Yes	74.854	7.616
Multiple Sources	Yes	67.242	6.5413
Create Delete	Yes	74.868	7.2561

6.2. Performance and scalability

To characterize the scalability of the Crux prototype we measured the time required to create data state vertices on local and remote (AWS based) deployments of Crux (Fig. 12). Next, we scaled out the instances of Crux's API server to 1, 2, and 3 instances, and performed the same time measurement (Fig. 13).

6.3. Crux overhead

The main overhead of Crux will scale with the number of instrumented API calls required to create a full program activity graph. To approximate the number of Crux API calls needed for a smaller scale HPC application we used the DroughtHPC example. We estimated the number of Crux data states that would have to be created at 300k for VIC and 12k for the python code, for a total of 312k Crux API calls. A full-scale deployment will be required to accurately assess the overhead.

Dedicating nodes to deploy Crux in an HPC cluster implies taking nodes away that could otherwise be used as compute resources. However, we demonstrate Crux's ability to run as containers on modest EC2 instances. A small HPC cluster could potentially dedicate one node for running Crux on virtual machines or containers instead of directly on bare metal systems.

6.4. Discussion

Our performance experiments suggest that network proximity of Crux to application clients improves Crux's performance. This is consistent with our original expectations. Surprisingly, we observed that scaling out Crux API instances did not improve overall performance of Crux on AWS. This suggests that the limiting factor to Crux's performance may be the load balancer responsible for distributing traffic to the API instances. Another limiting factor could be the performance of Crux's database. Having multiple API instances would have diminishing returns if Crux's Neo4j instance is unable to process more requests.

Through our tests we observed limitations with the capability of the Neo4j Browser for Crux: inability to highlight a path within a graph; incorrect inclusion of one or more edges; and poor scaling to more than 2,000 vertices. Also, Neo4j Browser did not offer ways for us to visualize results from previous workflow runs or easily export graph data. Although sufficient for our initial prototype, a more comprehensive UI would be warranted for a production tool.

7. Conclusions and future work

In this paper, we introduced a novel metric, Workflow Critical Path (WCP) for Holistic HPC Workflows. We described a prototype tool called Crux for calculating WCP. To evaluate Crux we developed a set of simulators to simulate HPC workflows and workflow patterns; and designed a cloud-based, test environment on AWS. Early results suggest that Crux can be used to efficiently calculate WCP. WCP shows promise as a useful diagnostic metric focused across an entire workflow.

Our continuing efforts include improvements to the prototype: a custom Crux GUI to address the limitations we observed with the Neo4j Browser, and the functionality to easily allow a user to save workflow critical path results from multiple runs. For Crux to be adopted to

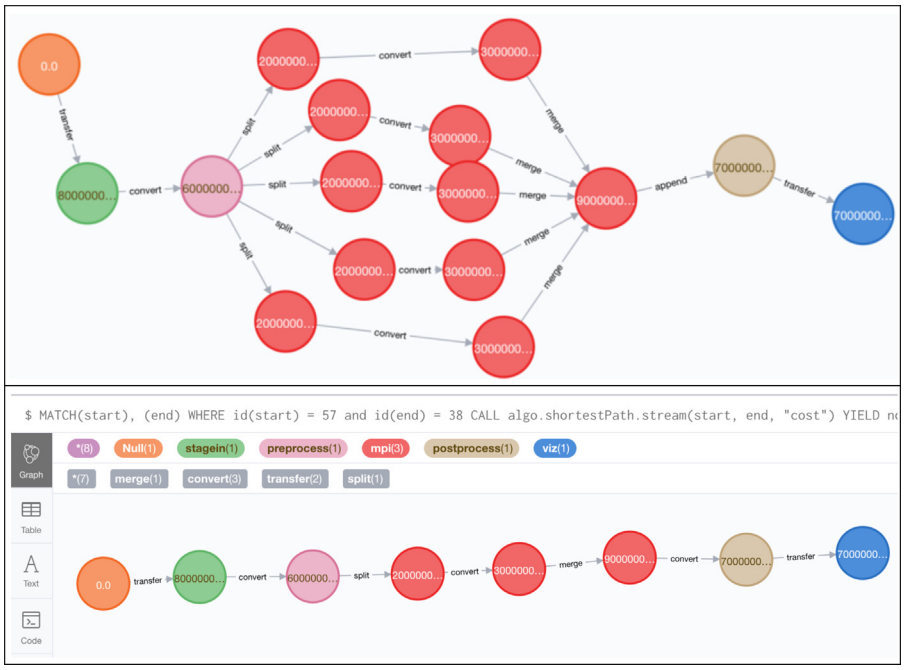


Fig. 11. Generic workflow PAG and WCP. Top image shows the entire PAG with 5 jobs: stage in (green), preprocess (pink), MPI (red), postprocess (tan), and visualization (blue). All Crux PAGs begin with a null vertex (orange) created during database initialization. Bottom image shows the WCP. The critical path in this execution is the path with longest elapsed time through the MPI job. Cost = 9.492.

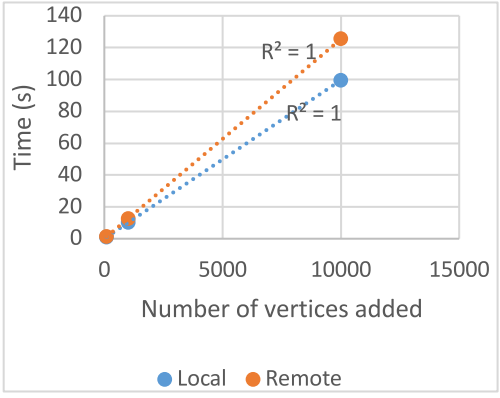


Fig. 12. Time to add vertices to local and remote deployment of Crux. Measured time required for a local Python client make Crux API calls to add 100, 1000, and 10000 data state vertices on local and remote deployments of Crux. In all cases, the time to created vertices was less on the locally deployed Crux. Trendline for both cases suggest a linear relationship between number of vertices to add and overall time ($R^2 = 1$).

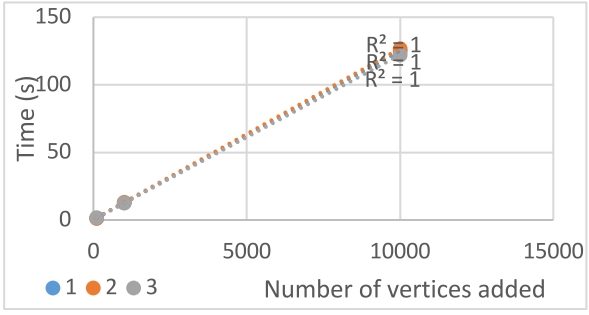


Fig. 13. Time to add vertices against number of Crux API instances. Measured time required for a local Python client make Crux API calls to add 100, 1000, and 10000 data state vertices on a remote deployment of Crux with 1, 2, and 3 API server instances. There was a significant difference in time to add 100 vertices between 1, 2, and 3 API server instances at $p < .05$ [$F(2, 6) = 2.6949$, $p = 0.0135$] where 1 API server instance performed fastest (mean = 1.3792 s). Results did not show significant difference in time for adding 1000 and 1000 vertices between 1, 2, and 3 API server instances. Trendline in all cases suggest a linear relationship between number of vertices to add and overall time ($R^2 = 1$).

production use, the instrumentation must be automated, whereas the initial prototype requires manual insertion of instrumentation. We are currently testing full-scale applications with Crux.

Acknowledgments

David Montoya was instrumental in the development of key insights and ideas contained in this work. Sonja Johanson performed testing and documented the Crux tool, and Yasodha Suriyakumar provided the DroughtHPC example. PSU students Jaspar Alt, Kobe Davis, and Kristina Frye participated in group discussions. Portions of this work were conducted at the Ultrascale Systems Research Center (USRC) supported by Los Alamos National Laboratory, United States under Contract No. DE-AC52-06NA25396 with the U.S. Department of Energy. This work supported in part by the New Mexico Consortium.

References

[1] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P.J. Maechling, R. Mayani, W. Chen, R.F. d. Silva, M. Livny, K. Wenger, Pegasus, a workflow management system for science automation, Future Gener. Comput. Syst. 46 (2015) 17–35.

[2] B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M.B. Jones, E.A. Lee, J. Tao, Y. Zhao, Workflow management and the Kepler system, in: Concurrency and Computation: Practice and Experience, vol. 18, (10) 2006, pp. 1039–1065.

[3] E. Deelman, T. Peterka, I. Altintas, C.D. Carothers, K.K. v. Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, J. Vetter, The future of scientific workflows, Int. J. High Perform. Comput. Appl. 32 (1) (2017) 159–175.

[4] S. Moore, D. Cronk, K. London, J. Dongarra, Review of performance analysis tools for MPI parallel programs, in: EuroPVM/MPI: European Parallel Virtual Machine/ Message Passing Interface Users' Group Meeting, Santorini/Thera, Greece, 2001, pp. 23–26.

[5] J. Sairabanu, M.R. Babu, A. Kar, A. Basu, A survey of performance analysis tools for OpenMP and MPI, Indian J. Sci. Technol. 9 (43) (2016) 1–7.

[6] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, N.R. Tallent, HPCToolkit: Tools for performance analysis of optimized parallel program, Concurr. Comput.: Pract. Exper. 22 (6) (2010) 685–701.
[7] S.S. Shende, A.D. Malony, The TAU parallel performance system, Int. J. High Perform. Comput. Appl. 20 (2) (2006) 287–311.
[8] S. Snyder, P. Carns, K. Harms, R. Ross, G.K. Lockwood, N.J. Wright, Modular HPC I/O characterization with Darshan, in: 2016 5th Workshop on Extreme-Scale Programming Tools, Salt Lake City, UT, 2016.
[9] D. Skinner, Performance monitoring of parallel scientific applications, 2005, [Online]. Available: https://www.osti.gov/servlets/purl/881368-dOvpFA/ (Accessed 15 2021).
[10] E. Deelman, T. Peterka, I. Altintas, C.D. Carothers, K.K. v. Dam, K. Moreland, M. Parashar, L. Ramakrishnan, M. Taufer, J. Vetter, The future of scientific workflows, Int. J. High Perform. Comput. Appl. 32 (1) (2017) 159–175.
[11] Gromacs, [Online]. Available: http://www.gromacs.org/ (Accessed 15 2021).
[12] F. Affinito, A. Emerson, L. Litov, P. Petkov, R. Apostolov, L. Axner, B. Hess, E. Lindahl, M.F. Iozzi, Performance Analysis and Petascaling Enabling of GROMACS, 2012, [Online]. Available: http://www.prace-ri.eu/IMG/pdf/Performance_Analysis_and_Petascaling_Enabling_of_GROMACS.pdf (Accessed15 2021).
[13] C. Herold, B. Williams, Top-down performance analysis of workflow applications, in: The International Conference for High Performance Computing, Networking, Storage, and Analysis, Dallas, TX, 2018.
[14] Y. Suriyakumar, K.L. Karavanic, H. Moradkhani, Performance Analysis of DroughtHPC and Holistic HPC Workflows, ICPP 2018 Research Poster Extended Abstract, 2018, [Online]. Available: http://oaciss.uoregon.edu/icpp18/publications/pos131s2-file1.pdf (Accessed 15 2021).
[15] J.J. Hamman, B. Nijssen, T.J. Bohn, D.R. Gergel, Y. Mao, The variable infiltration capacity model version 5 (VIC-5): infrastructure improvements for new applications and reproducibility, Geosci. Model Dev. 11 (8) (2018) 3481–3496.
[16] H. Cooney, H. Yan, K. Karavanic, H. Moradkhani, A Workflow-Based Performance Study of a Drought Prediction System, 2016, [Online]. Available: https://pdxscholar.library.pdx.edu/cgi/viewcontent.cgi?article=1002&context=mcecs_mentoring (Accessed 15 2021).
[17] APEX Workflows, 2016, [Online]. Available: https://www.nersc.gov/assets/apex-workflows-v2.pdf (Accessed 15 2021).
[18] N. Tallent, D. Kerbyson, A. Hoisie, Representative paths analysis, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17). ACM, 2017.
[19] J.K. Hollingsworth, Critical path profiling of message passing and shared-memory programs, IEEE Trans. Parallel Distrib. Syst. 9 (10) (1998).
[20] C. Yang, B. Miller, Critical path analysis for the execution of parallel and distributed programs, in: Proc. of the 8th Intl. Conf. on Distributed Computing Systems, IEEE, 1988, pp. 366–373.
[21] K.L. Karavanic, Performance Tools and Holistic HPC Workflows, 2018, [Online]. Available: https://dyninst.github.io/scalable_tools_workshop/petascale2018/assets/slides/Karavanic2018.pdf (Accessed 15 2021).
[22] B.H. Sigelman, L.A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C. Shanbhag, Dapper, a Large-Scale Distributed Systems Tracing Infrastructure, 2010, [Online]. Available: https://static.googleusercontent.com/media/research.google.com/en//archive/papers/dapper-2010-1.pdf (Accessed 15 2021).
[23] J. Aldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O'Neill, K.W. Ong, B. Schaller, P. Shan, B. Viscomi, V. Venkataraman, K. Veeraraghavan, Y.J. Song, Canopy: An end-to-end performance tracing and analysis system, in: SOSP '17: Proceedings of the 26th Symposium on Operating Systems Principles, 2017, pp. 34–50.
[24] M. Schulz, Extracting Critical Path Graphs from MPI applications, in: 2005 IEEE International Conference on Cluster Computing, Burlington, MA, 2005, pp. 27–30.
[25] I. Dooley, L.V. Kale, Detecting and using critical paths at runtime in message driven parallel programs, in: 2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW), Atlanta, GA, 2010, pp. 19–23.
[26] J. Chen, R.M. Clapp, Critical-path candidates: scalable performance modeling for MPI workloads, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, 2015, pp. 29–31.
[27] D. Bohme, F. Wolf, D.R. De Supinski, M. Schulz, M. Geimer, Scalable critical-path based performance analysis, in: Proc. of the 26th IEEE Intl. Parallel and Distributed Processing Symp, IEEE, 2012, pp. 1330–1340.
[28] C. Alexander, D. Reese, J. Harden, Near-critical path analysis of program activity graphs, in: Proc. of the Second Intl. Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, IEEE, 1994, pp. 308–317.
[29] U. Meyer, P. Sanders, Δ-Stepping: a parallelizable shortest path algorithm, J. Algorithms 49 (1) (2003) 114–152.
[30] M. Kranjčević, D. Palossi, S. Pintarelli, Parallel delta-stepping algorithm for shared memory architectures, in: 19th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2016), 2016.
[31] FastAPI, [Online]. Available: https://fastapi.tiangolo.com/ (Accessed 15 2021).
[32] neo4j, [Online]. Available: https://neo4j.com/ (Accessed 15 2021).

BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100002

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

MLHarness: A scalable benchmarking system for MLCommons

Yen-Hsiang Chang [a,*], Jianhao Pu [a], Wen-mei Hwu [a], Jinjun Xiong [a,b,*]

[a] University of Illinois at Urbana-Champaign, Urbana, IL, USA
[b] University at Buffalo, Buffalo, NY, USA

ARTICLE INFO

Keywords:
Machine learning
Inference
Benchmarking

ABSTRACT

With the society's growing adoption of machine learning (ML) and deep learning (DL) for various intelligent solutions, it becomes increasingly imperative to standardize a common set of measures for ML/DL models with large scale open datasets under common development practices and resources so that people can benchmark and compare models' quality and performance on a common ground. MLCommons has emerged recently as a driving force from both industry and academia to orchestrate such an effort. Despite its wide adoption as standardized benchmarks, MLCommons Inference has only included a limited number of ML/DL models (in fact seven models in total). This significantly limits the generality of MLCommons Inference's benchmarking results because there are many more novel ML/DL models from the research community, solving a wide range of problems with different inputs and outputs modalities. To address such a limitation, we propose MLHarness, a scalable benchmarking harness system for MLCommons Inference with three distinctive features: (1) it codifies the standard benchmark process as defined by MLCommons Inference including the models, datasets, DL frameworks, and software and hardware systems; (2) it provides an easy and declarative approach for model developers to contribute their models and datasets to MLCommons Inference; and (3) it includes the support of a wide range of models with varying inputs/outputs modalities so that we can scalably benchmark these models across different datasets, frameworks, and hardware systems. This harness system is developed on top of the MLModelScope system, and will be open sourced to the community. Our experimental results demonstrate the superior flexibility and scalability of this harness system for MLCommons Inference benchmarking.

1. Introduction

With the rise of machine learning (ML) and deep learning (DL) innovations in both industry and academia, there is a clear need for standardized benchmarks and evaluation criteria to facilitate comparison and development of ML/DL innovations. MLCommons Inference [1], a standard ML/DL inference benchmark suite with properly defined metrics and benchmarking methodologies, has emerged recently to facilitate such an effort. However, MLCommons Inference only included five models when it was first introduced in 2019, and has only included seven models recently [2]. This limited and slow-growing number of ML/DL models stifles the adoption of MLCommons Inference as a general benchmarking platform because there are many more novel ML/DL models from the research community, solving a wide range of problems with different inputs and outputs modalities. To address such a limitation, we propose MLHarness, a scalable benchmarking harness system for MLCommons Inference, to make MLCommons Inference embrace new models and modalities easily. MLHarness is developed on top of MLModelScope [3] with three distinctive new features: (1) it codifies the required benchmarking environment for MLCommons Inference explicitly, including models, datasets, frameworks, software and hardware stacks; (2) it provides an easy and declarative approach for model developers to contribute their models and datasets to MLCommons Inference; and (3) it supports a wide range of models with varying inputs and outputs modalities. Our experiments show that MLHarness is capable of reporting all the required metrics as defined by MLCommons Inference for models with different input/output modalities and for models both within and beyond MLCommons Inference.

In particular, we make the following contributions: (1) we propose MLHarness, a scalable benchmarking harness system for MLCommons Inference while supporting models beyond those in MLCommons Inference; (2) we extend MLModelScope to provide user-defined pre-processing and post-processing interfaces so that MLModelScope can easily support new models and new modalities; (3) we showcase MLHarness' capabilities as a scalable benchmarking harness system for MLCommons Inference by running experiments on a range of models both within and beyond MLCommons Inference under different frameworks and systems configurations; and (4) we further demonstrate the unique value of scalable benchmarking in identifying abnormal

* Corresponding authors.
E-mail addresses: yhchang3@illinois.edu (Y. Chang), jpu3@illinois.edu (J. Pu), w-hwu@illinois.edu (W. Hwu), jinjunx@illinois.edu, jinjun@buffalo.edu (J. Xiong).

https://doi.org/10.1016/j.tbench.2021.100002
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 4 November 2021

system behaviors and how MLHarness helps to explain those seemingly abnormal behaviors resulting from complex software and hardware interactions.

2. Background

2.1. ML/DL benchmark challenges

ML and DL innovations such as applications, datasets, frameworks, models, and software and hardware systems, are being developed in a rapid pace. However, current practice of sharing ML/DL innovations is to build ad-hoc scripts and write manuals to describe the workflow. This makes it hard to reproduce the reported metrics and to port the innovations to different environments and solutions. Therefore, having a standard benchmarking platform with an exchange specification and well-defined metrics to fairly compare and benchmark the innovations is a crucial step toward the success of ML/DL community.

Previous work includes (1) ML/DL model zoos curated by framework developers [4–9], but they only aim for sharing ML/DL models as a library; (2) package managers for a specific software environment such as Spack [10], while they are just targeting on maintaining packages in different software and hardware stacks; (3) benchmarking platforms such as MLCommons [1,11] and MLModelScope [3], but the former only focuses on few specific models and the latter only focuses on models in computer vision tasks; (4) collections of reproducible MLOps components and architectures [12–14], while their main focuses are on deployment and automation; (5) plug-and-play shareable containers such as MLCube [15], but its generality makes it hard to identify and locate the crucial components for the cause of abnormal behaviors in ML/DL models; (6) simulator of ML/DL inference servers such as iBench [16], but the main focus on capturing data transfer capabilities between clients and servers provides no insights on profiling models. As the above applications either only focus on a specific software and hardware stack, or use ad-hoc approaches to handle specific ML/DL tasks, or are lack of a consistent benchmarking method, it is hard to use them individually to have a well rounded experience when developing ML/DL innovations.

To address these ML/DL benchmark challenges, we propose a new scalable benchmarking system: MLHarness by taking advantage of two open-source projects: MLModelScope [3] for its exchange specification on software and hardware stacks, and MLCommons Inference [1] for its community-adopted benchmarking scenarios and metrics. With MLHarness, we are able to benchmark and compare quality and performance of models on a common ground through a set of well-defined metrics and exchange specification.

2.2. Overview of MLCommons

MLCommons [1,11], previously known as MLPerf, is a platform aims to answer the needs of the nascent machine learning industry. MLCommons Training [11] measures how fast systems can train models to a target quality metric, while MLCommons Inference [1] measures how fast systems can process inputs and produce results using a trained model. Both of these two benchmark suites target on providing benchmarking results on different scales of computing services, ranging from tiny mobile devices to high performance computing data centers. As the main focus of this paper is on benchmarking ML/DL inferences, we only focus on MLCommons Inference in the rest of this paper.

2.2.1. Characteristics of MLCommons Inference

MLCommons Inference is a standard ML/DL inference benchmark suite with a set of properly defined metrics and benchmarking methodologies to fairly measure the inference performance of ML/DL hardware, software, and services. MLCommons Inference focuses on the following perspectives when designing its benchmarking metrics:

- **Selection of Representative Models.** MLCommons Inference selects representative models that are mature, open source, and have earned community support. This permits accessibility and reproducible measurements, which facilitates MLCommons Inference becoming a standardized benchmarking suite.
- **Scenarios.** MLCommons Inference consists of four evaluation scenarios, including single-stream, multistream, server, and offline. These four scenarios aim for simulating realistic behaviors of inference systems in many critical applications.
- **Metrics.** Apart from the commonly used model metrics such as accuracy, MLCommons Inference also includes a set of systems related metrics, such as percentile-latency and throughput. These make MLCommons Inference appealing in satisfying the demand of different use cases, such as 99% percentile-latency for a data center to respond to a user query.

2.2.2. Workflows of MLCommons Inference

Fig. 1 shows the critical components as defined in MLCommons Inference, where the numbers and arrows denote the sequence and the directions of the data flows, respectively. The description of the components follows:

- **Load Generator (LoadGen).** The LoadGen produces query traffics as defined by the four scenarios above, collects logging information, and summarizes benchmarking results. It is a stand-alone module that stays the same across different models.
- **System Under Test (SUT).** The SUT consists of the inference system under benchmarking, including ML/DL frameworks, ML/DL models, software libraries and the target hardware system. Once the SUT receives a query from the LoadGen, it completes an inference run and reports the result to the LoadGen.
- **Data Set.** Before issuing queries to the SUT, the LoadGen needs to let the SUT fetch the data needed for the queries from the dataset and pre-process the data. This is not included in the latency measurement.
- **Accuracy Script.** After all queries are issued and the results are received, the accuracy script will be invoked to validate the accuracy of the model from the logging information.

2.2.3. Limitations of MLCommons Inference

As we can observe from the characteristics and the workflows of MLCommons Inference above, MLCommons Inference involves benchmarking under different scenarios with various metrics, which provides a community acknowledged ML/DL benchmark standard. However, the focus on the seven representative models shadows its advantage because MLCommons Inference only provides ad-hoc scripts for these representative models, and it is hard to extend them to many other models beyond MLCommons Inference.

In fact, the only critical component in MLCommons Inference is the LoadGen, while the other components can be replaced with any inference systems. In this paper, we present how to replace the components other than the LoadGen by MLModelScope [3], an inference platform with a clearly defined exchange specification and an across-stack profiling and analysis tool, and extend MLModelScope so that it becomes a scalable benchmarking harness for MLCommons Inference. This greatly extends the applicability of MLCommons Inference for models well beyond it.

2.3. Overview of MLModelScope

MLModelScope [3] is a hardware and software agnostic distributed platform for benchmarking and profiling ML/DL models across datasets, frameworks and systems.

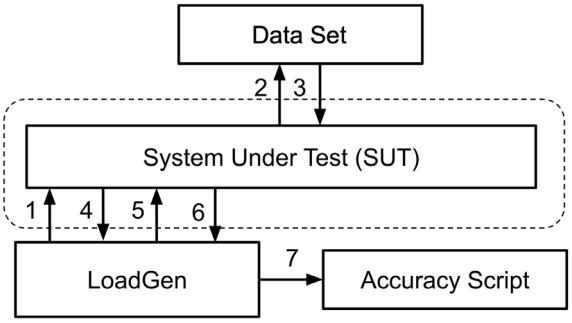


Fig. 1. Workflow of MLCommons Inference [1].

2.3.1. Characteristics of MLModelScope

MLModelScope consists of a specification and a runtime that enable repeatable and fair evaluation. The design aspects follow:

- **Specification.** MLModelScope utilizes the software and model manifests as proposed in DLSpec [17], which capture different aspects of an ML/DL task and ensure usability and reproducibility. The software manifest defines the software requirements, such as ML/DL frameworks to run an ML/DL task. The model manifest defines the logic to run the model for the ML/DL task, such as pre-processing and post-processing methods, and the required artifact sources. An example is shown in Listing 1.
- **Runtime.** The runtime of MLModelScope follows the manifests to set up the required environment for inference. Moreover, MLModelScope includes the across-stack profiling and analysis tool, XSP [18], which introduces a leveled and iterative measurement approach to overcome the impact of profiling overhead. As shown in Fig. 2, MLModelScope captures profiling data for different levels, which enables users to correlate the information and analyze the performance data in different levels.

```yaml
name: Inception-v3 # model name
version: 1.0.0 # semantic version of the model
task: classification
framework: # framework information
    name: TensorFlow
    version: ^1.x # framework version constraint
model: # model sources
    graph_path: https://.../inception_v3.pb
    graph_checksum: 328f68...3a813e
steps: # pre-processing steps
    decode:
        element_type: int8
        data_layout: NHWC
        color_layout: RGB
    crop:
        method: center
        percentage: 87.5
    resize:
        dimensions: [3, 299, 299]
        method: bilinear
        keep_aspect_ratio: true
    mean: [127.5, 127.5, 127.5]
    rescale: 127.5
...
```

Listing 1: An excerpt of manifest from MLModelScope [3]

2.3.2. Limitations of MLModelScope

Although MLModelScope involves a clearly defined specification and is able to run several hundreds of models in different ML/DL frameworks, it currently only supports models for computer vision tasks. While MLModelScope discussed the possibility of using user-defined pre-processing and post-processing inline Python scripts to serve as a universal handler for all kinds of models, MLModelScope did not implement those interfaces but only introduced built-in image manipulations to support computer vision tasks. In this paper, we have actually implemented the user-defined pre-processing and post-processing interfaces and demonstrated its usage on models with different modalities and different pre-processing and post-processing, such as question answering and medical 3D image segmentation.

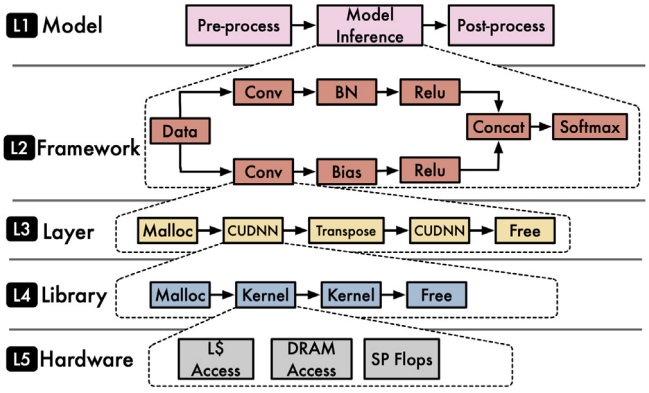


Fig. 2. Profiling levels in MLModelScope [3].

3. MLHarness implementation

This section describes the crucial implementations of MLHarness, a scalable benchmarking harness system for MLCommons Inference [1] for tackling the limitations of MLCommons Inference and MLModelScope [3]. The implementations include the support of user defined pre-processing and post-processing interfaces and the encapsulation of MLModelScope for MLCommons Inference.

3.1. Pre-processing and post-processing interfaces

As described in DLSpec [17] and MLModelScope [3], to make the user defined pre-processing and post-processing interfaces universal, inline Python scripts are chosen to allow great flexibility and productivity, as Python functions can download and run Bash scripts and some C++ code. On the other hand, MLModelScope is implemented in Go; therefore, it is necessary to build a bridge between the runtime of MLModelScope and the embedded Python scripts in the model manifest so that, within the MLModelScope runtime, we can invoke the Python runtime to execute user defined pre-processing and post-processing functions.

A naive solution is to save the input data and the functions as files and execute pre-processing and post-processing functions apart from MLModelScope. However, this approach is impractical since it introduces high serialization and process initialization overhead, and it also makes MLModelScope incapable of supporting streaming data [17].

In order to avoid using intermediate files, we instead use Python/C APIs [19] to embed a Python interpreter into MLModelScope to execute Python functions, as suggested by DLSpec. To use these APIs, we need to implement wrappers in Go to call them. Instead of building these wrappers from scratch, we use the open source Go-Python3 bindings [20]. In this fashion, the Python functions can be executed within MLHarness directly to avoid the problems mentioned in the naive solution.

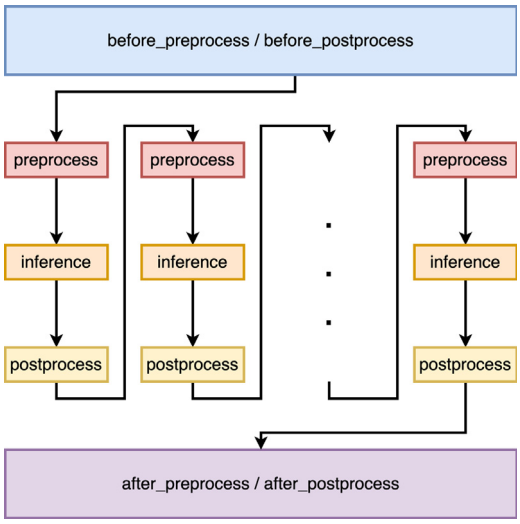


Fig. 3. Workflow of MLHarness.

3.1.1. Implementation details

Fig. 3 shows the invocation sequence of user defined pre-processing and post-processing functions by MLHarness. The `before_preprocess` and the `before_postprocess` functions are invoked only once at the startup stage; the `after_preprocess` function and the `after_postprocess` functions are invoked only once after all inferences are done. These four functions are for the sake of loading datasets, writing logging information to files, and specifying configurations during runtime if necessary. The `preprocess` function and the `postprocess` function are invoked right before and after every model inference, respectively, to pre-process and post-process the inputs and outputs of the model.

```
func Processing(tensor interface{}, functionName string) interface{} {
    pyData := MoveDataToPythonInterpreter(tensor)
    pyFunc := FindTheProcessingFunctionByItsName(functionName)
    pyResult := ExecuteProcessingFunction(pyFunc, pyData)
    result := GetResultFromPythonInterpreter(pyResult)
    return result
}
```

Listing 2: Pre-processing and post-processing implementation in Go

To embed a Python interpreter, we need to initialize it through Python/C APIs at the beginning of MLHarness. Then, as Listing 2 shows, the function handling the embedded Python pre-processing and post-processing scripts consists of four parts, utilizing the Go-Python3 bindings:

- `MoveDataToPythonInterpreter.` Moving data from Go to Python is not easy since the data being processed are large, for example, a tensor representing an image. One solution is to serialize the data at one end, transfer the data as a string, and deserialize at the other end. However, it introduces a high overhead due to the high cost of encoding and decoding. To overcome this problem and to make data transfer efficient, we propose to copy the data in-memory, i.e., we only send the shape of the tensor and the address of its underlying flattened array, and reconstruct the tensor by copying data from the address and by its shape. Note that to guarantee the validity of data transfer, we need to make sure that the underlying flattened array represents the tensor contiguously, particularly in case lazy operations were done on the tensor, such as transposition.

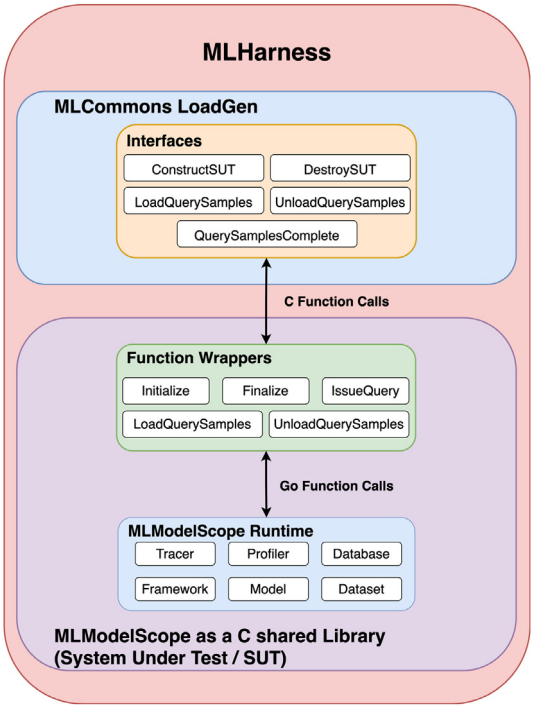


Fig. 4. Structure of MLHarness.

- `FindTheProcessingFunctionByItsName.` The processing functions in the model manifest are registered in the `__main__` module of the Python interpreter during its initialization. To get the corresponding `PyObject` of these functions, we query the `__main__` module by the names of functions, which are the six processing functions as listed in Fig. 3.
- `ExecuteProcessingFunction.` The signatures of the processing functions in the model manifest are in the form of `process(ctx, data)`, where `ctx` is a dictionary capturing additional information in the manifest, and `data` is the tensor we got from `MoveDataToPythonInterpreter`. Therefore, in order to invoke the processing function, we need to call the Go-Python3 binding with the `PyObject` of the processing function, the dictionary of `ctx` and the `data` going to be processed. Note that the `data` is only effective for `preprocess` and `postprocess`, and it is just a `PyObject` of `None` for the other four processing functions.
- `GetResultFromPythonInterpreter.` This is similar to the first part except that it moves data from Python to Go instead of the other way around. Note that we still copy the data in-memory to avoid unnecessary overhead.

3.2. Structure of MLHarness

Fig. 4 shows the encapsulation of MLModelScope for MLCommons Inference. In order to utilize MLCommons Inference defined scenarios and performance metrics, we keep the LoadGen and the accuracy script the same as they were in MLCommons Inference. On the other hand, We replace the built-in SUT and data set in MLCommons Inference with MLModelScope runtime to run the models. In this way, MLModelScope is capable of acting as an easy-to-use black box to respond to the queries issued by the LoadGen in MLCommons Inference, and it also provides across-stack profiling information for further analysis, which are not available when merely using MLCommons Inference.

3.2.1. Implementation details

MLModelScope is developed in Go, but the LoadGen in MLCommons Inference is developed in C++ and used in Python through Python bindings. In order to make the communication between MLModelScope and MLCommons Inference feasible, we build MLModelScope as a C shared library [21], use the ctypes module [22] in Python to load the shared library, and call the functions in the shared library. Three notable implementations are described below:

```yaml
name: MLPerf_BERT # model name
version: 1.0.0 # semantic version of the model
framework: # framework information
    name: PyTorch
    version: '>=1.5.0' # framework version constraint
inputs: # model inputs
    - type: text # input modality
      element_type: string
outputs: # model outputs
    - type: text # output modality
      element_type: string
model: # model sources
    graph_path: https://.../bert.pt
    graph_checksum: c3bb5a...aa1ccd
preprocess: |
  from transformers import BertTokenizer
  import numpy as np
  ...
  class SquadExample(object):
    ...
  class InputFeatures(object):
    ...
  def read_squad_examples(...):
    ...
  def convert_examples_to_features(...):
    ...
  features = []
  tokenizer = BertTokenizer(...)
  examples = read_squad_examples(...)
  convert_examples_to_features(features, examples, tokenizer, ...)
  def preprocess(ctx, data):
    cur = features[int(data)]
    return cur.input_ids, cur.input_mask, cur.segment_ids
postprocess: |
  import numpy as np
  import json
  def postprocess(ctx, data):
    res = np.stack([data[0], data[1]], axis = -1).squeeze(0).tolist()
    return [json.dumps(res)]
...
```

Listing 3: An excerpt of model manifest for BERT

- **Function wrappers.** To simplify the process of building the C shared library and leaving MLModelScope as a black box, we create function wrappers for critical applications in MLModelScope and only export them in the shared library. This includes the `Initialize` and `Finalize` wrappers to initialize and finalize the profiling tools in MLModelScope. It also includes the `LoadQuerySamples`, `IssueQuery`, and `UnloadQuerySamples` wrappers to pre-load and pre-process the data from the data set, handle queries from the LoadGen, and free the memory occupied by the pre-loaded data, respectively.
- **Data transmissions.** It is hard to directly exchange data between Go and Python, since there is no one-to-one correspondence between data types in these two languages. To solve this problem, we utilize the built-in primitive C compatible data types in ctypes [22] for Python and CGO [23] for Go, since they define how to transform data if there is no clear correspondence between data types in C and the corresponding languages. Using this method, the data conversion can be done in-memory instead of through serialization.
- **Blocking statements.** When we exchange data between Go and Python, the garbage collector at one end does not automatically know that it needs to keep the data before the data are really copied or used at the other end, which might result into undefined behaviors. To solve this problem, we need to manually create blocking statements to block garbage collection until a deep copy of the data is made at the other end. This can be done using the `KeepAlive` function [24] in Go and managing reference counts [25] in Python to prevent garbage collection being invoked until the `KeepAlive` is executed and the reference count is decreased to zero, respectively.

3.3. Example of MLHarness

With the help of user defined pre-processing and post-processing interfaces, MLHarness is able to handle various models' inputs and outputs modalities that are not supported in MLModelScope. Also, it is easy to use the model manifest to add models for MLModelScope to report MLCommons Inference defined metrics, which is hard when merely using MLCommons Inference. Listing 3 is the model manifest using the pre-processing and post-processing interfaces for BERT [26], a language representation model, to handle the question answering modality that was not supported in MLModelScope. The pre-processing step uses the tokenizer from the transformers Python third-party library [27] to parse data and prepare input features. The post-processing step reshapes the outputs into the format as defined by the accuracy script. The tedious implementation of the tokenizer is one of the reason why MLModelScope cannot support the question answering modality, since it is hard to create an equivalent built-in alternative inside MLModelScope using Go. On the contrary, through the user defined pre-processing and post-processing interfaces, MLHarness can utilize the community developed Python third-party libraries to overcome this obstacle.

4. Experimental results

We conduct two sets of experiments to demonstrate the success of MLHarness on overcoming the limitations in MLModelScope [3] and MLCommons Inference [1]. In the first set of experiments, we use MLHarness to benchmark models in MLCommons Inference and report MLCommons Inference defined metrics to show that it supports modalities that are not supported in MLModelScope, such as question answering and medical image 3D segmentation. In addition, we show that MLHarness is able to report the results for all four scenarios defined by MLCommons Inference. In the second set of experiments, we use MLHarness to benchmark models beyond MLCommons Inference and report MLCommons Inference defined metrics to show the usage of our newly extended MLModelScope as an easy-to-use black box for MLCommons Inference.

4.1. Experiment setup

Table 1 shows the systems used for experiments. The system naming convention follows the rule as the identifier of the CPU types followed by the acronym of the ML/DL framework, and then the identifier of the GPU type if a GPU is used. There are three system instance categories in total. The first category is an Intel desktop-grade CPU system, including system 9800-ORT-RTX, 9800-PT-RTX, 9800-ORT, 9800-PT, and 9800-MX-RTX. The second category is also an Intel but different desktop-grade CPU system, including system 7820-ORT-TITAN, 7820-PT-TITAN, and 7820-TF. The last category is a server-based system using AMD CPUs, including system AMD-ORT-A100 and AMD-ORT-V100. We choose different combinations of frameworks, software systems and hardware systems in order to demonstrate the flexibility and scalability of MLHarness as a harness for benchmarking.

For both sets of experiments, we report the accuracy, the throughput in the offline scenario, and the throughput and 90 percentile latency in

Table 1
Systems used for experiments.

System annotations	Framework	Processor	Accelerator
9800-ORT-RTX	ONNX Runtime	1x Intel(R) Core(TM) i7-9800X CPU @ 3.80 GHz	1x GeForce RTX 3090
9800-PT-RTX	PyTorch	1x Intel(R) Core(TM) i7-9800X CPU @ 3.80 GHz	1x GeForce RTX 3090
9800-ORT	ONNX Runtime	1x Intel(R) Core(TM) i7-9800X CPU @ 3.80 GHz	None
9800-PT	PyTorch	1x Intel(R) Core(TM) i7-9800X CPU @ 3.80 GHz	None
7820-ORT-TITAN	ONNX Runtime	1x Intel(R) Core(TM) i7-7820X CPU @ 3.60 GHz	1x TITAN V
7820-PT-TITAN	PyTorch	1x Intel(R) Core(TM) i7-7820X CPU @ 3.60 GHz	1x TITAN V
7820-TF	TensorFlow	1x Intel(R) Core(TM) i7-7820X CPU @ 3.60 GHz	None
AMD-ORT-A100	ONNX Runtime	1x AMD EPYC 7702 64-Core Processor	1x A100
AMD-ORT-V100	ONNX Runtime	1x AMD EPYC 7702 64-Core Processor	1x V100
9800-MX-RTX	MXNet	1x Intel(R) Core(TM) i7-9800X CPU @ 3.80 GHz	1x GeForce RTX 3090

Table 2
MLHarness and MLCommons reported results for all four scenarios on 9800-ORT-RTX.

Benchmark suite	Offline (sample/s)	Single-Stream		Server		Multi-Stream	
		(sample/s)	90th percentile latency (ms)	(sample/s)	99th percentile latency (ms)	(sample/query)	99th percentile latency (ms)
MLHarness	133	118	9.2	69	44	5	42
MLCommons Inference	315	308	3.2	121	14	12	44

the single-stream scenario. The accuracy is defined differently for each modalities, including the top-1 accuracy for image classification, mAP scores for object detection, F1 scores for question answering, and mean DICE scores for medical image 3D segmentation. As defined in MLCommons Inference [1], the offline scenario represents applications where all data are immediately available and latency is unconstrained, such as photo categorization; on the contrary, the single-stream scenario represents a query stream with sample size of 1, reflecting applications requiring swift responses, such as real time augmented reality. In order to facilitate the comparison between these two scenarios, we fix the batch size of the inferences to 1 in all experiments. This helps to demonstrate how scenarios can affect the throughput of models.

MLHarness is also capable of reporting results of the other two scenarios as defined by MLCommons Inference [1], which are the server and the multistream scenarios. The server scenario represents applications where query arrival is random and latency is important, such as online translation. The multistream scenario represents application with a stream of queries, where each query consists of multiple inferences, such as multi-camera driver assistance. We demonstrate that MLHarness is able to report the results of these two scenarios by running `ResNet50` on 9800-ORT-RTX.

Note that, because of the limited access to data-center scale systems, we are not able to develop and conduct experiments for the rest of the two models as provided by MLCommons Inference, which are `DLRM` for recommendation system and `RNNT` for speech recognition. But we believe our methodologies as discussed can be easily extended for the two models.

4.2. Results of models in MLCommons Inference

In this set of experiments, we demonstrate the capability of MLHarness on reporting MLCommons Inference [1] defined metrics, by benchmarking representative MLCommons Inference models with a variety of systems.

Table 2 shows the various MLCommons Inference defined experimental results of `ResNet50` produced by MLHarness on system 9800-ORT-RTX. From the results, we observe that using MLHarness, running `ResNet50` on such a target system is able to classify 133 images per second, respond to a query of one image in less than 9.2 ms in 90% of the time if the queries are received contiguously, respond to a query of one image in less than 44 ms in 99% of the time if the queries are received following the Poisson distribution with an average of 69 queries per second, and respond to a query of five images in less than 42 ms in 99% of the time if the queries are received contiguously. Note that the number of queries per second in the server scenario and the number of samples per query in the multistream scenarios are tunable parameters for the system to meet the latency requirements.

We also run the same set of experiments on 9800-ORT-RTX using the original MLCommons Inference flows, as shown in Table 2. The results show that MLCommons Inference performs two to three times better than MLHarness. In order to investigate the discrepancy that MLHarness has a worse performance than MLCommons Inference, we take the Offline scenario as an example and break down the execution time into two parts, including (1) model-inference time for the interval between the model receives pre-processed input tensors and returns output tensors and (2) post-processing time involving generating MLCommons Inference defined format. As Fig. 5 shows, while both MLHarness and MLCommons Inference spend nearly the same amount of time on model inference, the much higher latency for MLHarness to post-process data make it hard to achieve the same performance as reported by MLCommons Inference. The underlying reason of this high latency in MLHarness is due to the aggregated data transferring time between different languages, as data need to be moved several times among ML/DL frameworks, post-processing interfaces and wrappers, while it is not the case for MLCommons Inference since once the inference is done, data always reside in Python. One way to mitigate this high latency is to further optimize MLHarness for MLCommons Inference by responding directly to the LoadGen in the post-processing function instead of transferring data back to MLHarness and then reporting to MLCommons Inference suite through wrappers between different languages.

Table 3 shows the experimental results of `ResNet50` and `MobileNet` for image classification, `SSD MobileNet 300x300` and `SSD ResNet34 1200x1200` for object detection, `BERT` for question answering, and `3D-UNet` for medical image 3D segmentation. All of these models are provided by MLCommons Inference and can be found at its GitHub page [2].

An interesting observation from Table 3 is that the throughput of `ResNet50` on system AMD-ORT-V100, which is 202 samples per second, is higher than that on system AMD-ORT-A100, which is 159 samples per second. This seems to be counter-intuitive as the A100 GPU is two generations newer than the V100 GPU, hence the A100 GPU is supposed to have better performance than V100. With MLCommons' inference methodology alone, we are not able to figure out the reason of this "seemingly abnormal" behavior. This is the place for MLHarness to shine with its extended MLModelScope capabilities. Leveraging the across-stack profiling and analysis capabilities from MLModelScope, we are able to align framework-level spans from the ONNX Runtime profiler and the library-level spans from CUDA Profiling Tools Interface, and capture the detailed view of this strange

Table 3
MLHarness reported results for models in MLCommons Inference.

Model	System	Accuracy	Offline (sample/s)	Single-Stream	
				(sample/s)	90th percentile latency (ms)
MLPerf ResNet50	9800-ORT-RTX	Top1: 76.452%	133	118	9.2
	9800-ORT	Top1: 76.456%	63	62	16
	7820-ORT-TITAN	Top1: 76.456%	118	116	9.2
	7820-TF	Top1: 76.456%	20	20	57
	AMD-ORT-A100	Top1: 76.456%	159	146	6.7
	AMD-ORT-V100	Top1: 76.456%	202	154	6.4
MLPerf MobileNet	9800-ORT-RTX	Top1: 71.676%	196	160	6.5
	9800-ORT	Top1: 71.676%	61	58	23
	7820-ORT-TITAN	Top1: 71.676%	188	159	6.6
	7820-TF	Top1: 71.676%	24	24	44
	AMD-ORT-A100	Top1: 71.666%	358	270	3.7
	AMD-ORT-V100	Top1: 71.676%	382	319	3.2
MLPerf SSD MobileNet 300 × 300	9800-ORT-RTX	mAP: 23.172%	35	32	37
	9800-ORT	mAP: 23.173%	28	28	37
	7820-ORT-TITAN	mAP: 23.173%	30	28	41
	7820-TF	mAP: 23.173%	13	13	78
	AMD-ORT-A100	mAP: 23.170%	20	20	52
	AMD-ORT-V100	mAP: 23.173%	18	18	57
MLPerf SSD ResNet34 1200 × 1200	9800-ORT-RTX	mAP: 19.961%	20	19	54
	9800-ORT	mAP: 19.955%	1.4	1.7	816
	7820-ORT-TITAN	mAP: 19.955%	16	15	66
	7820-TF	mAP: 20.215%	1.4	1.4	704
	AMD-ORT-A100	mAP: 19.957%	23	21	54
	AMD-ORT-V100	mAP: 19.955%	14	13	95
MLPerf BERT	9800-ORT-RTX	F1: 90.874%	41	38	27
	9800-PT-RTX	F1: 90.881%	21	18	67
	9800-ORT	F1: 90.874%	2.2	2.5	487
	9800-PT	F1: 90.874%	0.86	0.85	1305
	7820-ORT-TITAN	F1: 90.874%	30	29	35
	7820-PT-TITAN	F1: 90.874%	27	26	39
	AMD-ORT-A100	F1: 90.879%	92	78	15
	AMD-ORT-V100	F1: 90.874%	29	29	37
MLPerf 3D-UNet	AMD-ORT-A100	mean: 0.85300	0.043	0.045	22655
	AMD-ORT-V100	mean: 0.85300	0.045	0.045	22194

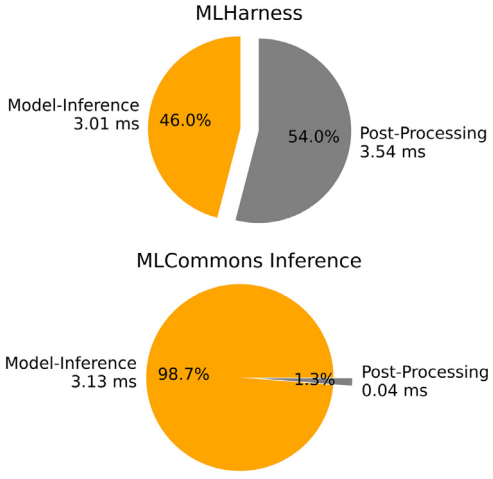


Fig. 5. Break down execution time into model-inference time and post-processing time for MLHarness and MLCommons Inference running Offline scenario with ResNet50 and a single input on 9800-ORT-RTX.

behavior by delving deeper into the results. Fig. 6 shows the performance of `ResNet50` with batch size one across the system AMD-ORT-V100 and system AMD-ORT-A100 at both the layer and the kernel (sub-layer) granularity levels, respectively. At the layer granularity, we observe that the end-to-end inference time on system AMD-ORT-V100 is indeed shorter than that on AMD-ORT-A100, and the reduced runtime mainly comes from the shortened runtime of many Conv2+ReLu layers (in orange color). For example, by focusing only on the second to the last Conv2+ReLu layer, we see that the duration on system AMD-ORT-A100 is almost twice as large as the duration on system AMD-ORT-V100. By zooming into that particular layer at the kernel level granularity, we quickly realize that the two systems have executed different GPU kernels. For system AMD-ORT-V100, there are two major kernels, i.e., cudnn::winograd::generateWinogradTilesKernel and volta_scudnn_winograd_128x128. In contrast, for system AMD-ORT-A100, there is only one major kernel, i.e., implicit_convolve_sgemm. We suspect that this discrepancy in performance is mainly due to the less optimized kernel selection algorithm offered by the newer system CUDNN library (v8.1) for A100 GPUs than for V100 GPUs. This further validates the importance of full-stack optimization for system performance.

In summary, we show that MLHarness is capable of reporting MLCommons Inference defined metrics by encapsulating MLModelScope [3] as an easy-to-use black box into MLCommons Inference, and that our harness system is able to benchmark models that are not supported in MLModelScope, including `BERT` for question answering and `3D-UNet` for medical image 3D segmentation, with the help of the new interfaces for user-defined pre-processing and post-processing functions. Moreover, as MLHarness is built on top of MLModelScope, we are able to utilize its across-stack profiling and analysis capabilities to align the information across the ML/DL framework level and the accelerating library level, and pinpoint critical distinctions between models, frameworks, and system.

4.3. Results of models beyond MLCommons Inference

Unlike the first set of experiments, which focuses on showcasing the success of MLHarness in orchestrating MLCommons Inference [1]

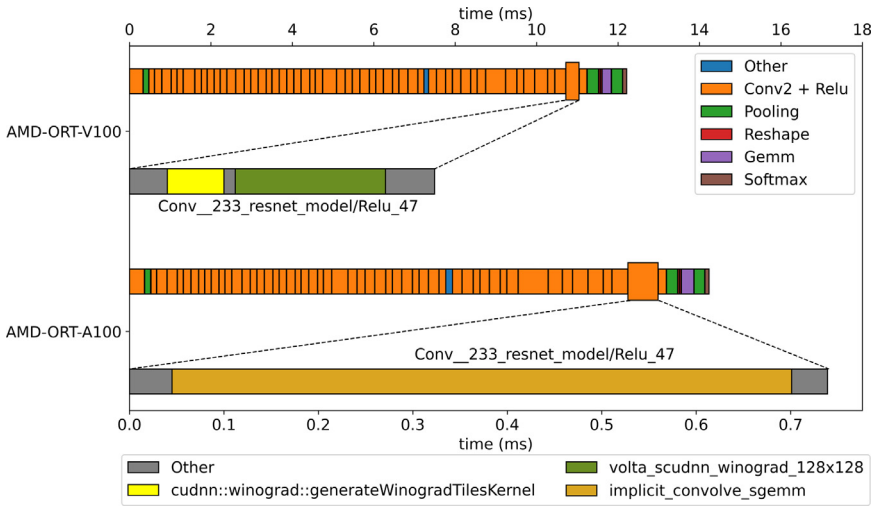


Fig. 6. Performance of ResNet50 with batch size one across systems AMD-ORT-V100 and AMD-ORT-A100 at both layer and kernel (sub-layer) granularity levels, respectively. The axis on the top is the duration to execute each layer in the model, while the axis at the bottom is the duration to execute kernels of the second to the last Conv2 + Relu layer.

Table 4
MLHarness reported results for models beyond MLCommons Inference using PyTorch and ONNX Runtime as ML/DL frameworks.

Model	System	Accuracy	Offline (sample/s)	Single-Stream	
				(sample/s)	90th percentile latency (ms)
TorchVision AlexNet	9800-ORT-RTX	Top1: 56.520%	218	171	6.1
	9800-PT-RTX	Top1: 56.516%	191	154	6.8
	9800-ORT	Top1: 56.522%	86	81	12
	9800-PT	Top1: 56.522%	12	12	90
	7820-ORT-TITAN	Top1: 56.522%	219	168	6.1
	7820-PT-TITAN	Top1: 56.522%	186	152	6.9
TorchVision ResNet18	9800-ORT-RTX	Top1: 69.758%	179	144	7.3
	9800-PT-RTX	Top1: 69.756%	122	113	9.6
	9800-ORT	Top1: 69.758%	128	118	8.8
	9800-PT	Top1: 69.758%	28	32	42
	7820-ORT-TITAN	Top1: 69.758%	175	145	7.2
	7820-PT-TITAN	Top1: 69.758%	132	119	9.2
TorchVision ResNet34	9800-ORT-RTX	Top1: 73.314%	142	124	8.7
	9800-PT-RTX	Top1: 73.306%	90	89	12
	9800-ORT	Top1: 73.314%	72	70	14
	9800-PT	Top1: 73.314%	20	19	65
	7820-ORT-TITAN	Top1: 73.314%	144	125	8.5
	7820-PT-TITAN	Top1: 73.314%	98	93	12
TorchVision ResNet50	9800-ORT-RTX	Top1: 76.130%	129	115	9.4
	9800-PT-RTX	Top1: 76.132%	76	76	15
	9800-ORT	Top1: 76.130%	63	61	16
	9800-PT	Top1: 76.130%	7.9	7.7	149
	7820-ORT-TITAN	Top1: 76.130%	128	112	9.6
	7820-PT-TITAN	Top1: 76.130%	79	78	15
TorchVision ResNet101	9800-ORT-RTX	Top1: 77.374%	100	89	12
	9800-PT-RTX	Top1: 77.376%	59	68	22
	9800-ORT	Top1: 77.374%	36	36	29
	9800-PT	Top1: 77.374%	4.7	4.8	239
	7820-ORT-TITAN	Top1: 77.374%	94	87	12
	7820-PT-TITAN	Top1: 77.374%	60	67	22
TorchVision ResNet152	9800-ORT-RTX	Top1: 78.310%	84	76	15
	9800-PT-RTX	Top1: 78.312%	46	64	26
	9800-ORT	Top1: 78.312%	26	26	41
	9800-PT	Top1: 78.312%	3.5	3.5	324
	7820-ORT-TITAN	Top1: 78.312%	74	72	15
	7820-PT-TITAN	Top1: 78.312%	52	60	26

way of benchmarking MLCommons Inference models, the second set of experiments illustrates how to make use of the exchange specification and the across-stack profiling and analysis tool in MLModelScope [3] to facilitate developments and comparisons of ML/DL innovations in the context of MLCommons Inference methodologies.

Table 4 shows a sample of six models to demonstrate how easy it is to use MLHarness to scale the MLCommons Inference way of benchmarking of various models beyond MLCommons Inference on a variety of system configurations. In this particular example, these results further show the relationships among the depth of the convolutional neural networks, the accuracy, and the throughput. The six models

Table 5
MLHarness reported results for models beyond MLCommons Inference using TensorFlow and MXNet as ML/DL frameworks.

Model	System	Accuracy	Offline (sample/s)	Single-Stream	
				(sample/s)	90th percentile latency (ms)
VGG16	7820-TF	Top1: 70.962%	9.3	9.2	115
	9800-MX-RTX	Top1: 72.852%	100	88	11
VGG19	7820-TF	Top1: 71.056%	8.7	8.6	123
	9800-MX-RTX	Top1: 73.814%	91	82	12

are `AlexNet` along with five models from the `ResNet` family. All of these models are from TorchVision [9], where the implementation details and the reference accuracy can be found at its GitHub page [28]. Again, the success of importing these models into MLHarness using the exchange specification is validated by the accuracy results, where all of them are within at least 99% of the reference accuracy as stated by TorchVision. In addition, the pre-processing and post-processing functions in the exchange specification can be regarded as a reusable component because these models share the same pre-processing and post-processing steps.

Fig. 7 compares the accuracy of the six convolutional neural networks in systems 9800-ORT-RTX to 7820-PT-TITAN as listed in Table 1. The models are placed in increasing order of depth from left to right, with `ResNet152` being the deepest. As expected, there is no huge variance on the accuracy across systems, but the deeper the convolutional neural network is, the more accurate the model is.

Fig. 8 compares the throughput of the six convolutional neural networks in system 9800-ORT-RTX to 7820-PT-TITAN as listed in Table 1. From Fig. 8, we observe that there is a trend that the deeper the convolutional neural network is, the lower the throughput it has. However, for the two systems 9800-ORT and 9800-PT, which are the two system configurations without GPUs, are not following the trend when comparing `AlexNet` and `ResNet18`. As our MLHarness is built on top of MLModelScope, we then use the across-stack profiling and analysis tool, XSP [18], to identify the bottleneck. Table 6 shows the top three most time-consuming layers of `AlexNet` and `ResNet18` identified by XSP on system 9800-PT, which has no GPU support and has PyTorch as the ML/DL framework. It clearly points out that the bottleneck of `AlexNet` is from matrix multiplications of fully connected layers. Although there is also a fully connected layer in `ResNet18` as recorded by XSP, its size is 512 by 1000, which is much smaller than the largest one in `AlexNet`, whose size is 4096 by 4096.

Although in Table 4, we use the same implementations of models by converting PyTorch models to ONNX formats that can be used in ONNX Runtime, it is also valuable to compare the same structure of model with different implementations and training processes. Table 5 shows the experiments on the models from the `VGG` family using TensorFlow and MXNet as ML/DL frameworks, where the models for TensorFlow can be found at TensorFlow Model Graden [29] and the models for MXNet can be found at GluonCV [4]. From Table 5, we can observe that the accuracy is different between implementations of the same model, which further illustrates the difficulty of model reproducibility. This also shows how flexible MLHarness is in terms of running scalable benchmarking across different combinations of models and frameworks by utilizing the extended exchange specification as discussed in this work, and how scalable experimentation helps to identify common issues convincingly.

In summary, these exemplar experiments as discussed in this section show not only that it is easy to add models into MLHarness by utilizing the extended exchange specification and to report MLCommons Inference defined metrics for models that are beyond MLCommons Inference, but also that, with the help of MLModelScope, MLHarness can easily and scalably compare models and extract critical and detailed information, which is impossible when merely using MLCommons Inference.

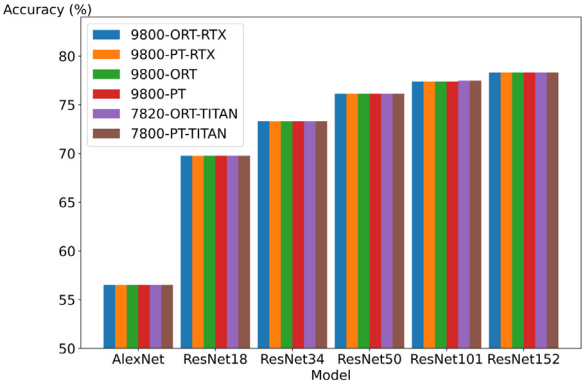


Fig. 7. Accuracy of models in different systems.

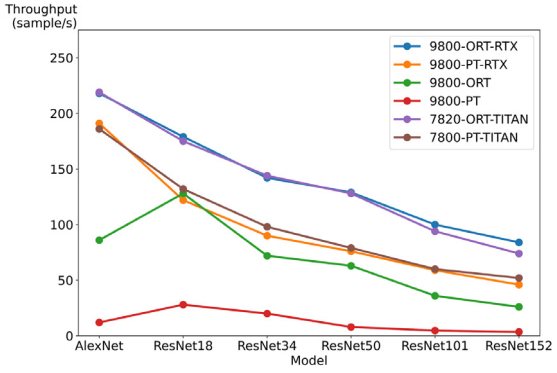


Fig. 8. Offline throughput of models in different systems.

Table 6
The top-3 most time-consuming layers of AlexNet and ResNet18 on system 9800-PT.

AlexNet		ResNet18	
Layer name	Latency (ms)	Layer name	Latency (ms)
aten::mm	47.99	aten::maxpool2d	6.31
aten::mm	17.99	aten::convolution	2.46
aten::mm	4.13	aten::convolution	2.24

4.4. Impact of MLHarness

The experimental results above demonstrate the success of MLHarness in benchmarking ML/DL model inferences by providing an extended exchange specification for researchers to easily plug in their ML/DL innovations and collect a set of well-defined metrics. One of our near future goals is to further extend MLHarness to support MLCommons training [11]. Nevertheless, the impact of MLHarness is not only restricted to ML/DL community. Benchmarking, reproducibility, portability, and scalability are important aspects in any computing-related research, such as high performance computing and computational biology. The success of MLHarness is only a starting point, from which we are aiming for extending the same techniques to other research domains

that utilize heterogeneous computational resources, and providing a scalable and flexible harness system to overcome the similar set of challenges.

5. Conclusion

As ML/DL community is flourishing, it becomes increasingly imperative to standardize a common set of measures for people to benchmark and compare ML/DL models quality and performance on a common ground. In this paper, we present MLHarness, a scalable benchmarking harness system, to remedy and ease the adoption of ML/DL innovations. Our experimental results show superior flexibility and scalability of MLHarness for benchmarking and porting, by utilizing the extended exchange specification and reporting community acknowledged metrics. We also show that with the help of MLHarness, we are able to easily pinpoint critical distinctions between ML/DL innovations, by inspecting and aligning profiling information across stacks.

Acknowledgments

This work is supported in part by IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a research collaboration as part of the IBM AI Horizons Network.

References

[1] V.J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J.S. Gardner, I. Hubara, S. Idgunji, T.B. Jablin, J. Jiao, T.S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A.T.R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, Y. Zhou, Mlperf inference benchmark, 2019, http://arxiv.org/abs/1911.02549.

[2] Mlcommons inference, 2021, URL https://github.com/mlcommons/inference, Accessed: 2021-07-22.

[3] A. Dakkak, C. Li, J. Xiong, W. Hwu, MlModelScope: A distributed platform for model evaluation and benchmarking at scale, 2020, CoRR https://arxiv.org/abs/2002.08295.

[4] Gluoncv, 2021, URL https://cv.gluon.ai/model_zoo/index.html, Accessed: 2021-07-21.

[5] Modelhub, 2021, URL http://modelhub.ai/, Accessed: 2021-07-22.

[6] Modelzoo, 2021, URL https://modelzoo.co/, Accessed: 2021-07-21.

[7] Onnx model zoo, 2021, URL https://github.com/onnx/models, Accessed: 2021-07-21.

[8] Tensorflow hub, 2021, URL https://www.tensorflow.org/hub, Accessed: 2021-07-21.

[9] Torchvision, 2021, URL https://pytorch.org/vision/stable/index.html, Accessed: 2021-07-21.

[10] T. Gamblin, M. LeGendre, M.R. Collette, G.L. Lee, A. Moody, B.R. de Supinski, S. Futral, The spack package manager: bringing order to HPC software chaos, in: SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2015, pp. 1–12, http://dx.doi.org/10.1145/2807591.2807623.

[11] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, A. Ike, B. Jia, D. Kang, D. Kanter, N. Kumar, J. Liao, G. Ma, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V.J. Reddi, T. Robie, T.S. John, T. Tabaru, C.-J. Wu, L. Xu, M. Yamazaki, C. Young, M. Zaharia, Mlperf training benchmark, 2019, http://arxiv.org/abs/1910.01500.

[12] Architecture for mlops using TFX, kubeflow pipelines, and cloud build, 2021, URL https://bit.ly/39P6JFk Accessed: 2021-07-21.

[13] G. Fursin, Collective knowledge: organizing research projects as a database of reusable components and portable workflows with common APIs, 2020, CoRR https://arxiv.org/abs/2011.01149.

[14] Mlops: Model management, deployment, lineage and monitoring with azure machine learning, 2021, URL https://docs.microsoft.com/en-us/azure/machine-learning/concept-model-management-and-deployment, Accessed: 2021-07-21.

[15] Mlcube, 2021, URL https://github.com/mlcommons/mlcube, Accessed: 2021-07-21.

[16] W. Brewer, G. Behm, A. Scheinine, B. Parsons, W. Emeneker, R.P. Trevino, Ibench: a distributed inference simulation and benchmark suite, in: 2020 IEEE High Performance Extreme Computing Conference (HPEC), 2020, pp. 1–6, http://dx.doi.org/10.1109/HPEC43674.2020.9286169.

[17] A. Dakkak, C. Li, J. Xiong, W. Hwu, Dlspec: A deep learning task exchange specification, 2020, CoRR https://arxiv.org/abs/2002.11262.

[18] C. Li, A. Dakkak, J. Xiong, W. Wei, L. Xu, W. Hwu, Across-stack profiling and characterization of machine learning models on GPUs, 2019, CoRR http://arxiv.org/abs/1908.06869.

[19] Python/c API reference manual, 2021, URL https://docs.python.org/3/c-api/, Accessed: 2021-07-21.

[20] Go-Python3, 2021, URL https://github.com/DataDog/go-python3, Accessed: 2021-07-21.

[21] Go package help, 2021, URL https://pkg.go.dev/cmd/go/internal/help, Accessed: 2021-07-21.

[22] Ctypes — A foreign function library for python, 2021, URL https://docs.python.org/3/library/ctypes.html, Accessed: 2021-07-21.

[23] Command cgo, 2021, URL https://pkg.go.dev/cmd/cgo, Accessed: 2021-07-21.

[24] Go package runtime, 2021, URL https://pkg.go.dev/runtime, Accessed: 2021-07-21.

[25] Reference counting, 2021, URL https://docs.python.org/3/c-api/refcounting.html, Accessed: 2021-07-21.

[26] J. Devlin, M. Chang, K. Lee, K. Toutanova, BERT: pre-training of deep bidirectional transformers for language understanding, 2018, CoRR http://arxiv.org/abs/1810.04805.

[27] T. Wolf, L. Debut, V. Sanh, J. Chaumond, C. Delangue, A. Moi, P. Cistac, T. Rault, R. Louf, M. Funtowicz, J. Davison, S. Shleifer, P. von Platen, C. Ma, Y. Jernite, J. Plu, C. Xu, T.L. Scao, S. Gugger, M. Drame, Q. Lhoest, A.M. Rush, Transformers: State-of-the-art natural language processing, in: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, Association for Computational Linguistics, Online, 2020, pp. 38–45, URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[28] Torchvision github, 2021, URL https://github.com/pytorch/vision/tree/v0.10.0, Accessed: 2021-08-05.

[29] H. Yu, C. Chen, X. Du, Y. Li, A. Rashwan, L. Hou, P. Jin, F. Yang, F. Liu, J. Kim, J. Li, TensorFlow Model garden, 2020, https://github.com/tensorflow/models.

BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100003

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Performance optimization opportunities in the Android software stack ☆

Varun Gohil [a,*,1], Nisarg Ujjainkar [b,1], Joycee Mekie [b], Manu Awasthi [a]

[a] Ashoka University, India
[b] Indian Institute of Technology Gandhinagar, India

ARTICLE INFO

Keywords:
CPU utilization
Android smartphone
Workload characterization

ABSTRACT

The smartphone hardware and software ecosystems have evolved very rapidly. Multiple innovations in the system software, including OS, languages, and runtimes have been made in the last decade. Although, performance characterization of microarchitecture has been done, there is little analysis available for application performance bottlenecks of the system software stack, especially for contemporary applications on mobile operating systems.

In this work, we perform system utilization analysis from a software perspective, thereby supplementing the hardware perspective offered by prior work. We focus our analysis on Android powered smartphones, running newer versions of Android. Using 11 representative apps and regions of interest within them, we carry out performance analysis of the entire Android software stack to identify system performance bottlenecks.

We observe that for the majority of apps, the most time-consuming system level thread is a frame rendering thread. However, more surprisingly, our results indicate that *all apps* spend a significant amount of time doing Inter Process Communication (IPC), hinting that the Android IPC stack is a ripe target for performance optimization via software development and a potential target for hardware acceleration.

1. Introduction

Smartphones have become an integral part of our daily lives. People depend on smartphones for many tasks related to business, finance, entertainment, and social interactions. Currently, there are more than 2 billion mobile devices in use worldwide [1]. The Ericsson Mobility Report 2019 states that there are 6.1 billion mobile broadband subscriptions globally and the number of Long-Term-Evolution (LTE) subscriptions have grown to 3.9 billion [2]. This widespread adoption of mobile devices can be largely attributed to increasing device affordability, which has been made possible due to numerous hardware and software innovations. This includes the open-source nature of the Android Operating System [1], which has allowed smartphone vendors to customize the software stack for their hardware. As a result, Android has quickly gained a majority market share for smartphones [3].

Smartphones are very interesting from a system design perspective since they need to provide a number of functionalities that require general purpose as well as special purpose compute. As a result, smartphone SoCs have evolved rapidly to become complex ecosystems incorporating many specialized IP blocks, including DSPs and GPUs in addition to general purpose CPUs [1]. The number and diversity of architectures of such units has also increased over time to accommodate the evolving needs of applications.

Many recent efforts have been made to understand the performance bottlenecks and utilization characteristics of smartphone devices [4–7]. However, most prior studies focus on bottom-up understanding of smartphone utilization from an architectural design perspective. For example, [5] present the distribution of computation amongst ARM's big and little cores. They also study clock frequencies at which one can perform computations on a mobile device in an energy efficient manner. These studies are important since mobile SoC architectures evolve rapidly and characterization of new architectures is important to understand and alleviate performance bottlenecks of new architectures.

The software stack for smartphones has been evolving even faster than hardware. Android has been following a yearly release cycle in recent years, with each iteration adding more functionality and optimizations [18]. As a result, every release causes major changes to the software stack which potentially lead to performance bottlenecks. Knowledge of these bottlenecks is not only useful for optimizing the next generation apps but also for making decisions about future architectural innovations. Despite its importance, there is a lack of understanding of software bottlenecks in both the apps as well as the system software. Understanding and enumerating performance bottlenecks of the software stack remains an important endeavor that has not been taken up in earnest by the systems research community. However,

☆ This work is supported through grants received from Huawei Technology India (DSA2020112121) and Ashoka University (R/IFR/CMS/MAW/18).
* Corresponding author.
E-mail address: varun.gohil@ashoka.edu.in (V. Gohil).
[1] Both authors contributed equally.

https://doi.org/10.1016/j.tbench.2021.100003
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 5 November 2021

Table 1
Applications traced and their Region of Interest.

Category	Application	Regions of Interest (ROI)
PDF Viewer	Adobe Acrobat [8]	Read PDF
Camera	Camera	Take a picture, Record a video
Game	Candy Crush [9]	Play one level of the game
Social Network	Facebook [10]	Scroll through the feed
Mailing app	Gmail [11]	Send mail
Virtual Assistant	Google Assistant [12]	Perform a query
Browsing app	Google Chrome [13]	Search, Scroll through a page
Location app	Google Maps [14]	Search a location, Zoom into a location
Audio Streaming	Spotify [15]	Play a song in background, Play a song in foreground
Messaging app	WhatsApp [16]	Send a message
Video Streaming	YouTube [17]	Play a video

Apart from the regions of interest mentioned above, we also trace the launch of each of the apps.

recent announcements from technology companies [19] indicate that there exists a large room for performance improvement in the Android software stack.

We believe that a top-down analysis of application characteristics will augment our understanding of mobile devices by supplementing prior work. Hence, we study the software subsystem of Android based smartphones by tracing the entire system (application + operating system) stack at runtime, capturing performance bottlenecks. Prior works [4,6,7] have measured CPU utilization using Thread Level Parallelism (TLP) as a metric to identify the amount of parallelism the hardware can exploit. While TLP is a useful metric to decide the number of cores to be placed on the chip, it does not provide information on the computation being performed by the cores and the functionality supported by the computations. Knowledge of the functionality for which the computation is being performed is necessary to optimize software and to design novel hardware accelerators to be used alongside the CPU. Generally, in Android smartphones, a particular thread or a group of threads is responsible for a particular functionality. By identifying the threads having high execution times, one can identify the functionality that consumes higher CPU time and should be optimized. Hence, we focus this paper on trying to answer the following questions.

- Which are the most time-consuming threads per app?
- Are there any common threads across a cross section of apps that end up consuming the most time?
- Which threads take up the most time during app launch?

We believe that this type of analysis will help the process of developing high performance software but and helps identify potential hardware acceleration opportunities for mobile devices. Since many previous studies have pointed out the importance of app launch times for user engagement and experience [20], we also pay special attention to app launches as a region of interest. Overall, the major contributions of this work are as follows:

- We identify and perform system-level tracing of eleven popular mobile applications on actual hardware, running Pie version of Android (Android 9), which helps us analyze time consumed by application and OS threads.
- To better represent performance information, we group threads into bins based on their functionalities. This helps us increase interpretablity of results and analyze the time consumed per functionality.
- We identify that for majority of applications, the most time consuming thread is a system-managed thread named `RenderThread` or another thread involved in frame rendering.
- Using thread bins, we identify that although the most time-consuming thread is almost always a thread related to frame rendering, a larger portion of execution time is consumed by the group of threads responsible for Inter Process Communication (IPC). This insight makes inter process communication a potential target for software optimization and hardware acceleration.

Table 2
Smartphone details.

Technical specifications	
Device model	Nokia 6.1 Plus
Operating System	Android Pie
Architecture	ARM 64-bit
CPU	Qualcomm Snapdragon 636
Cpu Cores	8
GPU	Adreno (TM) 509
RAM	6 GB
Resolution	1080 × 2280
Display PPI	431

2. Methodology

2.1. Applications traced

We choose eleven applications for our study, each of which represents a common use case of a smartphone. For example, we include Google Chrome [13] as a browsing app, Youtube [17] as video-streaming app, WhatsApp [16] as a messaging app, and Gmail [11] as a mailing app. Most of the selected apps come pre-installed in the majority of Android smartphones. We select the remaining apps based on their popularity which we measure using their position on Google Play [21] Store's Top Charts. The selected apps were at the top of the Top Charts when we performed our study .

Prior work [1] suggests that one should divide the applications into *regions-of-interest* (ROI) to gain deeper insight into the applications. A region-of-interest (ROI) is a smaller portion of the application's execution which performs a particular task. For example, Google Chrome has multiple regions-of-interest like performing a search, switching a tab, and scrolling. Each of these ROIs deals with a specific functionality of Google Chrome. The reason for dividing the applications into ROIs is that these individual ROIs can directly influence user-experience and studying them independently of each other reduces the complexity of analysis that needs to be performed. We provide a comprehensive list of all applications we trace and their ROIs in Table 1. Apart from the ROIs mentioned in Table 1, we also trace the app launches for all apps.

2.2. Tools and setup

For system-level (app + operating system) tracing, we use the Systrace [22] tool. Systrace is a tool shipped with Android Studio and is primarily used for analyzing the performance of an Android device. It is a wrapper around Atrace [23] and Ftrace [24]. Atrace performs user space tracing while ftrace traces the Linux kernel. The traces capture not only the threads spawned by the app, but also background threads being executed by the Android operating system. From the traces obtained using Systrace, we find the time for which each thread executes on the processor core.

To trace the ROIs, we start Systrace tracing and perform the task related to the ROI. We immediately stop Systrace tracing when the task

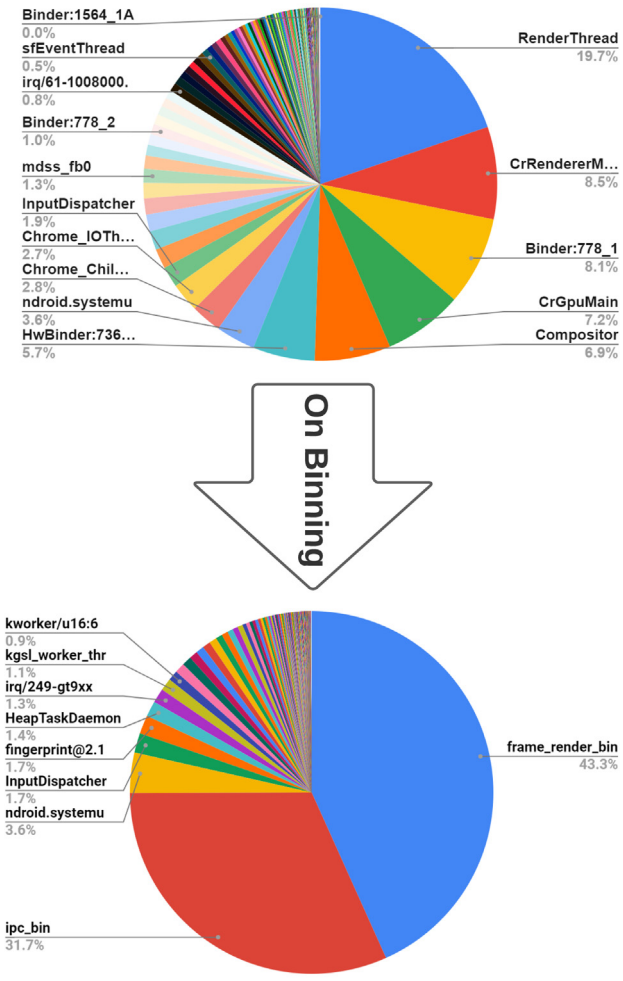


Fig. 1. Effect of Binning. Results for Google Chrome's scrolling ROI.

of the ROI ends. We perform tracing for each region of interest of each app at least five times.

We perform our experiments on Nokia 6.1 Plus [25] smartphone. It runs the stock Android Pie (Android 9) operating system. Further details about the smartphone are presented in Table 2.

2.3. Binning threads

The Android operating system and the apps spawn a large number of threads. Since Systrace performs system level tracing the generated traces have information for a large number of threads. This leads to the resulting plot being cluttered and difficult to interpret. Hence, to reduce clutter and improve interpretability, we group threads working for a common functionality into a single bin. We identify two major bins which aid our analysis. They are:

- Frame Rendering Bin (FR Bin)
- Inter-Process Communication Bin (IPC Bin)

Fig. 1 shows the effect of thread binning. The pie chart on the top in Fig. 1 shows the execution time distribution across individual threads for Google Chrome's scrolling ROI. After thread binning, the pie chart on top is transformed to the one on the bottom. The latter shows execution time distribution among selected bins and the remaining threads. Observing the bottom pie-chart, we can easily infer that the major portion of execution time is spent on frame rendering. We were able to create two classes of thread bins based on the functionality of individual threads. While binning threads, we ensured that threads were mapped to correct bins and that no thread was mapped to more than one bin.

Table 3
List of threads within bins.

Frame rendering bin
RenderThread
surfaceflinger
UiThread
Compositor
CrGpuMain
CrRendererMain
android.display
mdss_fb0
DispSync
android.anim
Above list is not exhaustive
Inter Process Communication Bin
Binder
HwBinder
Chrome_IOThread
Chrome_ChildIOT

Frame Rendering Bin : The Frame Rendering (FR) bin is a group of all threads which are responsible for rendering a frame on the mobile device's screen. Table 3 provides a list of threads within this bin. The major threads within this bin are `RenderThread`, `SurfaceFlinger` and `UiThread`.

Inter Process Communication Bin : The Inter Process Communication (IPC) bin is a group of all threads that are executed to share information between processes. Table 3 provides a comprehensive list of all threads within this bin. The major threads within the IPC bin are `Binder` and `HwBinder`.

3. Results and observations

In this section, we discuss the answers to the questions that we initially set out to answer in Section 1.

3.1. What are the most time-consuming threads/bins per app?

Table 4 shows the most time consuming threads for each region of interest for all eleven applications. We observe that for most ROIs across applications, `RenderThread` is the most time-consuming thread. `RenderThread` is a system-managed thread that is primarily responsible for offloading rendering work to GPU to reduce the burden on `UiThread` [26]. By doing so it ensures the animations are smooth even when the `UiThread` is delayed, which is essential to maintain Quality-of-Service (QoS) for the user [26]. `RenderThread` is the most time-consuming thread in ROIs like scrolling in Facebook and Chrome, messaging using WhatsApp and Gmail, recording a video, or playing a song in foreground on Spotify. All these ROIs involve frequent modifications to the user display which justify most time being consumed by `RenderThread`.

For the game Candy Crush, `GLThread` is the most time-consuming thread. `GLThread` is also a rendering thread and is responsible for performing OpenGL graphics rendering operations [27]. Similarly, for Google Maps' "Zoom into a location" ROI, `GLViewThreadImp` is the most time consuming thread. `GLViewThreadImp` is responsible for managing Views, which are basic building blocks of user-interface components, of the OpenGL graphics library [28,29]. For Google Chrome Search ROI, we observe that `CrRendererMain` is the most time-consuming thread. `CrRendererMain` is the renderer thread for a webpage. As per Chromium's documentation, `CrRendererMain` runs the javascript, html and css code which is displayed on the screen [30].

Table 4
Most time-consuming thread and bin per ROI. Numbers within parenthesis indicate percentage execution time.

Application	Region of Interest	Most time consuming thread	Most time consuming bin
Adobe	Read PDF	om.adobe.reade (13.6%)	FR (26.7%)
Camera	Take a picture	PostProcessingImag (14.5%)	IPC (30.3%)
Camera	Record Video	RenderThread (11.0%)	IPC (22.1%)
Candy Crush	Play 1 level	GLThread (45.2%)	FR (54.2%)
Facebook	Scroll	RenderThread (17.2%)	IPC (23.1%)
Gmail	Send Mail	RenderThread (17.3%)	IPC (29.8%)
Google Assistant	Query	RenderThread (11.2%)	IPC (25.4%)
Google Chrome	Scroll	RenderThread (19.4%)	FR (43.4%)
Google Chrome	Search	CrRendererMain (13.4%)	IPC (33.8%)
Google Maps	Search Location	Jit thread pool (11.6%)	IPC (26.4%)
Google Maps	Zoom into Location	GlViewThreadImp (17.1%)	FR (26.6%)
Spotify	Play Music in Background	AndroidOut_1D (6.2%)	IPC (20.6%)
Spotify	Play Music in Foreground	RenderThread (27.4%)	FR (39.2%)
Whatsapp	Send Message	RenderThread (19.4%)	IPC (35.8%)
YouTube	Play Video	ExoPlayerImplIn (8.8%)	IPC (38.6%)

Overall, the most time-consuming threads for these ROIs are involved in rendering the frame on user display.

For YouTube's "Play a Video" ROI we observe that `ExoPlayerImplIn` is the most time-consuming thread. This thread runs ExoPlayer that is an alternative media player for Android [31,32]. For Camera's "Take a Picture" ROI, `PostProcessingImag` thread is the highest time-consumer. From the thread's name, we hypothesize that this thread might be involved in an image's post-processing which involves tasks like setting the exposure, white balance, and applying selected filters. Unfortunately, we do not find any documentation on `om.adobe.reade` and `AndroidOut_1D` threads and hence cannot comment on their functionality.

Overall, for 7 out of 15 ROIs concerning the eleven applications involved in our study, `RenderThread` is the most time-consuming thread. Further, the highest time-consuming threads in 10 of 15 ROIs are working on the appropriate rendering of the frame. One should also note that the execution time of `RenderThread` is not contiguous. Execution times of multiple instances of `RenderThread` are added together to obtain the total execution time. We find that each individual instance of `RenderThread` is short-lived, on average it takes 0.73 ms to execute, and there exists thousands (1000–2000) of such instances within each region of interest.

The above results may lead one to conclude that frame rendering is the major time consumer for the applications since the most time consuming thread for majority of applications is related to frame-rendering. However, we find that this is not the case when we analyze the results for thread bins. Table 4 shows that the most time consuming bin is the Inter Process Communication (IPC) bin. The IPC bin is the highest time consumer for 10 out of 15 ROIs across the applications. This indicates that even though the major time-consuming thread is related to frame rendering, as a whole, threads used to communicate between processes are the larger time-consumer than threads involved in frame rendering. This observation indicates that inter process communication might be a bigger bottleneck for mobile applications than frame rendering.

3.2. What are the time-consuming threads which are common across apps?

We isolate the common time-consuming threads across applications. We believe optimizing these threads would result in higher performance benefits across applications. We observe the following time-consuming threads to be common across apps:

`RenderThread`: It is the most time consuming thread for 7 out of 15 ROIs under consideration and it is one of the top three most time consuming threads for 11 out of 15 ROIs. It offloads the rendering tasks to GPU from the `UiThread`, to maintain the smoothness of animations by avoiding frame drops [26].

`surfaceflinger`: It is the dominant time-consuming thread after `RenderThread` within the Frame Rendering bin. It is one of the top three most time consuming threads for 5 out of the 15 ROIs. The `surfaceflinger` thread takes in multiple items from various graphics buffers and composes them into a single buffer which is then sent to the user display [33].

`Binder`: The `Binder` threads are a major time consumer for the Inter Process Communication bin. They are used for communication within application processes and within framework and application processes [34]. The framework processes are managed by the Android framework and are device-independent.

`HwBinder`: Similar to `Binder` threads, `HwBinder` threads are also a major time consumer for the Inter Process Communication bin. They are used for communication between framework and vendor processes [34]. The vendor processes are processes spawned by the code that the vendors add to Android framework and are generally device-dependent.

Table 5
Most time-consuming thread and bins on app launch. Numbers within parenthesis indicate the percentage of execution time occupied by the thread/bin.

Application	Most time consuming thread	Most time consuming bin
Adobe	om.adobe.reade (12.6%)	FR (23.5%)
Camera	RenderThread (14.9%)	IPC (35.2%)
Candy Crush	GLThread (44.6%)	FR (57.9%)
Facebook	Jit thread pool (11.6%)	IPC (12.6%)
Gmail	Jit thread pool (10.2%)	IPC (32.1%)
Google Assistant	RenderThread (11.6%)	IPC (39.7%)
Google Chrome	RenderThread (11.6%)	IPC (36.6%)
Google Maps	Jit thread pool (14.0%)	IPC (23.2%)
Spotify	m.spotify.musi (13.9%)	FR (18.0%)
Whatsapp	RenderThread (19.4%)	IPC (36.0%)
YouTube	RenderThread (12.5%)	IPC (25.1%)

3.3. What are the time-consuming threads during an app launch?

App launches are crucial regions of interest in the context of smartphones. One might think that reducing app launch time results in fewer benefits than reducing the app's running time. Although this statement is true and intuitive, app launches are important because of the usage pattern of smartphones. Many users have a large number of short-lived sessions on their smartphones. These short sessions last for less than 10 seconds [20]. During these short sessions, a long app launch time significantly degrades user experience, which is the reason why several efforts have been made to optimize app launch time. For example, Android preserves an apps memory even after it is closed, so the time taken by an app launch in the future can be reduced [35].

We trace the app launches of each of the apps listed in Table 1. Table 5 shows the most time-consuming thread and bin during the launch of the applications. We observe that `RenderThread` consumes a large percentage of execution time for the majority of the

applications. During an app launch, `RenderThread` is the most time-consuming thread for 5 out of the 11 apps, while it is in the top 2 most time-consuming threads for 9 out of the 11 apps. This is expected since when a new application is launched, new views corresponding to the launched application need to be rendered on the screen.

Similar to other ROIs, the Inter Process Communication bin is the highest time consumer during an app launch. This indicates that optimizing Inter process communication would also optimize app launches which would directly improve Quality of Service (QoS).

4. Related work

Several prior publications have focused on evaluating performance and energy of smartphones by characterizing the hardware. For example, Gao et al. [6,7] demonstrated that mobile applications had low Thread-Level Parallelism (TLP) leading to under utilization of allocated cores. A recent work by [5] studied the core utilization in smartphone architectures which have both big and little cores. They report that standalone applications rarely utilize all big cores during execution, however during application launches or updates all big cores are utilized to meet latency targets and avoid degradation in user experience. Most of these works primarily try to answer the question, *"For what percentage of execution time is the core being utilized?"*. While answering the above question is crucial to identify performance inefficiencies, it does not provide insights into the system software stack that may help alleviate these bottlenecks. Our work supplements the prior work by identifying the functionalities (IPC and RenderThread) which have the highest execution time, which on optimization would lead to significant performance benefits.

There have been some research that takes a software-first approach for performance analysis of smartphone applications. [36] use static code analysis to identify frequently occurring performance bug patterns in applications. Further, [37] develop a tool that can automatically detect performance bottlenecks on Android smartphones. However given the nature of the Android ecosystem and the frequent major release cycles require constant performance bottleneck analysis of the system software stack as well. Our work complements such works which perform a software-focused performance analysis. Instead of using any form of static analysis, we identify the time consuming threads of smartphone applications by actually running the applications on a real-world smartphone and provide targets for performance optimization.

5. Limitations and future work

Our current study is limited to Android Version 9. Because of the quick moving nature of the Android ecosystem, owing to yearly release cycles, new versions of Android had been released while we were undertaking this study.

In addition, there is a lack of performance analysis tools for the Android ecosystem, unlike x86/x64, where a large number of open source, well maintained performance analysis tools exist, this is not the case for Android on ARM. Lack of performance analysis tools severely hampers the types of analyses that can be carried out. The analysis done in this paper was carried out using Systrace [22], which is supported for Android version 9. However, more recent Android versions provide a tool called Perfetto [38] for system-level tracing. Further, Perfetto on Android 9 requires the system tracing service to be turned on, which was not possible due to the fact that we performed our experiments on stock android [39]. These factors compelled us to limit out study to Android 9. However, we believe a study similar to this work across Android versions could potentially reveal important performance optimization trends. We also believe future work would be a more comprehensive study by using more smartphone models and different Android versions on each model.

The scope of this work is limited to answering the question *"Which functionality or subsystem of the Android system stack takes up highest portion of execution time?"*. Although extremely important, this work does not reveal what part within the subsystem needs to be optimized and what kind of optimizations would be beneficial. For example, our work indicates that the IPC bin consumes higher portion of execution time but it does not point out which exact components of the IPC subsystem should be optimized to reduce this time. As we have alluded to before, this is primarily due to the of lack of tools which can be used for such analysis. Tools like *Systrace* do not provide such information.

The presented analysis is limited to an Android smartphone. We could not perform similar analysis on smartphones with other operating systems because there do not exist any open-source tools that may act as alternatives/equivalents of *Systrace* for those operating systems.

Finally, the work focuses on Regions of Interest (ROIs) for analyzing the execution time breakdown. The authors have tried to select the most relevant ROIs for each application, which is similar to studies done in the past, which are based on the most common user behavioral patterns, and whose performance determined user engagement [1,20]. However, we acknowledge that the set of ROIs for each application is not necessarily the most representative nor is it necessarily exhaustive. Future work will focus on identifying a much more representative and exhaustive set of regions-of-interest for the application.

6. Conclusion

In this work, we performed a system level performance bottlenecks analysis for an Android smartphone for eleven popular applications. Our results demonstrate that for *all* applications, the highest time consuming thread is either `RenderThread` or another thread related to frame rendering. Further, on grouping threads into bins based on their functionality, we find that the highest time consuming functionality is Inter Process Communication. We find similar distribution in time consumption for both app executions and app launches. Our results identify that software optimization and hardware acceleration should target Inter Process Communication to maximize performance and improve user experience.

References

[1] V.J. Reddi, H. Yoon, A. Knies, Two billion devices and counting, IEEE Micro 38 (1) (2018) 6–21.
[2] Ericsson mobility report Q2 update August 2019, pp. 4, URL: https://www.ericsson.com/4912aa/assets/local/mobility-report/documents/2019/ericsson-mobility-report-q2-2019-update.pdf.
[3] IDC - Smartphone Market Share - OS, IDC: The Premier Global Market Intelligence Company (2021) URL: https://www.idc.com/promo/smartphone-market-share.
[4] M. Halpern, Y. Zhu, V.J. Reddi, Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction, in: 2016 IEEE International Symposium on High Performance Computer Architecture, HPCA, 2016, pp. 64–76, http://dx.doi.org/10.1109/HPCA.2016.7446054.
[5] J. Whitehouse, Q. Wu, S. Song, E. John, A. Gerstlauer, L.K. John, A study of core utilization and residency in heterogeneous smart phone architectures, in: Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering ICPE '19, 2019.
[6] C. Gao, A. Gutierrez, R.G. Dreslinski, T. Mudge, K. Flautner, G. Blake, A study of thread level parallelism on mobile devices, in: 2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2014, pp. 126–127, http://dx.doi.org/10.1109/ISPASS.2014.6844468.
[7] C. Gao, A. Gutierrez, M. Rajan, R.G. Dreslinski, T. Mudge, C. Wu, A study of mobile device utilization, in: 2015 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2015, pp. 225–234, http://dx.doi.org/10.1109/ISPASS.2015.7095808.
[8] Adobe acrobat reader: PDF viewer, editor & creator - Apps on google play, URL: https://play.google.com/store/apps/details?id=com.adobe.reader&hl=en.
[9] Candy crush saga – Apps on google play, URL: https://play.google.com/store/apps/details?id=com.king.candycrushsaga&hl=en{_}IN.
[10] Facebook, (2021) URL: https://www.facebook.com/.
[11] Gmail, URL: https://www.google.com/gmail/.
[12] Google Assistant | Your own personal google, URL: https://assistant.google.com/intl/en{_}in/.
[13] Google chrome - The new chrome & most secure web browser, URL: https://www.google.com/chrome/.

[14] Google Maps, (2021) URL: https://www.google.com/maps.
[15] Music for everyone - Spotify, URL: https://www.spotify.com/in/.
[16] WhatsApp, , URL: https://www.whatsapp.com/.
[17] YouTube, URL: https://www.youtube.com/.
[18] Can android "O" de-fragment android ?, URL: https://www.counterpointresearch.com/can-android-o-de-fragment-android/.
[19] D. Burke, What's new in Android 12 Beta, 2021 URL: https://android-developers.googleblog.com/2021/05/whats-new-in-android-12-beta.html.
[20] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, D. Estrin, 2021.
[21] Google play, URL: https://play.google.com/store.
[22] Overview of system tracing, Android Developers, (2021) URL: https://developer.android.com/studio/profile/systrace.
[23] Atrace/atrace.c - platform/system/extras - git at google, URL: https://android.googlesource.com/platform/system/extras/+/jb-mr1-dev-plus-aosp/atrace/atrace.c.
[24] FTrace, URL: https://www.kernel.org/doc/Documentation/trace/ftrace.txt.
[25] Nokia 6.1 plus, URL: https://www.nokia.com/phones/e_in/nokia-6-plus.
[26] E. Marletti, Understanding the RenderThread, Medium (2017) URL: https://medium.com/@workingkills/understanding-the-renderthread-4dc17bcaf979.
[27] GLSurfaceView, Android Developers, (2021) URL: https://developer.android.com/reference/android/opengl/GLSurfaceView.
[28] GLView | Tizen Docs, URL: https://docs.tizen.org/application/native/guides/ui/efl/mobile/component-glview/.
[29] View, Android Developers (2021) URL: https://developer.android.com/reference/android/view/View.
[30] Understanding about:tracing results - The chromium projects, URL: https://www.chromium.org/developers/how-tos/trace-event-profiling-tool/trace-event-reading.
[31] ExoPlayer, Android Developers, (2021) URL: https://developer.android.com/guide/topics/media/exoplayer.
[32] ExoPlayer, URL: https://exoplayer.dev/.
[33] SurfaceFlinger and WindowManager, Android Open Source Project (2021) URL: https://source.android.com/devices/graphics/surfaceflinger-windowmanager.
[34] Using binder IPC | Android open source project, URL: https://source.android.com/devices/architecture/hidl/binder-ipc.
[35] Manage your app's memory | Android developers, URL: https://developer.android.com/topic/performance/memory.
[36] Y. Liu, C. Xu, S.C. Cheung, Characterizing and detecting performance bugs for smartphone applications, in: Proceedings of the 36th International Conference on Software Engineering, 2014, pp. 1013–1024.
[37] Y. Gao, W. Dong, H. Huang, J. Bu, C. Chen, M. Xia, X. Liu, Whom to blame? Automatic diagnosis of performance bottlenecks on smartphones, IEEE Trans. Mob. Comput. 16 (2017) 1773–1785.
[38] Perfetto : System profiling, app tracing and trace analysis, URL: https://perfetto.dev/.
[39] Quickstart: Record traces on android, URL: https://perfetto.dev/docs/quickstart/android-tracing.

BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100004

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Benchmarking feature selection methods with different prediction models on large-scale healthcare event data

Fan Zhang [a], Chunjie Luo [a,b,c], Chuanxin Lan [a], Jianfeng Zhan [a,b,c,*]

[a] Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China
[b] School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100049, China
[c] International Open Benchmark Council (BenchCouncil)

ARTICLE INFO

Keywords:
Feature selection
Genetic algorithm
Deep neural networks
Healthcare prediction

ABSTRACT

With the development of the Electronic Health Record (EHR) technique, vast volumes of digital clinical data are generated. Based on the data, many methods are developed to improve the performance of clinical predictions. Among those methods, Deep Neural Networks (DNN) have been proven outstanding with respect to accuracy by employing many patient instances and events (features). However, each patient-specific event requires time and money. Collecting too many features before making a decision is insufferable, especially for time-critical tasks such as mortality prediction. So it is essential to predict with high accuracy using as minimal clinical events as possible, which makes feature selection a critical question. This paper presents detailed benchmarking results of various feature selection methods, applying different classification and regression algorithms for clinical prediction tasks, including mortality prediction, length of stay prediction, and ICD-9 code group prediction. We use the publicly available dataset, Medical Information Mart for Intensive Care III (MIMIC-III), in our experiments. Our results show that Genetic Algorithm (GA) based methods perform well with only a few features and outperform others. Besides, for the mortality prediction task, the feature subset selected by GA for one classifier can also be used to others while achieving good performance.

1. Introduction

Over the past decades, the Electronic Health Record (EHR) technique is developed; vast volumes of digital clinical data are generated, making it possible for Clinical Decision Support Systems (CDSSs) to make better decisions. For example, public databases such as MIMIC-III [1] have promoted the research in clinical predictions. Based on those databases, different severity scoring systems, traditional machine learning algorithms, and DNNs are developed and continuously improved to achieve better clinical prediction tasks such as patient mortality, disease classification, and length of hospital stay.

Traditional severity scores like Simplified Acute Physiology Score (SAPS-II) [2], the Sepsis-related Organ Failure Assessment (SOFA) [3], and Acute Physiology and Chronic Health Evaluation (APACHE) [4] are standard for mortality prediction in practice. Clinicians usually choose the patient-specific events they used based on their experience. Then a standard process is implemented. First, a severity score is calculated based on the relative events, usually measured within the first 24 h after ICU admission. Second, a simple model such as logistic regression is applied to the score to predict the final death probability.

Recent work shows that DNN and Super Learner (SL) algorithms perform better than single traditional classifiers and severity scoring systems [5–8]. To improve the predictive performance, many DNN models are developed. Purushotham et al. [6] proposed a Multimodal Deep Learning Model (MMDL) to process an extensive feature set, which consists of 141 features, and got very good predicting results. Harutyunyan et al. [7] proposed a multitask LSTM-based method to predict four clinical prediction tasks. In addition to DNN, SL is also studied extensively and shows promising results. Pirracchio et al. [5] provided and assessed the performance of the Super ICU Learner Algorithm (SICULA). Lee et al. [8] trained case-specific Random Forests (RF) to make mortality prediction and exhibited the best AUROC compared with other single models such as death counting, logistic regression, and decision tree.

No matter which method we use, feature selection is an important part. First, medical databases store vast amounts of clinical events and not all of them are related to the target task. Second, minimal clinical events enable doctors to make timely decisions. For severity scoring systems, a set of alternated related events is chosen based on clinicians' experience. A simple subset of those events is selected according to correlation coefficient or other index associated with the target concept of the prediction task [2–4,10]. Traditional machine learning and deep learning algorithms usually take the same features directly used in

* Corresponding author at: Institute of Computing Technology, Chinese Academy of Sciences, Beijing 100190, China.
E-mail addresses: zhangfan@ict.ac.cn (F. Zhang), luochunjie@ict.ac.cn (C. Luo), lanchuanxin@ict.ac.cn (C. Lan), zhanjianfeng@ict.ac.cn (J. Zhan).

https://doi.org/10.1016/j.tbench.2021.100004
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 3 November 2021

Table 1
Comparison of benchmarking works.

		[7]	[6]	[9]	This work
Number of features	Smaller feature set	✓	✓	✓	✓
	Larger feature set		✓	✓	✓
Feature type	Non-time series		✓	✓	✓
	Time-series	✓	✓	✓	✓
Feature selection methods	Severity score	✓	✓	✓	✓
	Machine learning			✓	✓
	Evolutionary computing GA			✓	✓
Classifications methods	Traditional machine learning	✓	✓	✓	✓
	DNN	✓	✓		✓
Prediction tasks	Mortality	✓	✓	✓	✓
	Length of stay	✓	✓		✓
	Phenotyping	✓			
	ICD-9 code group	✓	✓		✓

severity scores [6,7] or take a similar method to select features [9]. In this paper, we do an exhaustive evaluation of various feature selection methods, and our main contributions are listed below.

(1) Benchmarking feature selection methods including traditional severity scores, machine learning-based feature selection methods, and evolutionary computing GA for three clinical prediction tasks.
(2) Compare feature subsets selected by GA for different classifiers. The results show that for the mortality prediction task, the features chosen by GA are universal for different classifiers.

The rest of this paper is organized as follows: in Section 2, we provide an overview of the related work; in Section 3, we describe the dataset and methods we employed; in Section 4, the benchmarking experiments and results are reported and discussed in detail; in Section 5, we summarize the paper.

2. Related work

First, we summarize the feature selection algorithms that are applied in the medical field. Then we discuss the existing benchmarks on healthcare datasets, especially for MIMIC-III. The comparison of benchmarks is listed in Table 1.

The first severity scores proposed such as APACHE [4], APACHE-II [11], and SAPS [12] selected features based on experience of medical experts. Further work usually used statistical methods to calculate correlation coefficient associated with target prediction task, such as [2,13,14]. Since publication, all of the methods have been continuously modified to improve the predictive performance [15]. A lot of work employed GA to select risk factors and predict in-hospital mortality [9,10,16–19]. In this work, we report an exhaustive set of benchmarking results of feature selection methods, including GA.

Public datasets such as MIMIC-III have promoted the benchmarking of models for clinical prediction tasks. Purushotham et al. [6] benchmarked deep learning models based on an extensive feature set and get high Area Under the Receiver Operating Characteristic Curve (AUROC) and Area under Precision–Recall Curve (AUPRC). The complete feature set we used is the same as the feature set C in [6], which contains 136 time-series features and five non-time series features. Harutyunyan et al. [7] first benchmarked four clinical prediction tasks and presented a multitask classifier. The most significant difference between us and previous works is that we benchmark feature selection algorithms, especially GA, instead of classification or regression algorithms. Krishnan et al. [9] proposed a GA-based model to make mortality prediction. We extend the benchmark to the other two prediction tasks and combine GA with DNN models to get higher AUROC and AUPRC. Johnson et al. [20] reproduced 28 published works for mortality prediction, and the results showed that it is a big challenge to reproduce other people's work without public code.

Table 2
Summary statics of cohort selection.

Data	Total
Admissions in the MIMIC-III (V1.4)	58,976
The first admissions	46,520
First admissions of adult patients	38,424
Patients died 24 h after the admissions	35,643

3. Materials and methods

3.1. Dataset preprocessing

MIMIC-III is developed by the Massachusetts Institute of Technology (MIT)'s Laboratory for Computational Physiology and contains around 60,000 intensive care unit admissions. MIMIC-III (v1.4) consists of 46,520 distinct patients and 58,976 admissions, from where we select 35,643 admissions for our experiments. We extracted data from 5 commonly used tables, namely inputevents, outputevents, chartevents, labevents, prescriptions tables. The statistics of cohort selection are tabulated in Table 2. We selected the patient cohort based on the following criteria:

- Only adult patients, whose age was >15 years at the time of ICU admission, were selected.
- Only the first admission was included for each patient. This decision uses the earliest available data to predict and ensure similar data selection compared to other related works.
- We only include the patients who died 24 h after the first admission.

Because the original data from MIMIC-III has erroneous records such as missing values, inconsistent units, etc., we clean data according to [6], which includes the following procedures: (a) Unify the units. (b) Select one valid record. For multiple records simultaneously, take the average values for numerical data and bring the first for categorical data. (c) Re-sample and fill-in the data. Time-series data is divided into hours and a forward–backward imputation is done to impute the missing values.

3.2. Prediction tasks

For benchmark, we select three clinical prediction tasks which are important in critical care research and are commonly studied by machine learning researchers. The first is in-hospital mortality which is important for doctors to take effective actions for patient care in Intensive Care Units (ICUs) [9]. The second is ICD-9 code group prediction, where we divide the ICD-9 codes into 20 groups according to [6] and treat it as a multi-classification problem. The third is length of stay prediction, which is to predict the hospital stay after admission.

Table 3
Genetic algorithm.

Genetic Algorithm
Input: Dataset D(X, Y) , classifier C and target number of features N
Output: Optimal N features X' for C
1. Randomly generate N features from X as X_0
2. Calculate fitness for X_0, which is AUC for D(X_0, Y) and C.
3. Set $X'=X_0$
4. **while** i ≤ 10000 **do**
5. Generate a new feature set X_i by randomly replacing one feature in X'
6. Calculate fitness for X_i
7. **if** X_i.fitness > X'.fitness:
8. $X'=X_i$
9. **if** X'.fitness ⩾ 1
10. **return** X'
11. **return** X'

3.3. Feature selection/extraction methods

We extract 136 time-series features and five non-time series features as our full feature set according to [6]. Those features are selected based on clinical significance and missing rate while containing all features used in severity scoring systems such as SAPS-II and SOFA. Based on this feature set, different feature selection methods are evaluated including GA based methods, scoring methods and machine learning methods.

GA is a metaheuristic inspired by the process of natural selection. It can be used to generate high-quality solutions to optimization and search problems by the process of mutation, crossover, and selection [21]. It is proved to be useful in feature selection as a wrapper feature selection technique [9]. Table 3 lists the GA procedure we used.

For scoring methods, we choose two popularly used severity scores, namely SAPS-II and SOFA. SAPS-II [2] is designed to predict the probability of hospital mortality. It can be calculated based on 17 variables which can be expand into 20 raw features of our complete feature set. SOFA [3] score can be calculated based on 6 variables which can be expand into 17 raw features of our complete feature set.

For machine learning methods, Principal Component Analysis (PCA) [22] and Recursive Feature Elimination (RFE) are chosen.

PCA is a widely used filter feature extraction technique, which projects the data to a new orthogonal space and then chooses a few of the essential features to achieve dimensionality reduction. RFE is a wrapper feature selection technique that selects features based on the accuracy of the subsequent classifiers.

3.4. Classification/regression methods

For machine learning we use three common commonly used algorithms: decision tree, Bayesian ridge regression, and logistic regression. For DNN we use three types of deep models, namely Feedforward Neural Networks (FNN), Recurrent Neural Networks (RNN), and Multimodal Deep Learning Model (MMDL) according to [6].

4. Benchmarking results

Based on the MIMIC-III dataset, we report the experimental results for three prediction tasks, which answer the following questions: (a) Can DNN models use relatively small feature subsets to perform as well as the full feature set? (b) Whether the subset of features selected by GA is universal for different classifiers of the same task?

4.1. Mortality prediction task evaluation

Tables 4, 5 show the results of mortality prediction task. Because PCA cannot handle time-series data, it is blank for RNN and MMDL results. We can observe that: (a) Deep learning-based prediction models perform better than traditional machine learning-based models and obtain around 2%–20% and 10%–30% improvement for AUROC and AUPRC, respectively. (b) GA performs better and obtains around 6%–18% and 10%–40% improvement over other methods for AUROC and AUPRC, respectively. (c) Compared with using all features (141 features), GA gets similar or even better results with only 20 features.

Fig. 1 shows the result of applying the features selected by GA-MMDL to other classifiers. We can observe that: (a) Although GA is a wrapper feature selection method, the features that GA-MMD chooses can also be used to other classifiers and achieve almost as good results as the GA combined with the specific classifier.

4.2. ICD-9 code prediction task evaluation

We divided the dataset into 20 classes according to [6] and treated it as a multi-classification task. However, because Bayes and LR in the package of scikit-learn do not support multi-classification tasks, we perform binary classification for these two algorithms and then calculate the average AUROC and AUPRC as the final results.

Tables 6, 7 show the results of icd-9 code group prediction task. We can observe that: (a) Deep learning prediction models perform better than traditional machine learning models and obtain around 9%–25% and 20%–35% improvement for AUROC and AUPRC, respectively. (b) GA performs better and obtains around 1%–10% improvement over other methods for both AUROC and AUPRC. (c) Compared with using all features (141 features), GA gets similar or even better results with only 20 features.

Fig. 2 shows the result of applying the features selected by GA-MMDL to other classifiers. We can observe that: (a) Only for deep learning models, the features that GA-MMDL chooses achieve similar results with the GA combined with the specific classifier. For machine learning classifiers, the features that GA-MMDL chooses do not perform well.

4.3. Length of stay prediction task evaluation

Table 8 shows the results of the length of the stay prediction task. We remove the LR algorithm since it is not capable of processing regression problems. We can observe that: (a) GA performs better than others and obtains around 6%–30% improvement over other methods in terms of MSE (in hours). (b) Compared with using all features (141 features), GA gets similar or even better MSE with only 20 features, save time and money.

Fig. 3 shows the result of applying the features selected by GA-MMDL to other regressors. We can observe that: (a) Only for the RNN model, the features GA-MMDL selects have good performance. For the other classifiers, the features that GA-MMDL chooses do not perform well.

4.4. Statistical significance tests of GA

From the above results, we can see that GA always performs better than other feature selection methods. We think this is because as the number of iterations (epochs) increases, GA can reach the local optimum. If there are enough iterations, GA can even reach the global optimum. The price of high precision is time overhead. However, this feature selection method only need to be trained once offline and in actual application doctors can quickly make a diagnosis with a few features.

To further check whether GA's improved performance is statistically significant compared with others we conducted statistical tests. The

Table 4
AUROC of in-hospital mortality prediction task.

Algorithm		Score method (Features)		Feature extraction/selection (Features)			All features (141)
		SAPS-II (20)	SOFA (17)	PCA (20)	RFE (20)	GA (20)	
ML	DT	0.6055 ± 0.0171	0.6780 ± 0.0052	0.5856 ± 0.0119	0.7009 ± 0.0066	**0.7657 ± 0.0085**	0.7631 ± 0.0119
	Bayes	0.8002 ± 0.0046	0.8018 ± 0.0011	0.7672 ± 0.0057	0.8166 ± 0.0085	0.9158 ± 0.0058	**0.9177 ± 0.0047**
	LR	0.5448 ± 0.0042	0.5570 ± 0.0043	0.5824 ± 0.0691	0.5581 ± 0.0044	0.7206 ± 0.0090	**0.7348 ± 0.0053**
DL	FNN	0.7945 ± 0.0059	0.7978 ± 0.0036	0.7863 ± 0.0067	0.8034 ± 0.0089	0.9207 ± 0.0026	**0.9263 ± 0.0032**
	RNN	0.8459 ± 0.0017	0.8651 ± 0.0037	–	0.8309 ± 0.0016	**0.9326 ± 0.0064**	0.9312 ± 0.0023
	MMDL	0.8532 ± 0.0033	0.8781 ± 0.0003	–	0.8282 ± 0.0057	**0.9376 ± 0.0036**	0.9345 ± 0.0029

Table 5
AUPRC of in-hospital mortality prediction task.

Algorithm		Score method (Features)		Feature extraction/selection (Features)			All features (141)
		SAPS-II (20)	SOFA (17)	PCA (20)	RFE (20)	GA (20)	
ML	DT	0.1992 ± 0.0121	0.3196 ± 0.0035	0.1756 ± 0.0080	0.3619 ± 0.0087	**0.4652 ± 0.0024**	0.4564 ± 0.0141
	Bayes	0.3374 ± 0.0158	0.3795 ± 0.0154	0.2706 ± 0.0040	0.3790 ± 0.0183	0.6109 ± 0.0271	**0.6333 ± 0.0096**
	LR	0.1549 ± 0.0041	0.1750 ± 0.0021	0.1498 ± 0.0146	0.1696 ± 0.0087	0.4301 ± 0.0088	**0.4320 ± 0.0025**
DL	FNN	0.3548 ± 0.0139	0.3871 ± 0.0073	0.2900 ± 0.0196	0.4010 ± 0.0076	0.7050 ± 0.0059	**0.7122 ± 0.0119**
	RNN	0.4299 ± 0.0116	0.5454 ± 0.0077	–	0.4396 ± 0.0101	**0.7486 ± 0.0189**	0.7283 ± 0.0074
	MMDL	0.4410 ± 0.0116	0.5687 ± 0.0070	–	0.4439 ± 0.0153	**0.7551 ± 0.0093**	0.7389 ± 0.0057

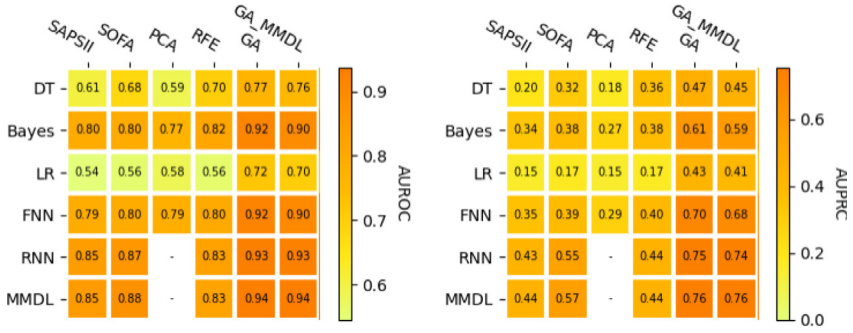


Fig. 1. Apply the features selected by GA-MMDL to other classifiers for the mortality prediction task.

Table 6
AUROC of icd-9 code group prediction task.

Algorithm		Score method (Features)		Feature extraction/selection (Features)			All features (141)
		SAPS-II (20)	SOFA (17)	PCA (20)	RFE (20)	GA (20)	
ML	DT	0.5735 ± 0.0004	0.5746 ± 0.0003	0.5609 ± 0.0005	0.5633 ± 0.0011	**0.5875 ± 0.0005**	0.5868 ± 0.0011
	Bayes	0.6818 ± 0.0010	0.6694 ± 0.0023	0.6523 ± 0.0027	0.6993 ± 0.0052	**0.7542 ± 0.0036**	0.7505 ± 0.0023
	LR	0.5454 ± 0.0004	0.5395 ± 0.0008	0.5438 ± 0.0006	0.5687 ± 0.0024	**0.6064 ± 0.0023**	0.6032 ± 0.0011
DL	FNN	0.8087 ± 0.0008	0.8034 ± 0.0014	0.8036 ± 0.0014	0.8158 ± 0.0002	0.8383 ± 0.0005	**0.8408 ± 0.0007**
	RNN	0.8147 ± 0.0012	0.8121 ± 0.0001	–	0.8229 ± 0.0003	0.8351 ± 0.0005	**0.8427 ± 0.0009**
	MMDL	0.8197 ± 0.0007	0.8179 ± 0.0003	–	0.8217 ± 0.0007	0.8384 ± 0.0007	**0.8440 ± 0.0007**

Table 7
AUPRC of icd-9 code group prediction task.

Algorithm		Score method (Features)		Feature extraction/selection (Features)			All features (141)
		SAPS-II (20)	SOFA (17)	PCA (20)	RFE (20)	GA (20)	
ML	DT	0.3719 ± 0.0012	0.3740 ± 0.0018	0.3522 ± 0.0013	0.3538 ± 0.0014	**0.3825 ± 0.0009**	0.3820 ± 0.0008
	Bayes	0.4523 ± 0.0006	0.4440 ± 0.0031	0.4096 ± 0.0031	0.4722 ± 0.0052	**0.5206 ± 0.0051**	0.5201 ± 0.0020
	LR	0.3440 ± 0.0004	0.3400 ± 0.0005	0.3414 ± 0.0002	0.3739 ± 0.0035	**0.4500 ± 0.0043**	0.3896 ± 0.0018
DL	FNN	0.6698 ± 0.0033	0.6602 ± 0.0022	0.6552 ± 0.0047	0.6717 ± 0.0003	0.7148 ± 0.0009	**0.7265 ± 0.0007**
	RNN	0.6820 ± 0.0006	0.6780 ± 0.0005	–	0.6897 ± 0.0008	0.7148 ± 0.0018	**0.7311 ± 0.0022**
	MMDL	0.6911 ± 0.0021	0.6902 ± 0.0004	–	0.6925 ± 0.0029	0.7214 ± 0.0015	**0.7340 ± 0.0005**

results are tabulated in Table 9. We can see that GA is statistically significant for mortality and length of stay prediction tasks but not for ICD-9 code group classification. This may be because it is more difficult to improve AUROC and AUPRC of multi-classification than binary-classification tasks.

5. Summary

This paper presented comprehensive benchmarking results of different feature selection methods and classification algorithms on three clinical prediction tasks. We demonstrated that: (a) GA always performs

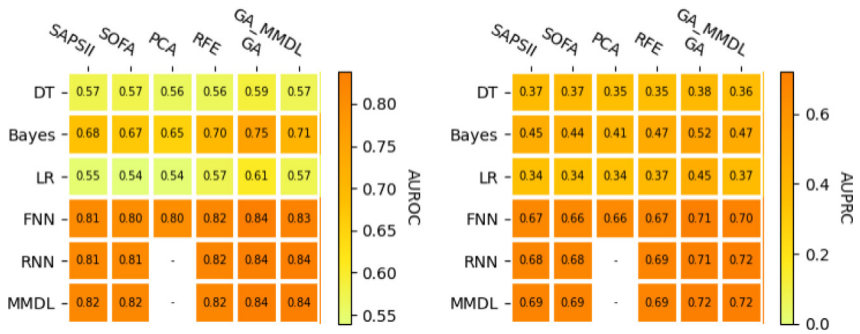


Fig. 2. Apply the features selected by GA-MMDL to other classifiers for icd-9 code prediction task.

Table 8
MSE of length of stay prediction.

Algorithm		Score method (Features)		Feature extraction/selection (Features)			All features (141)
		SAPS-II (20)	SOFA (17)	PCA (20)	RFE (20)	GA (20)	
ML	DT	54344.8815 ± 2390.9406	52836.9795 ± 3621.4408	54717.3409 ± 709.7653	48846.0888 ± 3862.0252	**43468.6583 ± 2223.4084**	45166.0388 ± 2853.9258
	Bayes	58715.9520 ± 2012.4531	74876.8679 ± 22148.1398	58784.5393 ± 3291.4266	61219.8744 ± 3515.9883	53172.5127 ± 4091.2336	**52564.0560 ± 2409.1296**
DL	FNN	60096.6133 ± 4174.1588	61843.7995 ± 6065.2225	57233.4701 ± 6296.6451	60358.6628 ± 4690.3527	**53843.4805 ± 3482.8460**	60998.5990 ± 9321.8411
	RNN	55031.8490 ± 3363.6808	54386.3086 ± 1304.0803	–	52148.4479 ± 1121.8866	43776.1263 ± 1633.5601	**42790.0521 ± 1290.5697**
	MMDL	54876.1159 ± 2560.8740	54138.1146 ± 2704.3223	–	51897.9544 ± 1752.6679	44385.0039 ± 3158.0891	**43398.7773 ± 4523.6935**

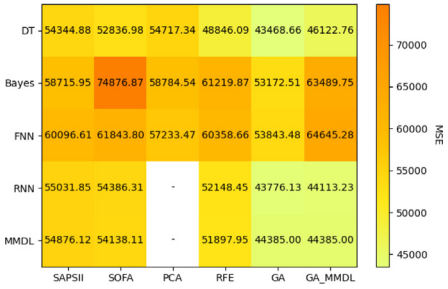


Fig. 3. Apply the features that GA-MMDL selects to other classifiers for the length of the stay prediction task.

Table 9
Whether is statistically significant with significance level 0.05.

Task	Metric	SAPS-II	SOFA	PCA	RFE
Mortality	AUROC	Yes	No	Yes	Yes
	AUPRC	Yes	Yes	Yes	Yes
ICD9	AUROC	No	No	No	No
	AUPRC	No	No	No	No
Length of stay	MSE	Yes	Yes	Yes	Yes

better than other feature selection methods; for mortality and length of stay tasks, the improved performance is statistically significant. (b) Compared with using all features, GA gets similar or even better predictive results with much fewer features, save time and money, which makes it more advantageous to detect and collect clinical data. (c) Other classifiers can also use the features that GA-MMDL selects for the mortality prediction task, achieving good performance.

As part of future work, we plan to make a severity-scoring system based on the features that GA-MMDL selects for the mortality task. This system promises doctors to quickly and accurately assess the severity of a patient's disease with a few simple variables.

References

[1] A.E. Johnson, T.J. Pollard, L. Shen, H.L. Li-Wei, M. Feng, M. Ghassemi, B. Moody, P. Szolovits, L.A. Celi, R.G. Mark, MIMIC-III, a freely accessible critical care database, Sci. Data 3 (1) (2016) 1–9.

[2] J.-R. Le Gall, S. Lemeshow, F. Saulnier, A new simplified acute physiology score (SAPS II) based on a European/North American multicenter study, JAMA 270 (24) (1993) 2957–2963.

[3] J.-L. Vincent, R. Moreno, J. Takala, S. Willatts, A. De Mendonça, H. Bruining, C. Reinhart, P. Suter, L.G. Thijs, The SOFA (Sepsis-related Organ Failure Assessment) score to describe organ dysfunction/failure, Springer-Verlag, 1996.

[4] G. Bhandoria, J.D. Mane, Can surgical apgar score (SAS) predict postoperative complications in patients undergoing gynecologic oncological surgery? Indian J. Surg. Oncol. 11 (1) (2020) 60–65.

[5] R. Pirracchio, M.L. Petersen, M. Carone, M.R. Rigon, S. Chevret, M.J. van der Laan, Mortality prediction in intensive care units with the super ICU learner algorithm (SICULA): a population-based study, Lancet Respir. Med. 3 (1) (2015) 42–52.

[6] S. Purushotham, C. Meng, Z. Che, Y. Liu, Benchmarking deep learning models on large healthcare datasets, J. Biomed. Inform. 83 (2018) 112–134.

[7] H. Harutyunyan, H. Khachatrian, D.C. Kale, G. Ver Steeg, A. Galstyan, Multitask learning and benchmarking with clinical time series data, Sci. Data 6 (1) (2019) 1–18.

[8] J. Lee, Patient-specific predictive modeling using random forests: an observational study for the critically ill, JMIR Med. Inform. 5 (1) (2017) e3.

[9] G.S. Krishnan, S. Kamath, A novel GA-ELM model for patient-specific mortality prediction over large-scale lab event data, Appl. Soft Comput. 80 (2019) 525–533.

[10] A.E. Johnson, A.A. Kramer, G.D. Clifford, A new severity of illness scale using a subset of acute physiology and chronic health evaluation data elements shows comparable predictive accuracy, Crit. Care Med. 41 (7) (2013) 1711–1718.

[11] W.A. Knaus, E.A. Draper, D.P. Wagner, J.E. Zimmerman, APACHE II: a severity of disease classification system, Crit. Care Med. 13 (10) (1985) 818–829.

[12] J.-R. Le Gall, P. Loirat, A. Alperovitch, P. Glaser, C. Granthil, D. Mathieu, P. Mercier, R. Thomas, D. Villers, A simplified acute physiology score for ICU patients, Crit. Care Med. 12 (11) (1984) 975–977.

[13] M. Hoogendoorn, A. El Hassouni, K. Mok, M. Ghassemi, P. Szolovits, Prediction using patient comparison vs. modeling: a case study for mortality prediction, in: 2016 38th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBC, IEEE, 2016, pp. 2464–2467.

[14] J. Calvert, Q. Mao, J.L. Hoffman, M. Jay, T. Desautels, H. Mohamadlou, U. Chettipally, R. Das, Using electronic health record collected clinical variables to predict medical intensive care unit mortality, Ann. Med. Surg. 11 (2016) 52–57.

[15] R. Pirracchio, Mortality prediction in the icu based on mimic-ii results from the super icu learner algorithm (sicula) project, in: Secondary Analysis of Electronic Health Records, Springer, 2016, pp. 295–313.

[16] R. Ahmad, P.A. Bath, Identification of risk factors for 15-year mortality among community-dwelling older people using cox regression and a genetic algorithm, J. Gerontol. (A Biol. Sci. Med. Sci.) 60 (8) (2005) 1052–1058.

[17] L.J. Adams, G. Bello, G.G. Dumancas, Development and application of a genetic algorithm for variable optimization and predictive modeling of five-year mortality using questionnaire data, Bioinform. Biol. Insights 9 (2015) BBI–S29469.

[18] C.-L. Chan, H.-W. Ting, Constructing a novel mortality prediction model with Bayes theorem and genetic algorithm, Expert Syst. Appl. 38 (7) (2011) 7924–7928.
[19] M.C. Engoren, R. Moreno, D.R. Miranda, A genetic algorithm to predict hospital mortality in an ICU population, Crit. Care Med. 27 (12) (1999) A52.
[20] A.E. Johnson, T.J. Pollard, R.G. Mark, Reproducibility in critical care: a mortality prediction case study, in: Machine Learning for Healthcare Conference, 2017, pp. 361–376.
[21] M. Mitchell, An Introduction to Genetic Algorithms, Vol. 1996, MIT press, Cambridge, Massachusetts. London, England, 1996.
[22] K. Pearson, LIII. on lines and planes of closest fit to systems of points in space, Lond. Edinb. Dublin Philos. Mag. J. Sci. 2 (11) (1901) 559–572.

BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100005

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Comparative evaluation of deep learning workloads for leadership-class systems☆

Junqi Yin *, Aristeidis Tsaris, Sajal Dash, Ross Miller, Feiyi Wang, Mallikarjun (Arjun) Shankar

Oak Ridge National Laboratory, United States of America

ARTICLE INFO

Keywords:
CORAL benchmark
Deep learning stack
ROCm

ABSTRACT

Deep learning (DL) workloads and their performance at scale are becoming important factors to consider as we design, develop and deploy next-generation high-performance computing systems. Since DL applications rely heavily on DL frameworks and underlying compute (CPU/GPU) stacks, it is essential to gain a holistic understanding from compute kernels, models, and frameworks of popular DL stacks, and to assess their impact on science-driven, mission-critical applications. At Oak Ridge Leadership Computing Facility (OLCF), we employ a set of micro and macro DL benchmarks established through the Collaboration of Oak Ridge, Argonne, and Livermore (CORAL) to evaluate the AI readiness of our next-generation supercomputers. In this paper, we present our early observations and performance benchmark comparisons between the Nvidia V100 based Summit system with its CUDA stack and an AMD MI100 based testbed system with its ROCm stack. We take a layered perspective on DL benchmarking and point to opportunities for future optimizations in the technologies that we consider.

1. Introduction

The share of deep learning (DL) scientific applications has steadily increased in the allocation portfolio among High-Performance Computing (HPC) centers. In recent years, it has reached a tipping point that the procurement of next-generation HPC infrastructures has to take the performance of the DL stack into consideration. In the case of DOE leadership class platforms, a ***C***ollaboration of ***O***ak ***R***idge, ***A***rgonne, and ***L***ivermore (CORAL) has established a set of benchmarks to gauge the hardware/software competitiveness. For the first time in the CORAL-2 benchmarks [1] suite, DL workloads are included in the evaluation for the acquisition of the systems: Frontier at Oak Ridge, Aurora at Argonne, and El Capitan at Livermore. Ranging from DL kernels to distributed training, the CORAL-2 DL benchmarks consist of micro-benchmarks, such as DeepBench [2], and DL suites including both ResNet50 on ImageNet [3] and application benchmarks such as the cancer distributed learning environment (CANDLE) [4]. Comparing to the industry-led benchmarking effort, MLCommons HPC (also referred to as MLPerf HPC [5]), the CORAL-2 benchmarks focus more on throughput and fundamental building blocks.

Regardless of the increasing complexities of deep neural net (DNN) models, the compute operations essentially boil down to three types of mathematical kernels, i.e., general matrix multiply (GEMM), convolution, and recurrent operation. Considering that distributed training at scale has became a common practice at data centers, the communication operation has to be taken into account as well. The overall performance of DL applications is hence mostly determined by the hardware/software stack for the aforementioned three mathematical and one communication operations. (While I/O is also an important determining factor, the benchmarks we consider here do not face an I/O bottleneck when high-performance node local storage, e.g., SSD, is used for the data and proper pipelining practices are followed.)

Different from simulation codes that traditionally dominate HPC workloads, DL applications rely heavily on the underlying frameworks, e.g., TensorFlow [6] and PyTorch [7], which provide all the building blocks from model components to training and inference supports. On the one hand, this ecosystem lowers the barrier for DL application developers; on the other hand, it requires hardware vendors to provide an optimized DL software stack to support high-level frameworks.

Currently, Nvidia GPUs are the major platforms for DL workloads, and the corresponding software stack, i.e. CUDA [8], cuDNN [9], and

* Corresponding author.
E-mail addresses: yinj@ornl.gov (J. Yin), tsarisa@ornl.gov (A. Tsaris), dashs@ornl.gov (S. Dash), rgmiller@ornl.gov (R. Miller), fwang2@ornl.gov (F. Wang), shankarm@ornl.gov (M. Shankar).

https://doi.org/10.1016/j.tbench.2021.100005
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 11 November 2021

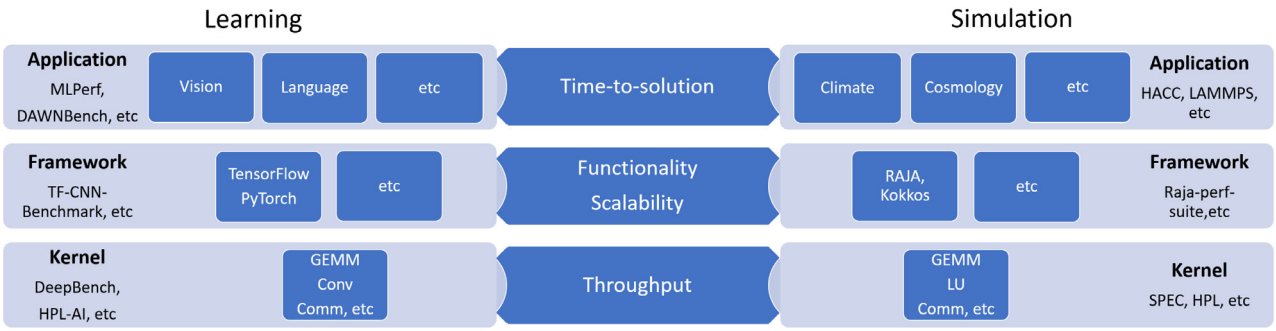


Fig. 1. Comparisons of simulation and learning benchmarks. The overall targets are similar, but facility focus varies due to distinctive development characteristics, e.g. framework plays a much bigger role in learning.

NCCL [10], are the dominant workhorses. As its counterpart, AMD GPUs and the associated ROCm [11], MIopen [12], and RCCL [13] stack, provide a similar ecosystem for DL applications. Though the Nvidia stack is more mature and widely deployed, the AMD stack is entirely open-sourced and progressing, and both platforms are supported by popular DL frameworks such as TensorFlow and PyTorch.

Employing the CORAL-2 DL benchmarks, in this paper we evaluate the performance of an early-access testbed for the upcoming Frontier Exascale system. From kernel primitives, model workloads, to framework and applications, we systematically explore benchmark performance differences between the MI100 based testbed with ROCm stack and the V100 based Summit [14] system with CUDA stack. Our contributions are the following,

- From the perspective of HPC facilities, we propose a layered approach and associated metrics, establish Roofline model, and FOM (Figure of Merits) to evaluate DL workloads from primitive kernels, popular models, to frameworks and applications.
- We provide the first look at an early-access emerging platform based on AMD MI100 GPUs, and show the performance comparisons against a top Nvidia V100 based system in production today.
- We introduce and leverage machine learning (ML) methods (XGBoost [15]) to model the relationship between input parameters and performance outcomes. It lays the groundwork to identify dominant factors to consider for further and future optimizations.
- We show an one-on-one comparison of the resource utilization for our two DL stacks on the same workloads.

The rest of the paper is organized as follows: Section 2 provides general background on differentiating aspects of traditional simulation-based HPC workloads versus emerging DL workloads, as well as an overview of DL benchmarks proposed for the CORAL systems. Section 3 details a layered approach, methodology, and metrics we will use for performance evaluation and comparison. Section 4 presents our results based on the proposed methodology covering compute kernel, model and workloads, frameworks, and applications, which aims to provide an end-to-end perspective on key performance metrics. Section 5 presents our conclusions and discusses opportunities for future work.

2. Background and overview

With the rise of DL applications and specialized hardware, DL benchmarking [16] has attracted a lot of attentions recently. Ranging from application level benchmarks, such as MLPerf, to kernel and model level benchmarks, such as DeepBench and HPL-AI [17], the scope touches almost every aspect of DL. The areas of focus, however, are quite different, as shown in Fig. 1. For application developers, the time-to-solution matters most. But for an emerging field such as DL, where the scientific DL community codes are still maturing in comparison to well-adopted simulation codes (e.g., LAMMPS [18]), understanding the kernel performance is of greater interest.

Table 1
CORAL-2 kernel, model workload, framework, and application benchmarks for learning.

Type	Benchmark	Task	Distributed
Kernel	DeepBench	GEMM	N
		CNN	N
		RNN	N
	N/RCCL-tests	Allreduce	Y
		Allgather	Y
		Reduce	Y
		ReduceScatter	Y
Model Workload	Deep Learning Suite	AlexNet	N
		GoogleNet	N
		OverFeat	N
		VGG	N
		RNN-Net	N
Framework	TF_CNN_Benchmark	ResNet50	Y
Application	CANDLE	P1B1	N
		P3B1	N

At HPC facilities, we make the following observations regarding traditional simulation and DL applications:

1. Unlike simulation applications, most DL applications strongly depend on the frameworks, and are implemented in high-level scripting languages and use pre-compiled framework binaries at run time.
2. The number of DL frameworks are converging to the two most popular ones, i.e., TensorFlow and PyTorch, while the adoption of simulation frameworks (e.g., RAJA, Kokkos) is still at a relative low level.
3. DL frameworks hide most of the complexities in code porting and optimization from developers, since hardware vendors of GPU, TPU, etc., generally upstream the optimized DL stack support to frameworks.

Overall, most DL developers interact mainly with frameworks (e.g., TensorFlow and PyTorch) in Python, and are transparent to underlying compute kernels and platform. This is one of the major distinctions from simulation codes, where the programming framework (e.g., C/C++/Fortran) provides merely basic APIs. In light of the above observations, we focus more on DL primitives and frameworks in facility benchmarking instead of application-level benchmarks. Nevertheless, an end-to-end application benchmark (CANDLE) is included to show the performance of the overall pipeline. A side by side comparison of key components of the DL and the traditional simulation stack is shown in Fig. 1.

CORAL-2 DL Benchmarks In Table 1, we list the benchmarks under study in this work. It covers key DL primitives such as operations for convolution, recurrent neural network (CNN/RNN), and model workloads, frameworks, and applications representative to HPC facilities.

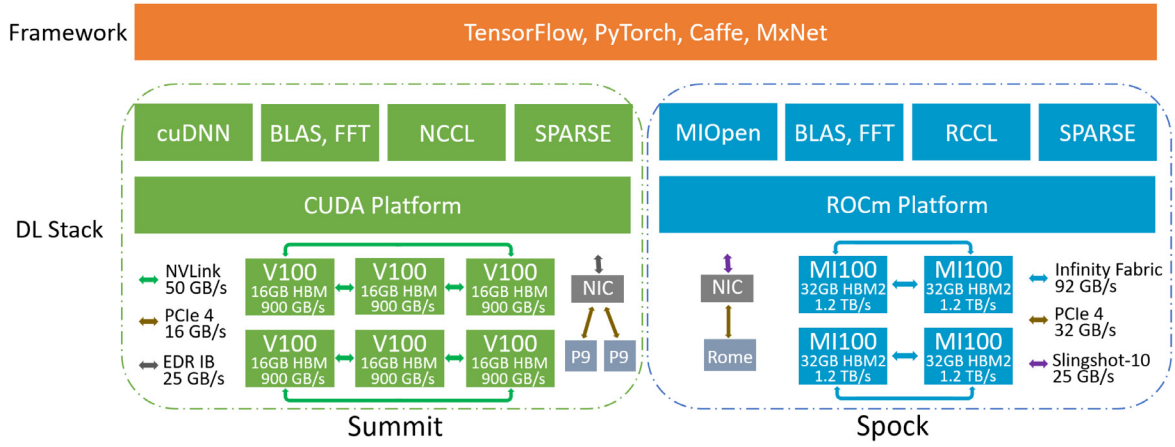


Fig. 2. The node architecture, DL core stack, and supporting framework for Summit and Spock systems.

The kernel benchmarks include DeepBench on a single device and N/RCCL tests for cross device communication. The model workloads consist of CNN models such as AlexNet [19], GoogleNet [20], etc., and an RNN model. The TF_CNN_Benchmarks [21] is used to evaluate TensorFlow framework for data parallel training. The CANDLE application is used to benchmark overall time-to-solution. In all, the scope involves a full spectrum of DL benchmarks corresponding to the application layer down to the foundational kernels as shown in Fig. 1.

DL Stack To evaluate the AMD and Nvidia DL stack, we execute the CORAL-2 benchmarks on both the Summit supercomputer and a testbed system for Frontier called Spock [22]. The node configurations of these two systems are shown in Fig. 2. Each Summit node is equipped with 6 Nvidia Volta GPUs (V100) and 2 IBM Power9 (P9) CPUs. Pairs of 3 V100s are fully connected with NVLink fabrics of 50 GB/s bandwidth, and nodes are then connected via EDR InfiniBand with a capability of 25 GB/s. Spock is an early-access system with an architecture similar to Frontier's but is a generation earlier in accelerator technology (MI100) compared to Frontier (MI200). Each Spock node is equipped with 4 AMD Instinct MI100 GPUs and 1 EPYC 7662 Rome CPU. All 4 MI100s are connected with each other using 92 GB/s Infinity Fabric, and nodes are connected via Slingshot-10. The node local storage are not illustrated because this study focuses on accelerator devices and associated software stack.

For DL frameworks, the support of different accelerator hardware (e.g. GPU, TPU, ARM) requires the corresponding linear algebra software for the devices. As shown in Fig. 2, for Nvidia GPUs, DL primitives of the CNN/RNN etc., are provided via cuDNN on top of the CUDA platform. Depending on the implementations (e.g., CNN can be based on matrix multiplication, Fourier transform, etc.), cuBLAS or cuFFT can be invoked. Similarly, MIOpen is the core DL primitive library for AMD GPUs on top of the ROCm platform, and works with rocBLAS, rocFFT, etc., to support upper level frameworks. In terms of the support for scaling up DL operations, both Nvidia and AMD provide a GPU direct communication library, i.e., NCCL and RCCL, respectively. The following studies are performed with CUDA v10.2, ROCm v4.1, and TensorFlow v2.3.

3. Methodology and metrics

Depending on the category and purpose (See Fig. 1 and Table 1) of the benchmarks, different metrics are utilized. Typically, for throughput benchmarks, floating point operations per second (FLOPS) is used and a similar metric in DL is the processed data samples per second (e.g., images/s [21]). For distributed DL in framework scalability benchmarks, we measure the scaling efficiency in terms of throughput. Application benchmarks usually resort to the end-to-end time-to-solution. To calculate the FLOPS for the GEMM operation, the formula

Fig. 3. The illustration of the key parameters in the GEMM.

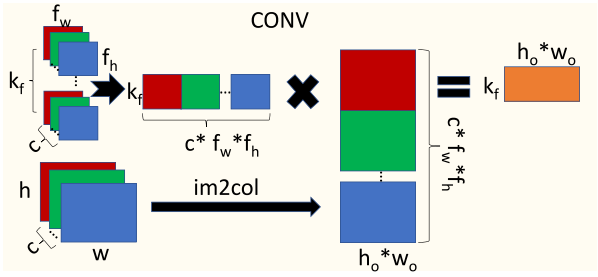


Fig. 4. The illustration of the key parameters in the convolution. It can be converted into matrix multiplication via "im2col" [9].

is defined as:

$$\mathrm{FLOPS}_{\mathrm{GEMM}} \sim 2 \times m \times n \times k/t, \tag{1}$$

where (m, k) and (k, n) are matrix dimensions as shown in Fig. 3, and t is the measured run time.

Since key compute operations in both CNN and RNN can be broken into matrix multiplications (See Figs. 4 and 5), the FLOPS formulas follow a similar scheme. (There are other types of implementations for convolution, e.g., Winograd and FFT [23] - for the simplicity of discussion, we focus on the GEMM based implementation.)

For the 2D Convolution operation (GEMM based) on input dimension of height h, width w, and channel c, FLOPS is calculated via,

$$\mathrm{FLOPS}_{\mathrm{Conv2D}} \sim 2 \times \left(h_o \times w_o\right) \times k_f \times \left(c \times f_w \times f_h\right)/t \tag{2}$$

$$h_o = \frac{\left(h + 2 \times \mathrm{pad}_h - f_h\right)}{\mathrm{stride}_h} + 1 \tag{3}$$

$$w_o = \frac{\left(w + 2 \times \mathrm{pad}_w - f_w\right)}{\mathrm{stride}_w} + 1 \tag{4}$$

where k_f is the number of filters each of dimension (f_h, f_w) with padding (`pad`) and `stride` specified in h and w dimension, respectively, and h_o and w_o [9] are the effective height and width after applying a filter.

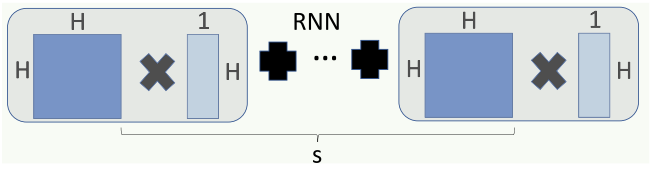


Fig. 5. The illustration of the key parameters in the RNN operation. The basic building block is also matrix multiplication.

Similarly, for the RNN operation (See Fig. 5), the FLOPS calculation follows,

$$\text{FLOPS}_{\text{RNN}} \sim 2 \times H \times H \times s/t, \tag{5}$$

Where H and s are the hidden size and time steps, respectively. For the data input with N samples (i.e., batch size N), the FLOPS for the operation will be simply multiplied by a factor of N.

Roofline Model In addition to FLOPS, another important metric to gauge the compute and memory performance is the so called Roofline model, which can visually demonstrate the bottleneck of the benchmark and hardware, i.e., whether it is compute or memory bound. To that end, the arithmetic intensity I, i.e., floating operations per memory load, needs to be calculated. For single-precision GEMM, this is given by,

$$I = \frac{FLOPs}{4(m * k + k * n + 2n * m)} \tag{6}$$

assuming the ideal data re-use of the two input matrices of element size $m * k$ and $k * n$. The Roofline model is then obtained by plotting the performance (FLOPS) versus the arithmetic intensity (FLOPs/bytes).

Figure of Merit Regardless of the types of the benchmarks, a relative metric, i.e., figure of merit (FOM), is often used in procurement. In this study, it is defined as follow,

$$\text{FOM} = \prod_{i}^{N} \left(\frac{\text{metric}^i}{\text{metric}^i_b} \right)^{1/N} \tag{7}$$

where the $metric^i_b$ is for the performance metric of ith task on the baseline system. To account for a balanced performance, the geometric mean is taken over either N sub-tasks within the benchmark or across N benchmarks. The metric for each sub-task or benchmark can be aforementioned FLOPS, images/s, scaling efficiency, or time-to-solution.

ML modeling The performance of DL kernels depends on many factors including algorithm, implementation, input shape, etc. It is hard to predict kernel runtime especially when there are multiple algorithms for the same operation (e.g., convolution) and built-in heuristics (e.g., in cuDNN, FFT-based convolution is used when f_h or f_w is bigger than 5) to select the algorithm at runtime. For the closed-source library such as cuDNN, it becomes even more challenging.

To identify the important parameters on kernel performance, we use XGBoost [15] to model the relationship between input parameters and performance outcome, and then rank the parameter based on its feature importance. Because the features are well-structured (in contrast to text and image) and limited in size, the traditional ML method such as XGBoost is well suited for the task.

Resource Utilization Another important way of understanding the performance of deep learning applications is by tracking resource utilization. This is typically used to find bottlenecks of the workload and identify operations that need optimization. In this work we use the `nvidia-smi` for the V100 GPUs on Summit and the `rocm-smi` for the MI100 GPUs on Spock to monitor the memory used and the GPU utilization for the framework and application benchmarks. Specifically the `memory.used` and the `utilization.gpu` flags were used for the `nvidia-smi`, and the `showuse` and `showmemuse` for the `rocm-smi`.

Even though those low level tools may not have been configured the same way, it is important to show early their default behavior on deep learning workloads, so that further optimization strategies can be made as more realistic HPC/DL workloads are applied. For example one noticeable difference from the documentation provided for those tools is that `nvidia-smi` sample period may be between 1 s and 1/6 s depending on the product, where `rocm-smi` samples every millisecond. Also higher level custom profilers usually use directly those low level tools, and by showing those results we hope to give a better understanding for the future developers on the current status.

The strategy is to request data from those tools on each batch/epoch iteration on the training stage, rather than monitoring the benchmark application itself. This way we can better focus on comparison between training steps, and eliminate differences between job schedulers or initial environment/system conditions between Spock and Summit, which might change over time. In all cases the flag `TF_FORCE_GPU_ALLOW_GROWTH` was used as true for better comparison between the two.

4. Evaluation results

Following the approach described in Section 3, we perform systematic evaluations of the DL stack on Summit and Spock system in terms of kernel, model, framework, and application benchmarks.

4.1. Kernel benchmarks

As previously discussed, we focus on the performance characteristics of kernel and model workloads (listed in Table 1) because they serve as common denominators across DL applications. For example, in DeepBench, the inputs for GEMM, CNN, and RNN kernels are selected from representative real DL workloads.

For kernel benchmarks, we employ DeepBench and N/RCCL tests for computing and communication primitives, respectively. These kernels usually account for a single tensor/layer operation of a neural network. Moving one level up, the workload benchmarks put together the kernel operations for popular DL models. Considering DL frameworks operate in single precision by default, we evaluate the kernels and model workloads in the same single precision.

Compute Kernels In Fig. 6, the generated FLOPS (See Eqs. (1), (3)) of a single device on Summit and Spock are plotted for a list of GEMM, CNN, and RNN operations, respectively. For GEMM, MI100 performs better for more computationally expensive operations, while for less expensive ones, the performance differences between MI100 and V100 are generally small. Of the predefined inputs (see examples in Table 4) in DeepBench, the best performance of 17.7 (77% of peak) and 14.7 (93% of peak) TFLOPS are achieved for MI100 and V100, respectively. The corresponding input parameters are annotated in Fig. 6 (GEMM), i.e., $(m = 6144, n = 48000, k = 2048)$ for MI100 and $(m = 4096, n = 7000, k = 4096)$ for V100. For ill-shaped inputs, e.g., $(m = 512, n = 8, k = 500000)$, the kernels become memory bound. For CNN, specifically the GEMM based (so called "im2col" implementation, see Fig. 4) 2D convolution, the MI100 outperforms V100 in most of cases, and a similar 79% of peak is obtained while 50% for V100 for the best case. On the other hand, V100 seems to perform better for RNN kernels, especially for larger inputs, but because RNN is more memory intensive, both devices are far below the peak compute performance.

To provide an overall comparison, in Fig. 7, we plot the FOM for all three kernel types combining performances for various inputs. With Summit as the baseline (FOM=1), Spock performs similarly for GEMM and RNN, and shows an edge over CNN. Considering the run time of a model usually dominated by the most expensive layer, we also calculate the FOM for the top 10 most expensive kernel operations. Spock shows a 20% and 10% advantage for GEMM and CNN, respectively.

According to the Roofline model, as shown in Fig. 8, the boundary for the memory and compute capability is 17.4 and 19.2 for V100

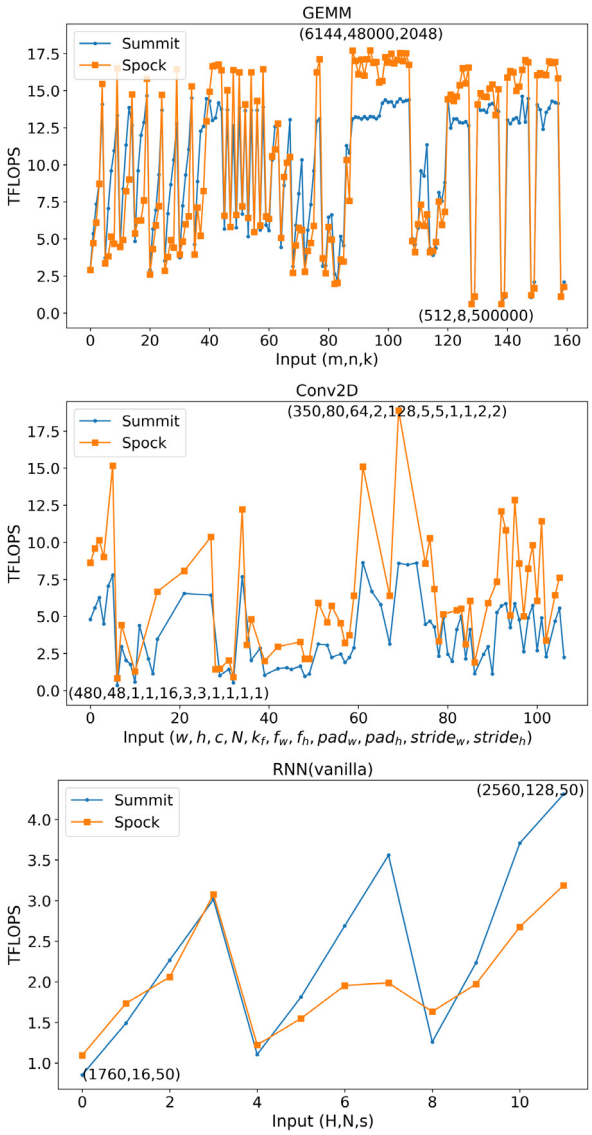


Fig. 6. The FLOPS of the GEMM, CNN, and RNN primitive operations for identified representative inputs in DeepBench. The parameters are annotated for the best and worst performance, respectively.

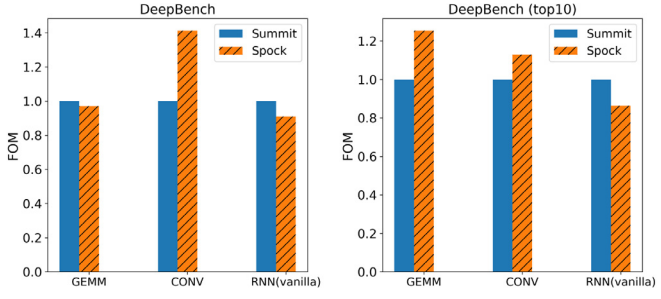


Fig. 7. The aggregated FOM of DeepBench benchmark for all and top 10 most expensive tasks, respectively.

on Summit and MI100 on Spock, respectively. In both regions (left and right of the dashed boundary line), data points for Summit is closer to the upper limit, i.e. maximum bandwidth and theoretical peak

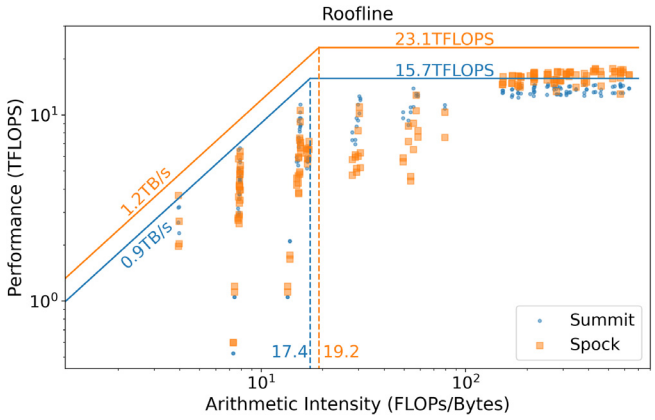


Fig. 8. The Roofline model for DeepBench GEMM benchmark (FP32).

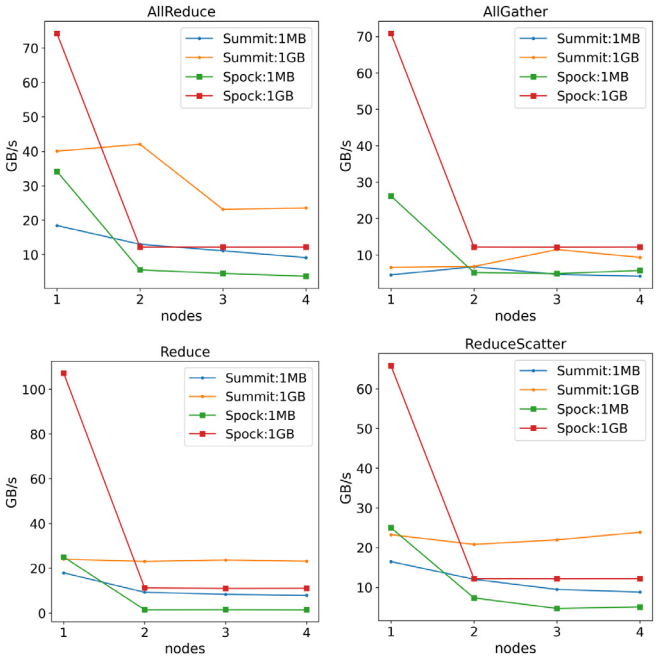


Fig. 9. The bandwidth of typical DL communication kernels up to four nodes on Summit and Spock.

(annotated in the plot) than those of Spock, indicating that there are still room for optimization in ROCm DL stack.

Communication Kernels Given that distributed training has became common practice to manage ever-growing data and model sizes, the communication kernels play increasingly important roles. For the popular data parallel training (each device has a model replica working on different data batch, and the gradient information is exchanged periodically), `allreduce` is the dominant communication pattern that is executed each (synchronized) or a few (stale or asynchronized) batch steps. Depending on the implementation, the `allreduce` can be realized via a single API or a combination of `allgather` and `reducescatter`, or `reduce` and `broadcast`. The performance depends on device communication libraries (e.g., N/RCCL) and the specific network topology of the platform.

In Fig. 9, we plot the bus bandwidth (GB/s) up to four nodes on Summit and Spock for four commonly used communication APIs in N/RCCL. The message size ranges from small (1 MB) to large (1 GB), covering the gradient size for popular DL models (e.g., 100 MB for ResNet50). For the intra-node communication, Spock shows an up to 3x lead across the board, thanks to high-bandwidth Infinity Fabric (see

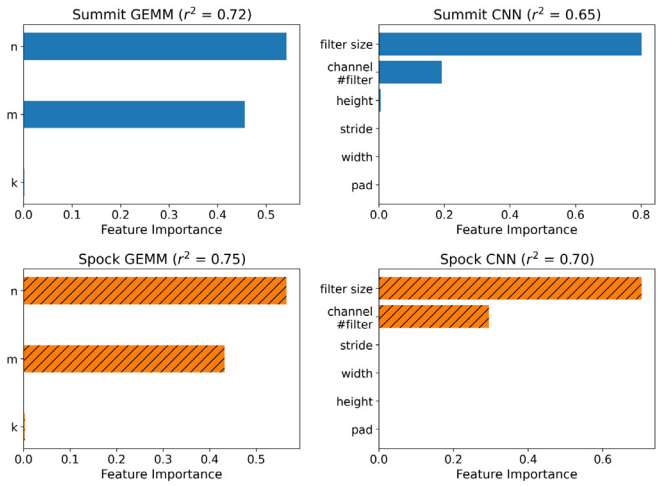


Fig. 10. The feature importance of XGBoost modeling of GEMM and CNN benchmarks in DeepBench. The ratio of explained variance (r^2) is listed.

Fig. 2). In the case of inter-node, Spock seems to perform better for `allgather` but lags behind in others, which is due to the slower PCIe connection comparing to Summit's NvLink between CPU and GPU. It indicates that a combination of inter-node `allgather` and intra-node `reducescatter` is the best way to realize the gradient `allreduce` on this particular system. Note that the network topology of the Frontier system will be significantly different from that of Spock.

Machine Learning on DL To further understand how the input parameters affect the kernel performance, we use the ML method to model the performance data on DL kernels. Because both NCCL and RCCL are open sourced and communication optimization typically relies on the framework-level libraries (e.g., torch.DDP, TF.distribute, Horovod) to overlap communication with computation, we mainly focus on the compute kernels. Because the input features are well structured, XGBoost method is used to model the relationship between input parameters and kernel performance. As shown in Fig. 10, the feature importance for GEMM and CNN are quite similar on both Summit and Spock, respectively, i.e., for GEMM, since in DL operations it is often between a squared matrix and an ill-shaped one (see example input parameters in Table 4), the run time can be well predicted by the shape of resulted matrix; for CNN (GEMM based), the filter size and input channel (often strongly correlated with number of filters) are dominant factors for run time. We can thus hypothesize that the implementations of GEMM based convolution are similar in cuDNN and MIOpen.

4.2. Model benchmarks

Putting together the tensor/layer operations, workload benchmarks on popular DL models show the combined performance of typical DL workloads. Because accelerators are of primary interest here, we focus on compute workloads and isolate them from the noise from I/O and communication.

In Table 2, we list the operation breakdown for the candidate CNN models. The model size ranges from 61M (AlexNet [19]) to 146M (OverFeat [24]) parameters with the number of convolution layers from 5 to 57. Comparing the earlier AlexNet to VGG [25], and then to ResNet50 and GoogleNet, the trend in DL modeling favors deeper models with relatively thin layers.

The number of parameters in a model is counted by

1. for a convolution layer with input channel c and k filters, each of size (f_w, f_h), the number of parameters is $c * k * f_w * f_h + k$.
2. for a recurrent layer with input size n and H hidden units, the number of parameters is the same as a feed-forward neural network, i.e. $H * (H + n) + H$.

Table 2
CORAL-2 CNN model workloads.

Model	# conv layers	Filter size	# filters	# weights	# MACs	% conv
AlexNet	5	3,5,11	96–384	61M	724M	92
OverFeat	5	3,5,11	96–1024	146M	2.8B	95
VGG	13	3	64–512	138M	15.5B	99
ResNet50	53	1,3,7	64–2048	25.5M	3.9B	99
GoogleNet	57	1,3,5,7	16–384	7M	1.4B	~100

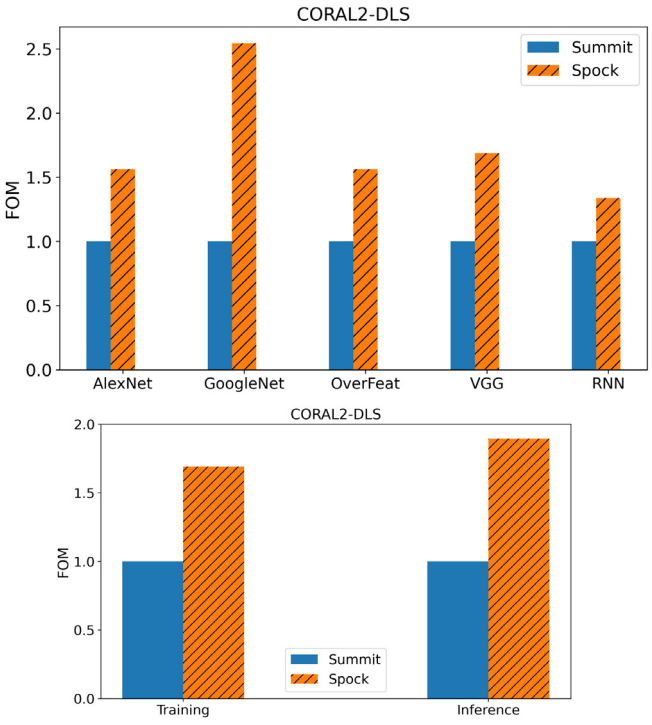


Fig. 11. The FOM for individual model workload and combined training and inference benchmarks in CORAL-2 DLS.

The corresponding multiply and accumulation (MAC) operations follow similar FLOPS counts discussed in Section 3. From Table 2, VGG is the most computationally expensive model with 15.5B MAC operations.

In Fig. 11, we plot the performance comparisons for DL model workloads. The FOM numbers are calculated from the processed samples/s with Summit being the baseline. Spock shows better performance across the board with the best FOM (~2.5x) for GoogleNet. To obtain an overall view for DL training and inference, we further break down the run time for forward pass (inference) and forward–backward pass (training), and calculate the FOM across all workload tasks. It is shown (See Fig. 11) that a speedup of 1.7x and 1.9x is achieved on Spock for training and inference, respectively. Note this is with the CORAL-2 deep learning suite (DLS) baseline implementation (TensorFlow, single precision).

4.3. Framework benchmarks

Although model workloads, as discussed in Section 4.1, already exercise the framework on a single device, there are many other aspects of the framework that require further examination. To this end, we use the `TF_CNN_Benchmark` to perform the distributed training on ResNet50, which is required by CORAL-2 DLS.

Functionality In terms of the performance functionalities, both TensorFlow and PyTorch support automatic mixed precision, runtime compilation e.g., accelerated linear algebra (XLA), etc. Frameworks operate in single precision by default because the mixed precision requires special

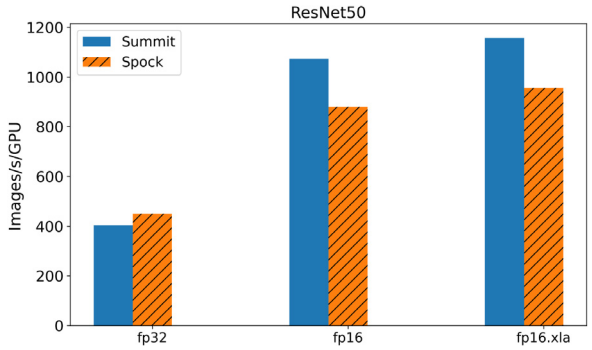


Fig. 12. The training images/s per GPU in FP32, FP16, and FP16 with XLA for `TF_CNN_Benchmarks`.

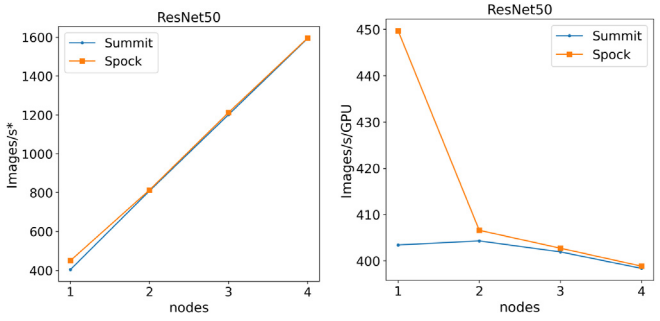


Fig. 13. The scaling of distributed training throughput for `TF_CNN_Benchmarks`. Images/s* is normalized to the number of GPUs per node for Summit (6) and Spock (4), respectively.

care, and by automating the mixed precision support it enables easier access to the full hardware capability. TensorFlow XLA can further accelerate the execution by generating optimized tensor operations for specific model rather than using the pre-built binary.

In Fig. 12, we plot the single device training performance for ResNet50 (batch size 128) with different accelerations. Consistent with Fig. 11, Spock has an edge at single precision, but lags behind in half precision. The speedup due to XLA though, are more or less the same.

Scaling Another important aspect of the framework is its scalability. Here we use a popular third-party distributed training library, i.e., Horovod, because it supports multiple frameworks including TensorFlow and PyTorch, and is highly optimized for HPC platforms. As shown in Fig. 13, the training images/s per GPU gradually decreases on Spock with a scaling efficiency $\sim 89\%$ up to four nodes, while Summit scales almost perfectly (scaling efficiency $\sim 99\%$). Given N/RCCL are used as communication backends, the results are consistent with Fig. 9.

Resource Utilization As described in Section 3, we use NVIDIA and ROCm tools to measure the resource utilization for every training step iteration between Summit and Spock. Fig. 14 shows the memory used and the GPU utilization for the ResNet50 benchmark (batch size 128) in single precision. The memory used for V100s seems to be constant across training steps, while for the MI100s it appears to vary across steps. This behavior more likely reflects the different sample frequencies as described in Section 3. The GPU utilization for Spock seems to be able to keep up more with each iteration compare to Summit, and that might reflect the fact that we get more number of images per second for single precision on Summit, as shown in Fig. 12.

4.4. Application benchmarks

Our goal at facilities is to enable leadership-scale scientific discoveries, hence the performance of scientific application is of ultimate interest. In CORAL-2, there are two sub-tasks enlisted from CANDLE

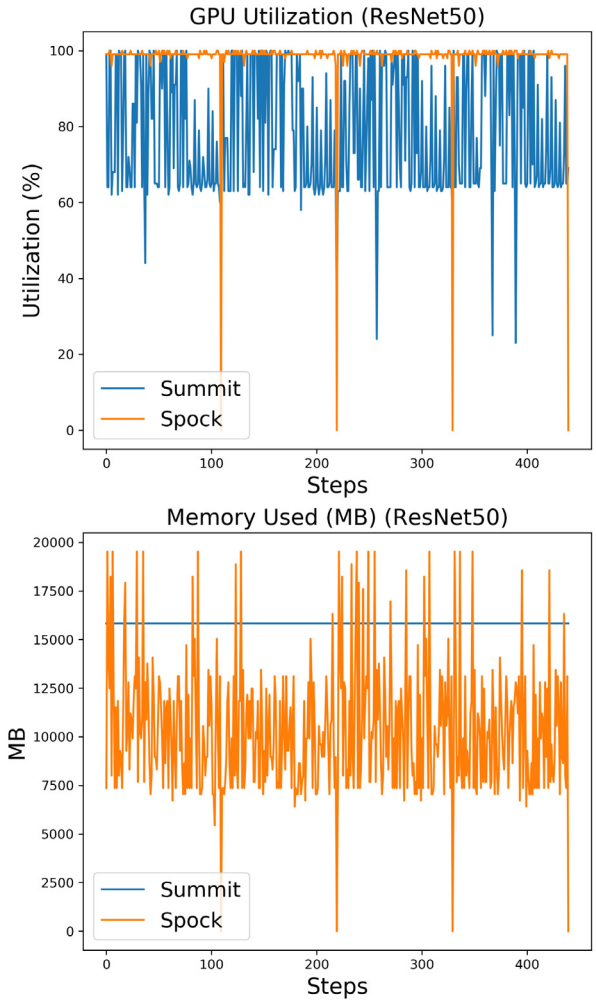


Fig. 14. Timeline plots for memory used and GPU utilization for `TF_CNN_Benchmarks`.

Table 3
Specification of 2 sub-tasks in CANDLE benchmark.

Task	Sample size	Model	Layer type	# layers	Hidden layer size	# weights
P1B1	4000	Autoencoder	Dense	6	2, 600, 1000	183M
P3B1	3000	Multi-task MLP	Dense	11	400, 1200	10M

benchmark (see Table 3): P1B1 is a regression task that use autoencoder to compress the gene expression; P3B1 is a classification task that use multi-task multilayer perceptron (MLP) for data extraction from clinic reports. Both models are based on fully connected dense layers, so the compute is dominated by GEMM kernel operations. The input data size is rather small (less or around 1 GB), and the impact of I/O is negligible (no noticeable performance differences in running with or without local storage).

In Fig. 15, the FOMs of time-to-solution are plotted for both tasks in CANDLE. Different random seeds are used to obtain the run time to the target reconstruction mean square error (P1B1) and classification accuracy (P3B1), but due to using the same baseline implementation and hyperparameter selection, it requires the same number of training steps to converge. Spock performs better in P3B1 task while Summit shows advantage in the other, mainly because of the performance differences in different shapes of matrices in GEMM (See Fig. 6).

Resource Utilization Fig. 16 shows the memory used and the GPU utilization for P1B1. The GPU utilization is higher for Spock,

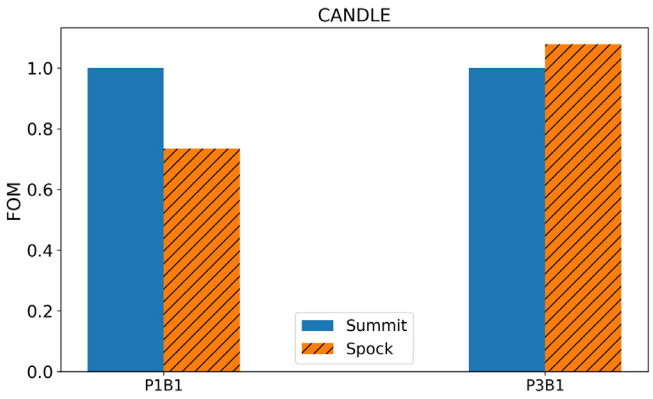


Fig. 15. The FOM of time-to-solution for 2 tasks in the CANDLE benchmarks.

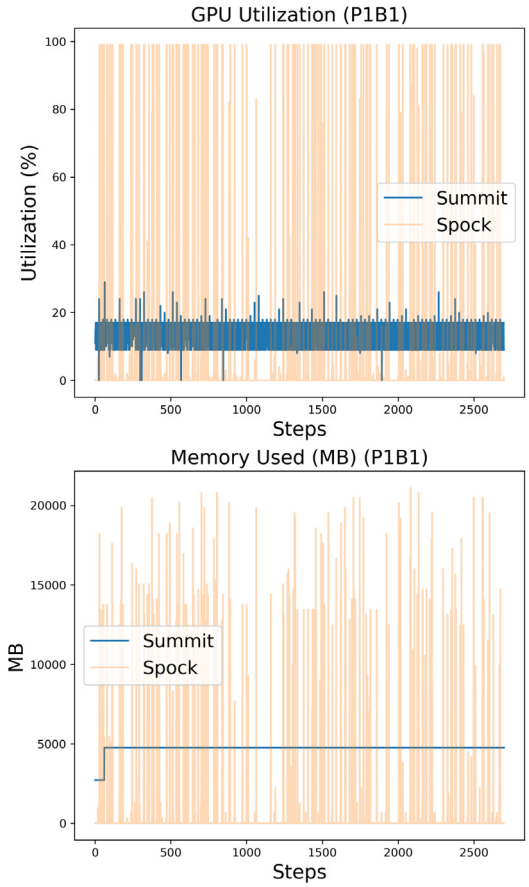


Fig. 16. Timeline plots for memory used and GPU utilization for `CANDLE-P1B1` benchmark.

compare to Summit and as expected for the V100s the `nvidia-smi` (as mentioned in Section 3) samples less frequently than `rocm-smi`. The memory used is higher for Spock compare to Summit. Compare to ResNet50 results, as shown in Fig. 14, the memory used is higher for Summit. We note that the model architecture is very different (dense vs convolution layers), and also that this benchmark uses a Keras implementation.

Fig. 17 shows the memory used and the GPU utilization for P3B1. This benchmark implementation is a mix of Keras and TF. It shows higher but more sparse GPU utilization for Spock, compare to Summit. Also the memory usage appears to be higher on Summit compare to Spock. If the `nvidia-smi` and `rocm-smi` are one-to-one comparable we could argue that larger models, or larger input size vectors can fit

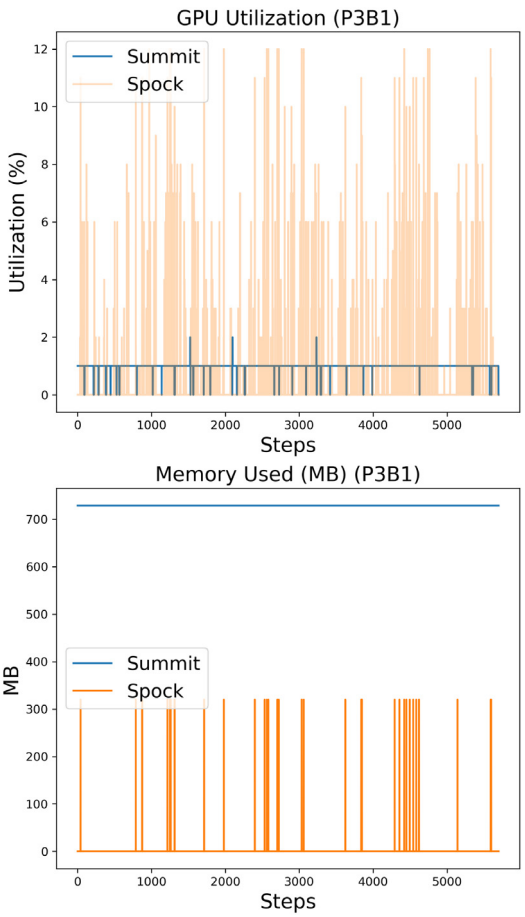


Fig. 17. Timeline plots for memory used and GPU utilization for `CANDLE-P3B1` benchmark.

on the MI100s with this implementation, but because of the differences in the tools, further investigation is needed.

5. Conclusion

We have presented a layered methodology and metrics to benchmark DL workloads at scale, involving kernels, models, frameworks, and applications. From the perspective of HPC facilities, we argue that understanding kernel and model level performance, and framework level scalability are more important than application FLOP counts given current scientific DL use cases and patterns. Using the CORAL-2 DL benchmarks, we evaluated the performance of Spock, an early-access testbed system for Frontier. Compared to the V100 based Summit system with CUDA DL stack, the MI100 based Spock with ROCm DL stack shows an edge in single precision performance for most kernel and model benchmarking tasks. However, there is currently a gap in its half precision performance, specifically for TensorFlow. Roofline modeling also indicates rooms for improvement in the ROCm stack, which is still maturing.

We also explored and demonstrated using machine learning an approach to model the relationship between input parameters and benchmark performance outcomes. And through a one-on-one comparison of the resource utilization for the two DL stacks on the same DL workloads, we are able to comment on the sources of performance differences. Although these two ways of gaining insight into performance comparisons are not conclusive in deducing underlying implementations or bottlenecks, our data does shed light on the direction for future optimizations in the DL stacks.

Table 4
The kernel input parameters in DeepBench [2]. RNN (vanilla) input parameters and first 20 of GEMM and Conv2D input parameters are listed.

Index	GEMM			Conv2D										RNN		
	m	n	k	w	h	c	N	k_f	f_w	f_h	pad_w	pad_h	$stride_w$	$stride_h$	H	N	s
0	1760	16	1760	700	161	1	4	32	20	5	0	0	2	2	1760	16	50
1	1760	32	1760	700	161	1	8	32	20	5	0	0	2	2	1760	32	50
2	1760	64	1760	700	161	1	16	32	20	5	0	0	2	2	1760	64	50
3	1760	128	1760	700	161	1	32	32	20	5	0	0	2	2	1760	128	50
4	1760	7000	1760	341	79	32	4	32	10	5	0	0	2	2	2048	16	50
5	2048	16	2048	341	79	32	8	32	10	5	0	0	2	2	2048	32	50
6	2048	32	2048	341	79	32	16	32	10	5	0	0	2	2	2048	64	50
7	2048	64	2048	341	79	32	32	32	10	5	0	0	2	2	2048	128	50
8	2048	128	2048	480	48	1	16	16	3	3	1	1	1	1	2560	16	50
9	2048	7000	2048	240	24	16	16	32	3	3	1	1	1	1	2560	32	50
10	2560	16	2560	120	12	32	16	64	3	3	1	1	1	1			
11	2560	32	2560	60	6	64	16	128	3	3	1	1	1	1			
12	2560	64	2560	108	108	3	8	64	3	3	1	1	2	2			
13	2560	128	2560	54	54	64	8	64	3	3	1	1	1	1			
14	2560	7000	2560	27	27	128	8	128	3	3	1	1	1	1			
15	4096	16	4096	14	14	128	8	256	3	3	1	1	1	1			
16	4096	32	4096	7	7	256	8	512	3	3	1	1	1	1			
17	4096	64	4096	224	224	3	8	64	3	3	1	1	1	1			
18	4096	128	4096	112	112	64	8	128	3	3	1	1	1	1			
19	4096	7000	4096	56	56	128	8	256	3	3	1	1	1	1			

Finally, we do note that Spock is a testbed early access system. Our benchmarking results and comparisons are most useful to concretely present our systematic approach to DL benchmarking. The kernels and frameworks are maturing and will continue to evolve (particularly for the ROCm ecosystem) and, therefore, specific observations reported in this paper are likely to change even if it does not affect the overall methodology that we have presented.

Acknowledgments

This research was sponsored by and used resources of the Oak Ridge Leadership Computing Facility (OLCF), which is a DOE Office of Science User Facility at the Oak Ridge National Laboratory supported by the U.S. Department of Energy under Contract No. DE-AC05-00OR22725.

Appendix. Kernel parameters

In Table 4, we list the input parameters for GEMM, Conv2D, RNN (vanilla) kernels defined in DeepBench [2].

References

[1] ORNL, ANL, LLNL, CORAL-2 benchmarks, 2017, https://asc.llnl.gov/coral-2-benchmarks. (Accessed 30 July 2021).
[2] Baidu, DeepBench, 2016, https://github.com/baidu-research/DeepBench. (Accessed 30 July 2021).
[3] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, 2015, arXiv preprint arXiv:1512.03385.
[4] CANDLE, Cancer distributed learning environment, 2018, http://candle.cels.anl.gov. (Accessed 30 July 2021).
[5] P. Mattson, C. Cheng, G. Diamos, C. Coleman, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, D. Brooks, D. Chen, D. Dutta, U. Gupta, K. Hazelwood, A. Hock, X. Huang, D. Kang, D. Kanter, N. Kumar, J. Liao, D. Narayanan, T. Oguntebi, G. Pekhimenko, L. Pentecost, V. Janapa Reddi, T. Robie, T. St John, C.-J. Wu, L. Xu, C. Young, M. Zaharia, Mlperf training benchmark, in: I. Dhillon, D. Papailiopoulos, V. Sze (Eds.), Proceedings of Machine Learning and Systems, Vol. 2, 2020, pp. 336–349, URL https://proceedings.mlsys.org/paper/2020/file/02522a2b2726fb0a03bb19f2d8d9524d-Paper.pdf.
[6] M. Abadi, et al., TensorFlow: Large-scale machine learning on heterogeneous systems, 2015, URL https://www.tensorflow.org/. Software Available from: tensorflow.org.
[7] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, S. Chintala, Pytorch: An imperative style, high-performance deep learning library, in: H. Wallach, H. Larochelle, A. Beygelzimer, F. d Alché-Buc, E. Fox, R. Garnett (Eds.), Advances in Neural Information Processing Systems, Vol. 32, Curran Associates, Inc. 2019, pp. 8024–8035, URL http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.
[8] J. Nickolls, I. Buck, M. Garland, K. Skadron, Scalable parallel programming with CUDA: Is CUDA the parallel programming model that application developers have been waiting for? Queue 6 (2) (2008) 40–53, http://dx.doi.org/10.1145/1365490.1365500.
[9] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, E. Shelhamer, Cudnn: Efficient primitives for deep learning, CoRR (2014) arXiv:1410.0759.
[10] Nvidia, Nvidia communication collectives library (NCCL), 2019, https://github.com/NVIDIA/nccl. (Accessed 30 July 2021).
[11] AMD, ROCM, 2016, https://rocmdocs.amd.com. (Accessed 30 July 2021).
[12] J. Khan, P. Fultz, A. Tamazov, D. Lowell, C. Liu, M. Melesse, M. Nandhimandalam, K. Nasyrov, I. Perminov, T. Shah, V. Filippov, J. Zhang, J. Zhou, B. Natarajan, M. Daga, Miopen: An open source library for deep learning primitives, 2019, arXiv arXiv:1910.00078.
[13] AMD, ROCM communication collectives library (RCCL), 2020, https://github.com/ROCmSoftwarePlatform/rccl. (Accessed 30 July 2021).
[14] S.S. Vazhkudai, B.R. de Supinski, A.S. Bland, A. Geist, J. Sexton, J. Kahle, C.J. Zimmer, S. Atchley, S. Oral, D.E. Maxwell, V.G.V. Larrea, A. Bertsch, R. Goldstone, W. Joubert, C. Chambreau, D. Appelhans, R. Blackmore, B. Casses, G. Chochia, G. Davison, M.A. Ezell, T. Gooding, E. Gonsiorowski, L. Grinberg, B. Hanson, B. Hartner, I. Karlin, M.L. Leininger, D. Leverman, C. Marroquin, A. Moody, M. Ohmacht, R. Pankajakshan, F. Pizzano, J.H. Rogers, B. Rosenburg, D. Schmidt, M. Shankar, F. Wang, P. Watson, B. Walkup, L.D. Weems, J. Yin, The design, deployment, and evaluation of the CORAL pre-exascale systems, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, in: SC '18, IEEE Press, 2018, http://dx.doi.org/10.1109/SC.2018.00055.
[15] T. Chen, C. Guestrin, XGBoost: A Scalable tree boosting system, in: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, in: KDD '16, ACM, New York, NY, USA, 2016, pp. 785–794, http://dx.doi.org/10.1145/2939672.2939785.
[16] J. Yin, S. Gahlot, N. Laanait, K. Maheshwari, J. Morrison, S. Dash, M. Shankar, Strategies to deploy and scale deep learning on the summit supercomputer, in: 2019 IEEE/ACM Third Workshop on Deep Learning on Supercomputers, DLS, 2019, pp. 84–94, http://dx.doi.org/10.1109/DLS49591.2019.00016.
[17] ICL, HPL-AI, 2019, https://icl.bitbucket.io/hpl-ai/. (Accessed 30 July 2021).
[18] S. Plimpton, Fast parallel algorithms for short-range molecular dynamics, J. Comput. Phys. 117 (1) (1995) 1–19, http://dx.doi.org/10.1006/jcph.1995.1039, URL https://www.sciencedirect.com/science/article/pii/S002199918571039X.
[19] A. Krizhevsky, I. Sutskever, G.E. Hinton, Imagenet classification with deep convolutional neural networks, in: Proceedings of the 25th International Conference on Neural Information Processing Systems, Vol. 1, in: NIPS'12, Curran Associates Inc. Red Hook, NY, USA, 2012, pp. 1097–1105.
[20] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, Going deeper with convolutions, in: 2015 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, 2015, pp. 1–9, http://dx.doi.org/10.1109/CVPR.2015.7298594.
[21] TensorFlow, Tf_cnn_benchmarks: High performance benchmarks, 2019, https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks. (Accessed 30 July 2021).
[22] OLCF, Spock, 2021, https://docs.olcf.ornl.gov/systems/spock_quick_start_guide.html. (Accessed 30 July 2021).

[23] A. Lavin, S. Gray, Fast algorithms for convolutional neural networks, in: 2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR, IEEE Computer Society, Los Alamitos, CA, USA, 2016, pp. 4013–4021, http://dx.doi.org/10.1109/CVPR.2016.435.

[24] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, Y. LeCun, Overfeat: Integrated recognition, localization and detection using convolutional networks, 2013, URL arxiv:1312.6229.

[25] K. Simonyan, A. Zisserman, Very deep convolutional networks for large-scale image recognition, CoRR (2014) arXiv:1409.1556.

BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100006

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Benchmarking for Observability: The Case of Diagnosing Storage Failures

Duo Zhang, Mai Zheng *

Department of Electrical and Computer Engineering, Iowa State University, Ames, IA, USA

ARTICLE INFO

Keywords:
Storage systems
Failure diagnosis
Observability
Reproducibility
Reliability
Robustness
Benchmarking
Debugging
Tracing
Record and replay

ABSTRACT

Diagnosing storage system failures is challenging even for professionals. One recent example is the "When Solid State Drives Are Not That Solid" incident occurred at Algolia data center, where Samsung SSDs were mistakenly blamed for failures caused by a Linux kernel bug. With the system complexity keeps increasing, diagnosing failures will likely become more difficult.

To better understand real-world failures and the potential limitations of state-of-the-art tools, we first conduct an empirical study on 277 user-reported storage failures in this paper. We characterize the issues along multiple dimensions (e.g., time to resolve, kernel components involved), which provides a quantitative measurement of the challenge in practice. Moreover, we analyze a set of the storage issues in depth and derive a benchmark suite called $BugBench^k$. The benchmark suite includes the necessary workloads and software environments to reproduce 9 storage failures, covers 4 different file systems and the block I/O layer of the storage stack, and enables realistic evaluation of diverse kernel-level tools for debugging.

To demonstrate the usage, we apply $BugBench^k$ to study two representative tools for debugging. We focus on measuring the observations that the tools enable developers to make (i.e., observability), and derive concrete metrics to measure the observability qualitatively and quantitatively. Our measurement demonstrates the different design tradeoffs in terms of debugging information and overhead. More importantly, we observe that both tools may behave abnormally when applied to diagnose a few tricky cases. Also, we find that neither tool can provide low-level information on how the persistent storage states are changed, which is essential for understanding storage failures. To address the limitation, we develop lightweight extensions to enable such functionality in both tools. We hope that $BugBench^k$ and the enabled measurements will inspire follow-up research in benchmarking and tool support and help address the challenge of failure diagnosis in general.

1. Introduction

The storage stack in the Linux kernel is witnessing a sea-change driven by the advances in non-volatile memory (NVM) technologies [1–13]. For example, the SCSI subsystem and the Ext4 file system, which have been optimized for hard disk drives (HDDs) for decades, are adding multi-queue support [14–16] and DAX support [17] for flash-based solid state drives (SSDs) and persistent memories (PMs), respectively. While such modifications may offer higher performance in general, the implications on system reliability is much less measured or understood.

One real-world example is the "When Solid-State Drives Are Not That Solid" incident occurred in Algolia data center [18], where a random subset of SSD-based servers crashed and corrupted files for unknown reasons. The developers "spent a big portion of two weeks just isolating machines and restoring data as quickly as possible". After trying to diagnose almost all software in the stack (e.g., Ext4, Software RAID [19]), they finally (mistakenly) concluded that it was Samsung's SSDs to blame. Samsung's SSDs were criticized and blacklisted, until one month later Samsung engineers found that it was a TRIM-related Linux kernel bug that caused the failure [20]. Similar confusing failures will likely increase in the foreseeable future as the system complexity keeps increasing [21–24].

Addressing the grand challenge will require cohesive efforts from the communities. Among others, a better understanding of real-world failure incidents and the potential limitations of state-of-the-art tools is critical. To this end, we first conduct an empirical study on 277 real-world storage failure issues in this paper. We characterize the issues along multiple dimensions (e.g., time to resolve, kernel components involved, hardware dependency), which enables us to quantitatively measure the reliability challenge as well as the need for better solutions.

Moreover, we analyze a set of storage issues in depth. By examining the user reports and bug patches meticulously and experimenting on real storage systems, we derive the necessary conditions (e.g., user/workload operations, library/kernel versions, system configurations) for triggering the failures deterministically. At the time of this

* Corresponding author.
E-mail addresses: duozhang@iastate.edu (D. Zhang), mai@iastate.edu (M. Zheng).

https://doi.org/10.1016/j.tbench.2021.100006
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 12 November 2021

writing, we are able to reproduce nine cases successfully, which covers four different file systems as well as the low-level block I/O layer of the Linux storage stack.

Based on the reproducible cases, we create a benchmark suite called *BugBench*^{k}, which includes a set of portable virtual machine (VM) images containing all the necessary software programs and environments to reproduce the nine storage failures caused by kernel-level bugs.[1] Complementary to existing benchmark suites which are mostly designed for measuring the performance [26–29], *BugBench*^{k} enables realistic evaluation on the capability of diverse reliability tools (e.g., kernel bug detectors [30,31], tracers [32], record & replay tools [33]), which is critical for identifying the potential limitations as well as the opportunities for further improvement.

To demonstrate the usage, we leverage *BugBench*^{k} to analyze two representative tools for debugging: (1) *FTrace*, the Linux kernel internal tracer [32]; and (2) *PANDA*, a VM-based record & replay tool [33]. Different from existing studies which mostly measure the tools' runtime overhead [34], we focus on measuring the *observability*, which means the observations that the tool allows the developers to make in order to diagnose the failure symptoms [35]. While the basic concept of observability is not new [35], we derive a set of concrete metrics to quantitatively and/or qualitatively measure the observability based on *BugBench*^{k}. Our experiments demonstrate the different design tradeoffs of the tools in terms of debugging information and space overhead. More importantly, we find that there are multiple tricky failure cases where both tools may fail to function properly. In other words, the usage of the tools may introduce interference to the target storage stack and make the failure symptom un-reproducible.

In addition, we find that neither tool can directly provide low-level information (e.g., storage device commands) on how the persistent storage states are changed, which is crucial for understanding host-device interactions in the storage stack. To address the limitation, we explore different ways to extend both FTrace and PANDA, and shows that it is possible to enhance both of them with such low-level observability without relying on special hardware (e.g., bus analyzer [36,37]).

It is well acknowledged that effective benchmark suites and tools are essential for improving various computer systems; on the other hand, building effective benchmark suites and tools is a long-term, iterative process that requires cohesive efforts from broad communities [25,34,38]. To the best of our knowledge, this work is the first effort to benchmark the observability of debugging tools with concrete metrics. We hope the *BugBench*^{k} prototype as well as the initial efforts demonstrated in this paper will inspire follow-up research on benchmarking and tool support for reliability, and help improve computer systems in general.[2]

The rest of the paper is organized as follows: in Section 2, we describe the background and extended motivation; in Section 3, we characterize real-world storage failures and derives the *BugBench*^{k}; in Section 4, we measure the observability of FTrace and PANDA based on *BugBench*^{k} and describe our extensions; in Section 5, we discuss related work; finally, we conclude the paper in Section 6.

2. Background & motivation

2.1. The storage stack & reliability challenge

Fig. 1 shows the typical storage stack,[3] which traditionally includes several major layers such as file systems, the block I/O layer, and device drivers. The recent introduction of PM technologies can provide access latencies less than 3X of DRAM while maintaining durability guarantee [40], which blurs the line between the storage management and the memory management in the kernel. Consequently, the memory management subsystem is also becoming part of the storage stack for persistent data.

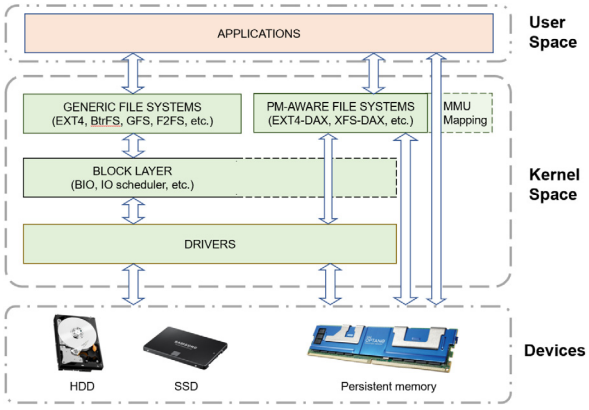


Fig. 1. The Storage Stack. The kernel-level software modules (green) are the major focus of this work.
Source: Adapted from [39].

The storage system is notoriously complex and difficult to get right despite decades of efforts [41–44]. Moreover, almost all layers in the storage stack are being optimized aggressively in recent years. For example, SSDs and PMs are replacing HDDs as the durable device [10–13,18,45]; NVMe [46] and CXL [47] are redefining the host-device interface; *blk-mq* [48] alleviates the single queue and lock contention bottleneck at the block I/O layer; the SCSI subsystem and the Ext4 file system are being adapted for NVM (e.g., scsi-mq [14–16] and DAX [17]); in addition, various NVM-oriented new designs/optimizations have been proposed (e.g., F2FS [49], NOVA [50], Kevlar [51]), some of which require cohesive modifications throughout the storage stack (e.g., the TRIM support [52]). Such modifications could potentially introduce various software bugs leading to system failures [41,53].

In practice, storage failures may occur due to various reasons including but not limited to software bugs [41–43,53], power outages [41,54], device errors [42,55,56], etc. Once a failure occurs, it is often difficult to diagnose the root cause due to the complexity of the storage stack, as demonstrated in the Algolia incident described in Section 1. However, despite the various anecdotes, there is little quantitative measurement or understanding of the characteristics of storage failures occurred in the real world. We attempt to address the issue in this work.

2.2. Debugging tools

Many tools have been built to improve system reliability. For example, *testing* tools (e.g., model checkers [41], fuzzers [30,31,57], fault injectors [58–61]) focus on triggering the potential bugs in target systems in a controlled testing environment *before* deployment to reduce the possibility of real-world failures. Once a system failure occurred in practice, however, testing tools can help little for pinpointing the root causes due to the different environments and assumptions.

Another category is *debugging* tools, which aims to facilitate diagnosing the root causes *after* a failure occurred in practice. While a benchmark suite containing real world cases may be used for evaluating both testing tools and debugging tools, we focus on debugging tools in this paper because: (1) they are much less studied compared to the abundant efforts on testing tools; (2) they are much needed in diagnosing real-world failure incidents. We classify existing debugging tools into three types as follows:

Interactive Debuggers. This category includes classic debuggers such as GDB/KDB/KGDB [62–64], which represents the *de facto* way to

[1] Coincidentally, there is an early work on application-level BugBench [25]; we elaborate on the difference and the synergy in Section 5.

[2] We release *BugBench*^{k} publicly on GitLab: https://git.ece.iastate.edu/data-storage-lab/prototypes/bugbench.

[3] Adapted from SNIA NVM Programming Model [39].

diagnose software system failures. They usually support fine-grained manual inspection (e.g., setting breakpoints at specific statements, checking the values of specific memory bytes). However, significant human efforts and expertise are needed to harness the power and to diagnose the complicated storage stack efficiently. Such traditional debugging methods are arguably not scalable, because the required manual effort and experience will keep increasing as the system becomes more and more complex. More automation and/or intelligence are probably needed to make debugging scalable.

Software & Hardware Tracers. Software tracers [32,65–69] can collect various events from a running target system automatically, which are typically implemented via dynamic instrumentation. The traced logs can help understand the system anomalies (among other purposes), which are often invaluable for failure diagnosis. Similarly, bus analyzers [36,37] are hardware equipments that can capture the low-level communication data (e.g., SCSI commands [70]) between the storage software and the device. However, since they only trace bus-level information, they cannot help much on understanding system-level behaviors.

Record & Replay Tools. Record & replay tools [33,71] have been applied to debugging for both user-level applications and the kernel. Typically, these tools leverage virtual machines to run the target software stack as a whole. Meanwhile, they record system snapshots as well as non-deterministic events (e.g., interrupts) to ensure replaying system execution faithfully. Developers can replay the recorded whole system execution logs repeatedly without the needs of re-running the workloads. Also, it is possible to integrate record & replay tools with interactive debuggers (e.g., GDB) to perform traditional debugging (e.g., setting breakpoints) during the replay. Additional analysis passes may also be implemented based on the record & replay mechanism (e.g., plugins in PANDA [33]).

Note that all of the three types of tools mentioned above are important debugging tools widely used in practice. But to the best of our knowledge, there is little quantitative measurement of their debugging capability. We attempt to address the deficiency in this work. We focus on software tracers and record & replay tools because they enable different degrees of automation for failure diagnosis, which we believe is critically important for a scalable debugging methodology. We leave the measurement of interactive debuggers (which requires manual effort/expertise that is difficult to quantify) and hardware tracers (which requires special hardware) as future work.

2.3. Observability of debugging tools

A few researchers have studied and benchmarked debugging tools [34] due to their prime importance in practice. However, existing studies mostly focus on the performance (e.g., runtime overhead) instead of their effectiveness, largely due to the lack of reliability benchmark suites.

Complementary to the existing efforts, we focus on the effectiveness of failure diagnosis. Specifically, we propose to measure the *observability* of debugging tools, which is a concept proposed recently for improving system reliability [35]. The *observability* includes three desired properties (i.e., visibility, repeatability, and expressibility) for debugging failures. Intuitively, the concept describes the observations that a tool allows the developers to make [35], which is critically important for debugging. While the concept of observability is well known, there is no practical methodologies to measure it to the best of our knowledge. We demonstrate how to measure the observability using realistic cases and concrete metrics in this work.

3. Characterization of storage failures

In this section, we describe how we collect the storage failure dataset (Section 3.1); the overall characteristics of the dataset (Section 3.2); and how we derive the $BugBench^k$ (Section 3.3).

Table 1
Overview of storage issues on Bugzilla.

Group	Count (%)	Avg. Days	Avg. Comments/ Participants
Resolved	136 (49.1%)	146.9	8/3
Unresolved	141 (50.9%)	1444.2	5/2
Overall	277	807.3	6/2

3.1. Methodology

To understand the characteristics of real-world problems of the storage stack, we collect failure issues reported by the end users from Linux Bugzilla [72]. We choose this platform because it is one major venue for regular users to report encountered failures to the Linux kernel community, and the reported issues are typically examined by the kernel developers with detailed status updates. Since the platform includes issues of the entire Linux kernel which is beyond the scope of the storage stack, we apply the following methods to refine the dataset.

First, Bugzilla organizes the issues based on major kernel components (e.g., "Process Management", "Networking"), so we search for the issues tagged with storage-related components (e.g., "File Systems", "IO/Storage", "Memory Management", "Drivers") or generic components (i.e., "Others"); also, the time of the issues is limited to the recent ten years. Next, in order to identify a manageable and important subset for study, we refine the dataset further by using a set of keywords implying severe failure consequences (e.g., "data loss", "corrupt"). Moreover, for the "Other" category, we further analyze the issues manually based on our domain knowledge and only keep storage-related ones (e.g., keeping software RAID issues but excluding GPU buffer corruptions). The resulting dataset contains 277 issues in total, which represents a subset of severe storage failures experienced by Linux end users. Note that this method is similar to the keyword search in previous studies [53,73].

Threats to validity. The characterization results presented in this section should be interpreted with the methodology in mind. In particular, the dataset was refined via critical keywords and manual efforts, which might be incomplete. Also, only a limited subset of Linux end users are aware of Linux Bugzilla and only a limited subset of them would report issues. Therefore, it is likely that there are more storage-related issues occurred in the wild but not captured in this study. Nevertheless, we believe our effort is one important step toward addressing the challenge.

3.2. Overall characteristics

Bugzilla maintains various status tags for the issues reported (e.g., "new", "closed"). For simplicity, we classify the issues into two groups based on their status tags: (1) *Resolved*, which includes issues with the "resolved" or "closed" status; (2) *Unresolved*, which includes issues with "new", "assigned", "reopened", or "verified" status. We summarize the two groups in Table 1. The second column shows the number of issues (and the percentage) in each group. The third column shows the average duration of the issues in days. For the Resolved group, the duration is calculated based on the report date and the date of the last comment; for the Unresolved group, it is the period between the report date and the time of this writing. The last column shows the average numbers of comments and participants in resolving the issues. We make multiple observations as follows:

First, *the issues took multiple months to resolve on average* (e.g., 146.9 days for the Resolved group in Table 1), and the diagnosis process typically involve multiple rounds of discussions and multiple people (e.g., 8 comments and 3 participants for the Resolved group), which quantitatively suggests the difficulty of diagnosing storage failures.

Second, *the issues involve all major components in the storage software stack*. As shown in Fig. 2, both Resolved and Unresolved groups span

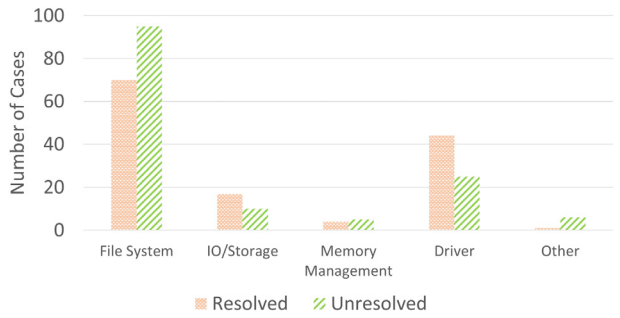


Fig. 2. Distributions of Resolved (orange) and Unresolved (green) Issues across Different Storage Components.

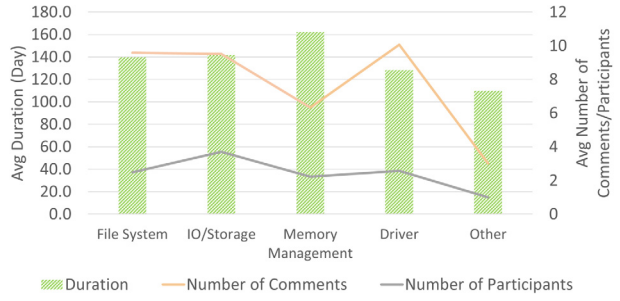


Fig. 3. Characteristics of Resolved Issues across Storage Components in terms of Average Duration (green bar), Average Number of Comments (orange line) and Average Number of Participants (gray line).

over all the five storage components studied. This implies that an ideal debugging tool should provide the full-stack observability. In particular, "File System" and "Driver" contains the most issues reported, which is consistent with previous studies [53].

Third, *for the resolved issues, the average debugging time is consistently long across components*. As shown in Fig. 3, the average debugging time of all five components is more than 100 days. This implies that due to the complexity of the storage stack, no single component is particularly easier to diagnose, which again suggests the importance of capturing the full-stack observability for debugging tools.

Fourth, *37 out of 136 (26.3%) resolved issues involve multiple OS distributions or kernel versions*. The manifestation symptoms of the issues often differ on different systems, which suggests that the software environment (e.g., kernels, libraries) is critically important for reproducing the failures for diagnosis.

Fifth, *only 5 out of 136 (3.7%) resolved issues were caused by hardware*. This implies that software remains the major source of storage failures, which is consistent with previous studies [74]. Also, it suggests that observing the behavior of the storage software stack is critically important for failure diagnosis.

3.3. $BugBench^k$

To identify the limitations of state-of-the-art debugging tools as well as the opportunities for further improvement, it is necessary to have a set of *reproducible* failure cases, so that we can apply the target tools and conduct the measurement. To this end, we analyze a set of storage failure issues in depth, identify the specific conditions required to trigger the issues (e.g., user/workload operations, software libraries involved, Linux kernel versions and configurations), and attempt to reproduce the cases on our server machines. This turns out to be a challenging and time-consuming process due to the complexity of the Linux kernel as well as the diversity of the Linux end users' system setups. For example, the reproducing procedure typically requires finding and (re)compiling specific versions of the Linux kernel with non-default configurations, pulling specific software packages which may not be well maintained, deriving workload programs to emulate various users' inputs, etc. At the time of this writing, we have identified 61 cases with relatively complete information for reproducing, and we are able to reproduce 3 cases successfully in our environment. This first-hand experience further confirms the challenge of failure diagnosis and the needs for a readily reproducible benchmark suite.

To ensure that the cases can be reliably reproduced and to enable easy sharing of the reproduced cases in the communities, we package all the required software programs and system environments in VM images. We create two VM images for each of the successfully reproduced cases: the first VM image contains the buggy storage stack and the necessary workload programs, libraries, etc. for reproducing the case; the second VM image contains the patched kernel (i.e., the bug in the storage stack has been fixed by the corresponding patch) to serve as a reference for verification.

In addition, to improve the case count as well as the diversity of the reproducible cases, we collect additional storage-related bug cases from the Linux mailing lists [34]. The cases reported on the Linux mailing lists are often discovered by the developers directly during the internal regression testing, so they may not contain the same information as the issues reported by the end users on Bugzilla (e.g., no user experienced consequences, user environments, or resolving status). Such developer-discovered cases are relatively less valuable for characterizing the real-world failure impact or diagnosis difficulty (as in Section 3.2). However, these cases are still realistic in that they may affect Linux distributions released earlier (i.e., before the bug patch). In other words, if they can be reproduced readily, they are as valuable as the Bugzilla issues for measuring debugging tools and other reliability utilities. At the time of this writing, we are able to reproduce 6 storage-related cases from the Linux mailing list successfully. We also create the corresponding VM images for the 6 cases to facilitate future reproducible research.

Based on the 9 reproducible cases (i.e., 3 from Bugzilla and 6 from the Linux mailing list), we have created a benchmark suite called $BugBench^k$, which includes a set of VM images containing all the necessary workloads and system environments/configurations to reproduce the 9 cases. We summarize the 9 cases in Table 2. As shown in the table, the current prototype of $BugBench^k$ covers 4 different file systems, including 2 cases for Ext4 (i.e., "1-EXT4", "2-EXT4"), 3 cases for Btrfs file system (i.e., "3-BTRFS", "4-BTRFS", "5-BTRFS"), 1 case for F2FS (i.e., "6-F2FS"), and 1 case for GFS2 (i.e., "7-GFS"). Moreover, there are 2 cases for the low-level block I/O layer of the storage stack (i.e., "8-BLK" and "9-BLK").

The "Critical Function" column in Table 2 shows the major kernel functions that are identified as problematic for each case. We can see that the number of critical functions ranges from 1 to 7 (in "6-F2FS"), depending on the complexity of the bug fixes.

The root causes of the 9 cases can be classified into either "Semantics" bugs (7 cases) or "Memory" bugs (2 cases) based on the bug patterns defined in the literature [53,73,75]. Unlike other types of bugs that have well-studied patterns to understand (e.g., deadlocks, data races), "Semantics" bugs is among the hardest issues in practice because they typically require deep understanding of system design logic to detect or diagnose. In other words, the cases included $BugBench^k$ require sophisticated methodologies to diagnose effectively.

The "Bug Size" is defined as the sum of lines of insertion and deletion code (LoC) in the bug patch, which ranges from 6 (in "2-EXT4') to 121 (in "4-BTRFS") LOC depending on the complexity of the cases. The last column shows the language we used to implement the bug triggering workloads. We use Bash, C, or a combination of both to implement the workloads, depending on the input characteristics described in the user reports (for Bugzilla cases) or bug patches (for Linux mailing list cases).

Table 2
Overview of reproducible cases in *BugBench*^{k}.

Case ID	OS Image	Linux Kernel	Storage Component	Critical Function	Bug Type	Bug Size	Workload Language
1-EXT4	Ubuntu 20.04	v5.4.0	Ext4 File System	ext4_do_update_inode, ext4_isize_set, ext4_clear_inode_state, cpu_to_le16,cpu_to_le32, ext4_update_inode_fsync_trans	Semantics	8	C
2-EXT4	Ubuntu 20.04	v5.4.0	Ext4 File System	parse_options	Semantics	6	C & Bash
3-BTRFS	Ubuntu 16.04	v4.4.107	BTRFS File System	btrfs_ioctl_snap_destroy, btrfs_record_snapshot_destroy, btrfs_set_log_full_commit, check_parent_dirs_for_sync, btrfs_log_inode, btrfs_release_path	Semantics	71	C
4-BTRFS	Ubuntu 16.04	v4.4.107	BTRFS File System	btrfs_log_trailing_hole	Semantics	121	C
5-BTRFS	Ubuntu 20.04	v5.4.0	BTRFS File System	btrfs_log_all_parents, btrfs_log_inode, btrfs_must_commit_transaction, btrfs_record_unlink_dir	Semantics	13	C
6-F2FS	Ubuntu 16.04	v4.15.0	F2FS File System	f2fs_submit_page_bio, f2fs_is_valid_blkaddr, verify_block_addr, zero_user_segment,, validate_checkpoint, datalock_addr	Memory	94	C & Bash
7-GFS	Ubuntu 16.04	v4.4.0	GFS2 File System	gfs2_check_sb, fs_warn	Memory	18	Bash
8-BLK	Ubuntu 20.04	v5.4.0	Block Layer	blkdev_fsync, sync_blkdev	Semantics	12	C
9-BLK	Ubuntu 18.10	v4.19.1	Block Layer	_blk_mq_issue_directly, blk_mq_update_dispatch_busy, _blk_mq_requeue_request	Semantics	9	Bash

We choose Ubuntu to reproduce the cases by default because Ubuntu is one of the most well supported OS distributions for many Linux tools. Also, since many utilities and packages are outdated or even not usable on old kernels, we port the cases to the latest kernel (i.e., v5.4.0) when possible. For 5 cases (i.e., "3-BTRFS", "4-BTRFS", "6-F2FS", "7-GFS", "9-BLK"), we are not able to reproduce the cases in the latest kernel because the affected kernel structures have been changed significantly and the original problematic functions are no longer compatible with the latest kernel. Therefore, we have to reproduce them in relatively old versions (e.g., v4.4.107, v4.15.0, v4.4.0) where the cases are still reproducible.

To sum up, the current prototype of *BugBench*^{k} includes a set of VM images for reproducing 9 realistic storage failure cases. These reproducible cases, including the complete software workloads and environments encapsulated in VMs, enable us to measure and evaluate the effectiveness of tools conveniently. We demonstrate the usage of *BugBench*^{k} in the context of two representative debugging tools in the next section (Section 4).

4. Measuring the observability

In this section, we apply *BugBench*^{k} to study FTrace and PANDA, both of which are state-of-the-art tools widely used for debugging (among other usages). We find that *BugBench*^{k} can help measuring tools with completely different design principles. Also, we find that both FTrace and PANDA may provide useful information for the majority of the cases evaluated. On the other hand, both of them may behave abnormally when diagnosing some tricky cases. We elaborate on the experimental results of FTrace and PANDA in Section 4.1 and Section 4.2, respectively. In addition, we find that both tools fall short of providing low-level information on how the persistent states are changed. We discuss our extensions to improve their low-level observability in Section 4.3.

4.1. FTrace

FTrace is the Linux kernel internal tracer that has been included in the mainline Linux since v2.6.27 [32]. We measure the observability of its major feature (i.e., kernel function tracing) in this subsection.

Table 3 summarizes the results of applying FTrace to diagnose the 9 cases in *BugBench*^{k} (labeled from "1-EXT4" to "9-BLK" in the first column). The second column ("Still Reproducible") shows whether the bug cases can still be reproduced when enabling FTrace to trace the target storage stack. We can see that FTrace do not affect the reproducibility of any case. In other words, the tool is non-intrusive for debugging all the cases in *BugBench*^{k}.

The third column shows the total number of functions (Func.) traced by FTrace in each case, which may include duplicated entries if a kernel function is invoked multiple times during the workload execution. We run FTrace for three times and calculate the average count (e.g., "12,506" for "1-EXT4") and the range of variance (e.g., "±4.1%"). We can see that FTrace can generate a large amount of functions for most cases, ranging from "6867" (in "8-BLK" case) to "110,772,722" (in "9-BLK" case), which implies that the tool can provide rich function-level information for diagnosing the target system behavior.

Similarly, the fourth column shows the number of unique functions traced in each case (i.e., excluding duplicated entries), which is generally much smaller compared to the total number of functions traced (i.e., the third column). This implies that the same kernel functions may be invoked many times in all the failure cases. From debugging's perspective, the large redundancy in the trace could exacerbate the challenge of diagnosing system behavior.

The fifth column ("Critical Func. Observed") measures how many of the critical functions can be observed by FTrace in each case. As mentioned in Section 3.3, a critical function is a problematic kernel

Table 3
FTrace results on 9 $BugBench^k$ cases.

Case ID	Still Reproducible?	Total # of Func. Traced	Total # of Unique Func.	Critical Func. Observed	Shortest Distance	Log Size (MB)
1-EXT4	Yes	12,506 (±4.1%)	1,152 (±0.7%)	1/7	–	2.07 (±0.01)
2-EXT4	Yes	54,796 (±2.3%)	1,436 (±15.9%)	0/1	2	9.17 (±0.03)
3-BTRFS	Yes	46,370 (±5.6%)	1,339 (±1.5%)	3/6	–	6.87 (±0.10)
4-BTRFS	Yes	92,476 (±5.5%)	1,381 (±1.0%)	0/1	1	14.1 (±0.43)
5-BTRFS	Yes	30,528 (±3.6%)	1,419 (±1.5%)	3/4	–	5.2 (±0.03)
6-F2FS	Yes	0	0	0/7	–	0
7-GFS	Yes	0	0	0/2	–	0
8-BLK	Yes	6,867 (±2.7%)	901 (±4.3%)	1/2	–	1.1 (±0)
9-BLK	Yes	110,772,722(±6.4%)	1,165(±0.8%)	2/3	–	7,496.2 (±153.1)

function that contributes to the storage failure. A failure case may have multiple critical functions as the root cause, depending on the complexity of the failure. We can see that although FTrace can trace many functions, it may not be able to capture the critical functions effectively for the cases in $BugBench^k$. For example, in "1-EXT4", there are 7 critical functions but only one of them can be captured by FTrace (i.e., "1/7"). Similarly, in four other cases (i.e., "3-BTRFS", "5-BTRFS", "8-BLK", and "9-BLK"), only partial critical functions can be observed (i.e., "3/6", "3/4", "1/2", and "2/3", respectively).

In terms of "2-EXT4" and "4-BTRFS", none of the critical functions in the two cases can be directly observed by FTrace (i.e., "0/1" in the fifth column for both cases). To measure the relevance of the traced functions in these two cases, we further calculate the "Shortest Distance" (the sixth column), which is defined by the minimum number of function invocations needed from the traced functions to the critical functions. We find that although FTrace misses the critical function in "4-BTRFS", it actually captures the parent function (i.e., the "Shortest Distance" is "1") correctly. Similarly, it captures the parent's parent function of the missing critical function in "2-EXT4" (i.e., the "Shortest Distance" is "2"). This implies that FTrace may still be helpful for diagnosing failures even if it may miss some specific functions.

To understand why FTrace may not be able to trace all critical functions for debugging the cases in $BugBench^k$, we look into the internals of FTrace. We find that FTrace relies on a pre-defined list for identifying traceable functions, which is stored in $available\_filter\_functions$ file in the $debugfs$ of the target system. Moreover, the default list may contain different functions on different kernel versions we evaluated. This list fundamentally limits the observability of FTrace for debugging diverse failure scenarios, as exposed by the incomplete critical functions in the fifth column ("Critical Func. Observed"),

In terms of "6-F2FS" and "7-GFS", the two cases are still reproducible with FTrace enabled, but FTrace cannot help much in either case (i.e., "0" in "Total # of Func. Traced"). This is because the manifestation of the two cases is kernel panics. Under such a scenario, FTrace cannot function normally. This result exposes a fundamental limitation of FTrace for debugging the storage stack in the kernel: FTrace itself depends on the probes or tracepoints embedded in the kernel, so it cannot survive severe kernel problems (e.g., kernel panics), let alone help diagnosing the problem in such severe scenarios.

The last column shows the size of the logs generated by FTrace under $BugBench^k$. We can see that FTrace consumes a relatively small amount of storage space for most cases, ranging from 1.1MB ("8-BLK") to 14.1MB ("4-BTRFS"). Since the log size largely depends on the amount of workload operations, the low storage overhead implies that workloads included in $BugBench^k$ are concise and effective for triggering the 7 cases.

On the other hand, the last case ("9-BLK") incurs a relatively large amount of storage overhead (i.e., around 7496 MB). This is because the failure requires a relatively heavy workload to trigger. Specifically, the workload includes pulling and installing many software packages from the Internet via the *dpkg* package manager, which involves both the network subsystem and the storage stack and leads to a large amount of kernel functions being traced (i.e. , "110,772,722"). Since only 3 critical functions contribute to the failure in the "9-BLK" case, the substantial amount of traced functions may dilute the debugging focus. In other words, more intelligent methodologies are likely needed to help derive insights from the abundant FTrace logs for debugging.

4.2. PANDA

PANDA (Platform for Architecture-Neutral Dynamic Analysis) is an open-source platform for program analysis [33]. By leveraging virtualization (i.e., QEMU [76]) and the LLVM compiler infrastructure [77], PANDA can help understand the behavior of the entire storage software stack. We focus on measuring its major feature (i.e., record & replay) and 4 related plugins (i.e., Show Instructions, Taint Analysis, Identify Processes, Process-Block Relationship) in this subsection because they are most relevant for diagnosing storage failures.

Since PANDA records the full state of a target system hosted in QEMU as well as all non-deterministic events in snapshots, it can achieve full-stack, all-instruction observability by design (i.e., all executed instructions are observable by replaying). Therefore, we do not calculate the function-based metrics as used in measuring FTrace (Section 4.1). Instead, we qualitatively measure if the target features are applicable in diagnosing failures.

Table 4 summarizes the results of applying PANDA to diagnose the 9 cases in $BugBench^k$. Similar to Table 3, the second column ("Still Reproducible") shows whether the bug cases can still be reproduced when using PANDA. We observe that PANDA do not introduce any interference for the first 8 cases, similar to FTrace.

Nevertheless, PANDA fails in the last case ("9-BLK"). Specifically, we observe that the guest VM is hanging when applying PANDA to diagnose the "9-BLK" case. Multiple factors may contribute to the hang. First, as mentioned in Section 4.1, the workload requires installing many packages which are pulling from the Internet via *dpkg*. In other words, this workload involves the network subsystem and tends to generate many non-deterministic events within the kernel. Secondly, the QEMU-based PANDA needs to record all such events in order to ensure a successful replay, which incurs significant overhead in the critical path of QEMU's translation of guest instructions. As a result, QEMU is overloaded by PANDA's event recording, and cannot finish the translation of guest instructions on time. Eventually, the guest kernel (to be diagnosed) hangs in the QEMU VM. This result suggests that the state-of-the-art record & replay mechanisms may not be lightweight enough for diagnosing tricky storage failures.

For the remaining 8 cases, we find that PANDA's major record & replay feature and the 4 relevant plugins can all work normally (i.e., "✓") to support full-stack observability. On the other hand, the full-stack, all-instruction observability comes at the cost of overhead. The last column shows that PANDA incurs hundreds of MB storage overhead for its snapshots and event logs in most cases, which is orders of magnitude larger than the logs generated by FTrace on the same cases (Table 3).

Note that in terms of "6-F2FS" and "7-GFS" where FTrace fails due to kernel panics, PANDA can still work properly. This suggests a unique advantage of VM-based debugging tools like PANDA: by isolating the target storage software stack in the guest VM, the tool itself can survive severe problems of the target system and still provide effective support for diagnosing the problem.

Table 4
PANDA results on 9 $BugBench^k$ cases.

Case ID	Still Reproducible?	Record & Replay	Plugin				Log Size (MB)
			Show Instructions	Taint Analysis	Identify Processes	Process-Block Relationship	
1-EXT4	Yes	✓	✓	✓	✓	✓	659.9
2-EXT4	Yes	✓	✓	✓	✓	✓	671.9
3-BTRFS	Yes	✓	✓	✓	✓	✓	380.7
4-BTRFS	Yes	✓	✓	✓	✓	✓	811.7
5-BTRFS	Yes	✓	✓	✓	✓	✓	683.1
6-F2FS	Yes	✓	✓	✓	✓	✓	451.7
7-GFS	Yes	✓	✓	✓	✓	✓	408.7
8-BLK	Yes	✓	✓	✓	✓	✓	658.7
9-BLK	No	N/A	N/A	N/A	N/A	N/A	N/A

Fig. 4. An Example of Augmented FTrace Log for 8-BLK.

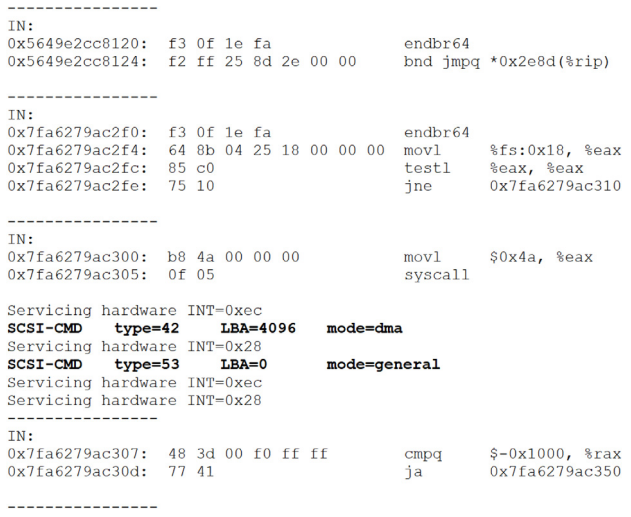


Fig. 5. An Example of Augmented QEMU Log for 8-BLK.

4.3. Enhancing low-level observability

Through the experiments with $BugBench^k$, we find that neither FTrace nor PANDA can provide direct observability on the lowest level of information communicated between the storage software and the storage device, i.e., the device commands (e.g., SCSI [70]). Such command-level information is valuable in diagnosing storage failures because the persistent storage states are changed by the device commands directly. Traditionally, bus analyzers [36,37] are used to capture such command-level information. But as mentioned in Section 2.2, bus analyzers are hardware-based tools which are not as convenient as software tools. We introduce software extensions to capture bus-analyzer-like command information and enhance the low-level observability of both FTrace and PANDA in this subsection.

FTrace Extension. To avoid introducing unnecessary complexity to FTrace's probe mechanism in the kernel, we use an indirect way to extend FTrace. Specifically, we use a customized iSCSI driver [78] to collect device commands with timestamp, and align the collected device commands with the original FTrace logs based on timestamp. In doing so, the kernel functions are augmented with low-level device commands under the corresponding critical I/O paths.

We have verified that this extension method works for all the cases where FTrace can work normally without the extension. As an example, Fig. 4 shows a simplified version of the extended FTrace logs for the "8-BLK" case. The traced kernel functions are augmented with additional SCSI commands (bolded) based on timestamp. Fig. 4(a) shows the augmented log of an abnormal run (i.e., the bug is triggered), where we can see that the function $blkdev\_fsync$ eventually generates a write command to the device ("SCSI-CMD: WRITE"). This is problematic because the high-level sync function (i.e., $blkdev\_fsync$) should generate a low-level sync operation (instead of simply a regular write operation) at the device command level. Fig. 4(b) shows the augmented log of a corresponding normal run. We can see that an additional sync command ("SCSI-CMD: SYNC_CACHE") is actually generated within the scope of the $blkdev\_fsync$ function, which is expected.

Essentially, our extension combines the features of FTrace and the traditional hardware-based bus analyzer [36,37]. By extending FTrace logs with the command-level information in this way, we enhance the low-level observability of FTrace without using special hardware.

PANDA Extension. As mentioned in Section 4.2, PANDA uses QEMU to host the entire storage software stack in the guest VM, so the iSCSI driver solution for FTrace does not work for PANDA. Instead, we modify QEMU to capture all command-level information and leverage QEMU's internal logging mechanism to align commands with instructions.

Specifically, in QEMU, the guest OS kernel communicates with a SCSI device by sending Command Descriptor Blocks (CDBs) over the bus. QEMU maintains a 'struct SCSICommand' for each SCSI command, which contains a 16-byte buffer ('SCSICommand->buf') holding the CDB. Every SCSI command type is identified by the opcode at the beginning of the CDB, and the size of CDB is determined by the opcode. For example, the CDB for the WRITE_10 command is represented by

the first 10 bytes of the buffer. For simplicity, we always transfer 16 bytes from the buffer to the command log and use the opcode to identify valid bytes. QEMU classifies SCSI commands into either Direct Memory Access (DMA) commands (e.g., READ_10) or Admin commands (e.g., VERIFY_10), and both are handled in the same way in our extension since they share the same data structure.

As an example, Fig. 5 shows a simplified version of the augmented QEMU log for the "8-BLK" case. The two bold lines (i.e., "SCSI-CMD ...") are the added device command information. and the remaining lines are the original QEMU log which includes both instructions (i.e., lines starting with addresses "0x5649e2..." etc.) and interrupts (e.g., "Servicing hardware INT=0xec"). The dash lines show the translation iteration of QEMU, each of which includes one basic block of instructions and the relevant device commands (if any). Similar to the FTrace extension, we enhance the low-level observability of PANDA/QEMU log without relying on special hardware.

4.4. Summary & discussion

To sum up, we have measured and evaluated the debugging observability of FTrace and PANDA via $BugBench^k$. Through the experiments, it is clear that FTrace and PANDA have different design tradeoffs and provide different level of observability. Moreover, we have demonstrated that it is possible to enhance their low-level observability via different lightweight extensions without hardware.

In particular, the results of FTrace suggest that tracing-based tools may be fundamentally limited for diagnosing storage failures in two aspects: (1) they may trace too many functions/events most of which may not be relevant to the root cause; (2) they may fail to function properly when the target storage system is malfunctional severely (e.g., kernel panics as in "6-F2FS" and "7-GFS").

On the other hand, the results of PANDA suggest that VM-based tools may be more viable for diagnosing storage failures because they can isolate the entire storage software stack from the core debugging functionality. However, in complicated failure scenarios, the events monitored may overwhelm the virtualization layer (e.g., "9-BLK" for PANDA), which suggests that more lightweight and less intrusive methods are needed to leverage virtualization for debugging.

We focus on measuring the observability of the debugging tools in this work because this is one of the most important metrics for debugging failures [35]. We envision many opportunities for further improvements based on the initial effort. For example, we recognize that observability is only one desired property proposed recently for improving system reliability [35]. There are other important properties and tools which may be measured by using $BugBench^k$ (e.g., runtime overhead of debugging tools, false positive rates of bug detection tools). Also, the current prototype of $BugBench^k$ only contains 9 reproducible cases due to the difficulty of reproducing real-world storage failures with incomplete information (as discussed in Section 3.3). And unfortunately, based on our investigation, none of the 277 issues collected in our dataset (Section 3) are directly related to the PM modules introduced to the storage stack recently. In terms of debugging tools, we only measure the core features of FTrace (i.e., kernel function tracing) and PANDA (i.e., record & replay and 4 related plugins) in our experiments. In fact, both FTrace and PANDA provide a rich set of additional features which might also be helpful for failure diagnosis. For example, FTrace allows users to add additional events tracing based on tracepoints [32]. Similarly, PANDA has additional plugins built on top of its record & replay framework. We leave reproducing PM-specific cases, deriving additional metrics, and measuring other debugging features and tools as future work.

While we only touch the observability of diagnosing the storage stack in this paper, the concept is also applicable in other contexts. In particular, researchers and practioners have recognized the critical importance of observability in the Cloud Native environment [79], where various components have been developed to enhance the observability to meet service-level objectives (SLOs) [79]. However, different from the modern Cloud-Native environment which supports loosely-coupled microservices and enables flexible integration of monitoring, tracing, logging, etc. services for observability, the storage stack in the monolithic Linux kernel has more constraints. How to improve the observability for the monolithic kernel with minimal intrusion remains an open question. Our effort on measuring the observability of state-of-the-art tools is one first step toward addressing the challenge.

Finally, we would like to point out that our goal of measuring the observability of different debugging tools in this work is not to imply which one is better. Instead, we hope to identify the potential limitations of the state-of-the-art tools in the context of diagnosing realistic storage failures, and inspire further improvements to address the debugging challenge. And as shown in our experiments, although the total number of reproducible cases is relatively small, $BugBench^k$ and the associated metrics have already exposed the limitations of the state-of-the-art tools evaluated, which suggests the needs and opportunities for more advanced diagnosis support. We hope that our initial $BugBench^k$ effort and the proof-of-concept extensions can inspire more follow-up research efforts in the communities, and contribute to the development of benchmarking for system reliability in general.

5. Related work

In this section, we discuss four categories of related work that have not been covered sufficiently in the previous sections.

Benchmarking Storage Systems. Great efforts have been made to benchmark and measure various storage systems [25–28,80,81]. For example, FIO [27] allows specifying diverse I/O patterns (e.g., sequential/random/mixed read or write operations) in multiple threads or processes. WHISPER [80] includes ten PM applications covering three types of access interfaces to PM, which enables analyzing the characteristics of PM applications (e.g., percentage of writes to PM, number of ordering points). Complementary to these existing benchmarking efforts which mostly focus on measuring the performance metrics, we introduce $BugBench^k$ and a set of metrics to enable quantitative and qualitative measurement of debugging observability.

Coincidentally, there is an early work by Lu et al. which is called BugBench [25]. The authors collected 17 bug cases in C/C++ applications, and proposed to evaluate bug detections tools based on the bug cases. Different from BugBench which includes application-level bug cases, $BugBench^k$ includes buggy OS kernels covering major components of the storage stack (e.g., multiple file systems, and low-level block I/O software), which are arguably more difficult to package, reproduce, or diagnose compared to user-level applications. Also, different from BugBench which focuses on evaluating user-level *testing* tools, we focus on evaluating the `debugging` tools for diagnosing failures, which is critically important for resolving failures in the real world but unfortunately is much less investigated compared to the abundant research on bug detection [30,31]. On the other hand, both BugBench and $BugBench^k$ focus reliability-oriented metrics (e.g., false positive rates, observability) and aims to improve the system robustness, which is different from traditional performance-oriented metrics. Therefore, we view the two efforts as complementary to each other. We hope that by reviving the concept of benchmarking for observability and other important reliability-oriented properties of computer systems, this work will inspire follow-up research and help improve the robustness of systems in general.

Studies of Software Bugs and Failures. Many researchers have performed empirical studies on bugs or failures in software systems [73–75,82–84]. For example, Lu et al. [75] studied 5079 patches from 6 Linux file systems and identified evolution trends; Lu et al. [73] studied 105 concurrency bugs from 4 applications and identified a number of common bug patterns (e.g., atomicity-violation and order-violation); Duo et al. [53] studied 1350 PM-related kernel patches and identified a number of PM bug characteristics including PM patch

categories, PM bug patterns, consequences, and fix strategies; Gunawi et al. [85] studied 597 cloud service outages and derived multiple lessons including the outage impacts, causes, etc; Liu et al. [86] studied hundreds of incidents in Microsoft Azure.

Generally, our work is complementary to the existing ones as we focus on bugs in the entire storage stack experienced by the end users, which has a different scope compared to most of the existing studies. Moreover, we reproduce a set of cases and derive a $BugBench^k$ to measure representative debugging tools, which is beyond the scope of existing empirical studies.

Characterizing Storage Devices. Many researchers have studied the behaviors of storage devices in depth, including both HDDs [87–89] and SSDs [1,55,56,59,90–96]. For example, Bairavasundaram et al. [88] analyze the data corruption and latent sector errors in production systems containing 1.53 million HDDs; Maneas et al. [56] study the reliability of 1.4 million SSDs deployed in NetApp RAID systems. Schroeder et al. [89] analyze the disk replacement data of seven production systems over five years. Generally, these studies may provide valuable insights for reasoning complex storage failures caused by device. Different from these device-level studies, we analyze the storage failures at the system level involving different kernel components, which is complementary to the existing work.

Testing Storage Software. Great efforts have been made to test various storage software systems [41,54,58,60,61,97–101], with the goal of exposing bugs that could lead to failures. For example, EXPLODE [41] uses modeling checking to find storage system bugs [41], and B^3 applies bounded black-box testing to detect crash-consistency bugs in file systems [58]. However, testing tools are generally not suitable for diagnosing system failures because they typically require a well-controlled environment (e.g., a highly customized kernel [41,58]), which may be substantially different from the storage stack that need to be diagnosed. While the goal of this work is not to develop a new bug detection tool, the $BugBench^k$ created in this work may be used to evaluate such tools as well (e.g., false positive rate on detecting the reproducible bugs), which we leave as future work.

6. Conclusions

We have studied 277 real-world storage failures to quantitatively understand their characteristics. Based on the characterization, we derived a $BugBench^k$ which includes the necessary workloads and software environments to reproduce 9 realistic storage failure cases. We applied $BugBench^k$ to study two representative open source tools and derived concrete metrics to quantitatively/qualitatively measure their debugging observability. Moreover, we demonstrated that it is possible to enhance the observability of the state-of-the-art tools via lightweight extensions.

To the best of our knowledge, this work is the first effort to measure the observability of debugging tools. The work demonstrated in this paper suggests many opportunities for further improvements such as reproducing and packaging other types of bugs cases, deriving additional metrics for other desirable system properties, and measuring other tools or features, which we leave as future work. We hope that our initial effort will inspire follow-up research in the communities and help measure and improve the robustness of computer systems in general.

Acknowledgments

We thank the anonymous reviewers for their insightful feedback. We also thank researchers from Western Digital including Adam Manzanares, Filip Blagojevic, Qing Li, and Cyril Guyot for valuable discussions on the internals of the storage stack and the latest storage technology. In addition, we thank Wei Xu, Om Rameshwar Gatla, Prakhar Bansal, Runzhou Han, and Philip Ma for the help on reproducing and/or analyzing failure reports and bug patches. This work was supported in part by NSF, United States under grants CNS-1566554 and CNS-1943204. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

References

[1] Ryan Gabrys, Eitan Yaakobi, Laura M. Grupp, Steven Swanson, Lara Dolecek, Tackling intracell variability in TLC Flash through tensor product codes, in: 2012 IEEE International Symposium on Information Theory Proceedings, 2012.
[2] Yu Cai, Erich F. Haratsch, Onur Mutlu, Ken Mai, Error patterns in MLC NAND flash memory: Measurement, characterization, and analysis, in: Proceedings of the Conference on Design, Automation and Test in Europe, DATE, 2012.
[3] Laura M. Grupp, Adrian M. Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, Jack K. Wolf, Characterizing flash memory: anomalies, observations, and applications, in: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO, 2009.
[4] H Kurata, K Otsuga, A Kotabe, S Kajiyama, T Osabe, Y Sasago, S Narumi, K Tokami, S Kamohara, O Tsuchiya, The impact of random telegraph signals on the scaling of multilevel flash memories, in: VLSI Circuits, 2006. Digest of Technical Papers, 2006.
[5] Hanmant P Belgal, Nick Righos, Ivan Kalastirsky, Jeff J Peterson, Robert Shiner, Neal Mielke, A new reliability model for post-cycling charge retention of flash memories, in: Proceedings of the 40th Annual Reliability Physics Symposium, 2002.
[6] Kang-Deog Suh, Byung-Hoon Suh, Young-Ho Lim, Jin-Ki Kim, Young-Joon Choi, Yong-Nam Koh, Sung-Soo Lee, Suk-Chon Kwon, Byung-Soon Choi, Jin-Sun Yum, Jung-Hyuk Choi, Jang-Rae Kim, Hyung-Kyu Lim, A 3.3v 32mb NAND flash memory with incremental step pulse programming scheme, in: IEEE J. Solid-State Circuits (JSSC), 1995.
[7] T. Ong, A. Frazio, N. Mielke, S. Pan, N. Righos, G. Atwood, S. Lai, Erratic erase in ETOX/sup TM/ flash memory Array, in: Symposium on VLSI Technology, VLSI, 1993.
[8] Adam Brand, Ken Wu, Sam Pan, David Chin, Novel read disturb failure mechanism induced by FLASH cycling, in: Proceedings of the 31st Annual Reliability Physics Symposium, 1993.
[9] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, Sam H Noh, Failure-atomic slotted paging for persistent memory, ACM SIGPLAN Not. (2017).
[10] Advanced Flash Technology Status, Scaling Trends & Implications to Enterprise SSD Technology Enablement, https://www.flashmemorysummit.com/English/Collaterals/Proceedings/2012/20120821_TA12_Yoon_Tressler.pdf.
[11] Justin Meza, Qiang Wu, Sanjeev Kumar, Onur Mutlu, A large-scale study of flash memory failures in the field, in: ACM SIGMETRICS Performance Evaluation Review, 2015.
[12] Flash array, https://patents.google.com/patent/US4101260A/en.
[13] Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, https://arxiv.org/abs/1903.05714.
[14] Scsi-mq, 2017, https://lwn.net/Articles/602159/, March 20.
[15] Blake Caldwell, Improving block-level efficiency with scsi-mq, 2015, arXiv preprint arXiv:1504.07481.
[16] Bart Van Assche, Increasing SCSI LLD driver performance by using the SCSI multiqueue approach, 2015.
[17] DAX: Page cache bypass for filesystems on memory storage, https://lwn.net/Articles/618064/.
[18] When solid state drives are not that solid, 2015, https://blog.algolia.com/when-solid-state-drives-are-not-that-solid/, June 15.
[19] A guide to mdadm, https://raid.wiki.kernel.org/index.php/A_guide_to_mdadm.
[20] raid0: data corruption when using trim, 2015, https://www.spinics.net/lists/raid/msg49440.html, July 19.
[21] Failure on freebsd/SSD: Seeing data corruption with zfs trim functionality , 2013, https://lists.freebsd.org/pipermail/freebsd-fs/2013-April/017145.html, April 29.
[22] Discussion on kernel TRIM support for SSDs: [1/3] libata: Whitelist SSDs that are known to properly return zeroes after TRIM , 2014, http://patchwork.ozlabs.org/patch/407967/, Nov 7 - Dec 8.
[23] Discussion on data loss on mSATA SSD module and Ext4 , 2016, http://pcengines.ch/msata16a.htm.
[24] HP warns that some SSD drives will fail at 32,768 hours of use, https://www.bleepingcomputer.com/news/hardware/hp-warns-that-some-ssd-drives-will-fail-at-32-768-hours-of-use/.
[25] Shan Lu, Zhenmin Li, FengQin, Lin Tan, Pin Zhou, YuanyuanZhou, BugBench: Benchmarks for evaluating bug detection tools, in: Workshop on the Evaluation of Software Defect Detection Tools, 2005.
[26] FIO Benchmark, https://fio.readthedocs.io/en/latest/fio_doc.html.
[27] Vasily Tarasov, Erez Zadok, Spencer Shepler, Filebencha flexible framework for file system benchmarking, in: Login Usenix Magazine, 2016.
[28] TPC Benchmarks, http://tpc.org/information/benchmarks5.asp.
[29] SPEC's benchmarks, https://www.spec.org/benchmarks.html.

[30] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, Taesoo Kim, Krace: Data race fuzzing for kernel file Systems, in: 2020 IEEE Symposium on Security and Privacy, SP, 2014.
[31] Dae R. Jeong, Kyungtae Kim, Basavesh Shivakumar, Byoungyoung Lee, Insik Shin, Razzer: Finding kernel race bugs through fuzzing, in: 2019 IEEE Symposium on Security and Privacy, SP, 2019.
[32] ftrace, https://elinux.org/Ftrace.
[33] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, Ryan Whelan, Repeatable reverse engineering with PANDA, in: Proceedings of the 5th Program Protection and Reverse Engineering Workshop, 2015.
[34] Mohamad Gebai, Michel R. Dagenais, Survey and analysis of kernel and userspace tracers on linux: Design implementation and overhead, in: ACM Computing Survey, 2018.
[35] Andrew Quinn, Jason Flinn, Michael Cafarella, You can't debug what you can't see: Expanding observability with the OmniTable, in: Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS, 2019.
[36] How to Read a SCSI Bus Trace, https://www.drdobbs.com/how-to-read-a-scsi-bus-trace/199101012.
[37] SCSI bus analyzer, https://www.ibm.com/support/knowledgecenter/en/ssw_aix_72/diagnosticsubsystem/header_54.html.
[38] Benchmarks by BenchCouncil, https://benchcouncil.org/benchmarks.html.
[39] SNIA NVM Programming Model (NPM).
[40] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, Steve Swanson, An empirical guide to the behavior and use of scalable persistent memory, 18th USENIX Conference on File and Storage Technologies, FAST, 2020.
[41] Junfeng Yang, Can Sar, Dawson Engler, Explode: a lightweight, general system for finding serious storage system errors, in: Proceedings of the 7th Symposium on Operating Systems Design and Implementation, OSDI, 2006.
[42] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, IRON File Systems, in: Proceedings of the 20th ACM Symposium on Operating Systems Principles, SOSP, 2005.
[43] Lanyue Lu, Yupu Zhang, Thanh Do, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Physical disentanglement in a container-based file system, in: 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2014.
[44] Richard Stallman, Roland Pesch, Stan Shebs, et al., Debugging with GDB, Free Software Foundation, 2002.
[45] Brent Welch, Geoffrey Noer, Optimizing a hybrid SSD/HDD HPC storage system based on file size distributions, in: 2013 IEEE 29th Symposium on Mass Storage Systems and Technologies, MSST, 2013.
[46] NVM Express, 2016, https://nvmexpress.org/.
[47] Computeexpresslink(CXL), https://www.computeexpresslink.org/.
[48] Matias Bjorling, Jens Axboe, David Nellans, Philippe Bonnet, Linux Block IO: Introducing multi-queue SSD access on multi-core systems, in: Proceedings of the 6th International Systems and Storage Conference, SYSTOR, 2013.
[49] Changman Lee, Dongho Sim, Joo-Young Hwang, Sangyeun Cho, F2FS: A new file system for flash storage, in: Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST, 2015.
[50] Jian Xu, Steven Swanson, NOVA: A log-structured file system for hybrid volatile/non-volatile main memories, in: 14th USENIX Conference on File and Storage Technologies, FAST, 2016.
[51] Vaibhav Gogte, William Wang, Stephan Diestelhorst, Aasheesh Kolli, Peter M. Chen, Satish Narayanasamy, Thomas F. Wenisch, Software wear management for persistent memories, in: 17th USENIX Conference on File and Storage Technologies, FAST, 2019.
[52] Libata: add TRIM support, 2009, https://lwn.net/Articles/362108/, November 15.
[53] Duo Zhang, Om Rameshwar Gatla, Wei Xu, Mai Zheng, A study of persistent memory bugs in the linux kernel, in: Proceedings of the 14th ACM International Systems and Storage Conference, SYSTOR, 2021.
[54] Jinrui Cao, Om Rameshwar Gatla, Mai Zheng, Dong Dai, Vidya Eswarappa, Yan Mu, Yong Chen, PFault: A general framework for analyzing the reliability of high-performance parallel file systems, in: Proceedings of the 2018 International Conference on Supercomputing, ICS, 2018.
[55] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W Zhao, Elizabeth S. Yang, Reliability analysis of SSDs under power fault, ACM Trans. Comput. Syst. (TOCS) (2016).
[56] Stathis Maneas, Kaveh Mahdaviani, Tim Emami, Bianca Schroeder, A study of SSD reliability in large scale enterprise storage deployments, in: 18th USENIX Conference on File and Storage Technologies, FAST, 2020.
[57] Seulbae Kim, Meng Xu, Sanidhya Kashyap, Jungyeon Yoon, Wen Xu, Taesoo Kim, Finding semantic bugs in file systems with an extensible fuzzing framework, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP, 2019.
[58] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, Vijay Chidambaram, Finding crash-consistency bugs with bounded black-box crash testing, in: 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2018.
[59] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Understanding the robustness of SSDs under power fault, in: Proceedings of the 11th USENIX Conference on File and Storage Technologies, FAST, 2013.
[60] Mai Zheng, Joseph Tucek, Dachuan Huang, Feng Qin, Mark Lillibridge, Elizabeth S. Yang, Bill W Zhao, Shashank Singh, Torturing databases for fun and profit, in: 11th USENIX Symposium on Operating Systems Design and Implementation, OSDI, 2014.
[61] Om Rameshwar Gatla, Muhammad Hameed, Mai Zheng, Viacheslav Dubeyko, Adam Manzanares, Filip Blagojević, Cyril Guyot, Robert Mateescu, Towards robust file system checkers, in: 16th USENIX Conference on File and Storage Technologies, FAST, 2018.
[62] GDB: The GNU Project Debugger, https://www.gnu.org/software/gdb/.
[63] Peter A. Buhr, Martin Karsten, Jun Shih, KDB: a multi-threaded debugger for multi-threaded applications, in: Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, SPDT, 1996.
[64] Kgdb, https://elinux.org/Kgdb.
[65] lTTng, https://lttng.org/.
[66] SystemTap, https://sourceware.org/systemtap/.
[67] XRay: A function call tracing system, https://research.google/pubs/pub45287/.
[68] An introduction to KProbes, https://lwn.net/Articles/132196/.
[69] Dtrace, http://dtrace.org/blogs/.
[70] SCSI Commands Reference Manual by Seagate, https://www.seagate.com/files/staticfiles/support/docs/manual/Interface%20manuals/100293068j.pdf.
[71] Samuel T. King, George W. Dunlap, Peter M. Chen, Debugging operating systems with time-traveling virtual machines, in: Proceedings of the 2005 USENIX Technical Conference, 2005.
[72] Kernel bugzilla, https://www.bugzilla.org/.
[73] Shan Lu, Soyeon Park, Eunsoo Seo, Yuanyuan Zhou, Learning from mistakes: a comprehensive study on real world concurrency bug characteristics, in: Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2008.
[74] Haryadi S. Gunawi, Thanh Do, Agung Laksono, Mingzhe Hao, Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Riza O. Suminto, What bugs live in the cloud? A study of issues in scalable distributed systems, in: Proceedings of the ACM Symposium on Cloud Computing, SOCC, 2014.
[75] Lanyue Lu, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Shan Lu, A study of linux file system evolution, in: Proceedings of the 11th USENIX Conference on File and Storage Technologies, FAST, 2013.
[76] Fabrice Bellard, QEMU, a fast and portable dynamic translator, in: USENIX Annual Technical Conference, FREENIX Track, 2005.
[77] The LLVM Compiler Infrastructure, https://llvm.org/.
[78] Linux SCSI target framework (tgt), http://stgt.sourceforge.net/.
[79] Catherine Paganini, Danyel Fisher, Franciss Espenido, Gabriel H. Dinh, Heather Joslyn, Jason Morgan, Joab Jackson, Judy Williams, Libby Clark, Peter Putz, Steve Tidwell, Susan Hall, Cloud native observability for Devsops teams, in: The New Stack, 2021.
[80] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, An analysis of persistent memory use with WHISPER, in: Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS, 2017.
[81] Yahoo! Cloud Serving Benchmark, https://en.wikipedia.org/wiki/YCSB.
[82] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, DawsonEngle, An empirical study of operating systems errors, in: Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles, SOSP, 2001.
[83] David Lazar, Haogang Chen, Xi Wang, Nickolai Zeldovich, Why does cryptographic software fail? A case study and open problems, in: Proceedings of the Second Asia-Pacific Workshop on Systems, APSys, 2014.
[84] Haogang Chen, Yandong Mao, Xi WangDong Zhou, Nickolai Zeldovich, M. Frans Kaashoek, Linux kernel vulnerabilities:State-of-the-art defenses and open problem, in: Proceedings of the Second Asia-Pacific Workshop on Systems, APSys, 2014.
[85] Haryadi S. Gunawi, Mingzhe Hao, Riza O. Suminto, Agung Laksono, Anang D. Satria, Jeffry Adityatama, Kurnia J. Eliazar, Why does the cloud stop computing? Lessons from hundreds of service outages, in: Proceedings of the ACM Symposium on Cloud Computing, SOCC, 2016.
[86] Haopeng Liu, Shan Lu, Madan Musuvathi, Suman Nath, What bugs cause production cloud incidents? in: Proceedings of the Workshop on Hot Topics in Operating Systems, HotOS, 2019.
[87] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, Bianca Schroeder, An analysis of data corruption in the storage stack, Trans. Storage (2008).
[88] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, Jiri Schindler, An analysis of latent sector errors in disk drives, in: Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, 2007.
[89] Bianca Schroeder, Garth A. Gibson, Disk failures in the real world: What does an MTTF of 1, 000, 000 hours mean to you? in: Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST), 2007.
[90] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark Manasse, Rina Panigrahy, Design tradeoffs for SSD performance, in: USENIX 2008 Annual Technical Conference, ATC, 2008.

[91] H Kurata, K Otsuga, A Kotabe, S Kajiyama, T Osabe, Y Sasago, S Narumi, K Tokami, S Kamohara, O Tsuchiya, The impact of random telegraph signals on the scaling of multilevel Flash memories, in: 2006 Symposium on VLSI Circuits, 2006.

[92] Jiangpeng Li, Kai Zhao, Xuebin Zhang, Jun Ma, Ming Zhao, Tong Zhang, How much can data compressibility help to improve NAND flash memory lifetime? in: Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST, 2015.

[93] Bianca Schroeder, Raghav Lagisetty, Arif Merchant, Flash reliability in production: The expected and the unexpected, in: 14th USENIX Conference on File and Storage Technologies, FAST, 2016.

[94] Huang-Wei Tseng, Laura M. Grupp, Steven Swanson, Understanding the impact of power loss on flash memory, in: Proceedings of the 48th Design Automation Conference, DAC, 2011.

[95] Erci Xu, Mai Zheng, Feng Qin, Yikang Xu, Jiesheng Wu, Lessons and actions: What we learned from 10K SSD-related storage system failures, in: Proceedings of the 2019 USENIX Annual Technical Conference, ATC, 2019.

[96] Erci Xu, Mai Zheng, Feng Qin, Jiesheng Wu, Yikang Xu, Understanding SSD Reliability in Large-scale Cloud Systems, in: Proceedings of the 3rd IEEE/ACM International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, PDSW, 2018.

[97] Changwoo Min, Sanidhya Kashyap, Byoungyoung Lee, Chengyu Song, Taesoo Kim, Cross-checking semantic correctness: The case of finding file system bugs, in: Proceedings of the 25th Symposium on Operating Systems Principles, 2015.

[98] Jinrui Cao, Simeng Wang, Dong Dai, Mai Zheng, Yong Chen, A generic framework for testing parallel file systems, in: Proceedings of the 1st Joint International Workshop on Parallel Data Storage & Data Intensive Scalable Computing Systems, PDSW, 2016.

[99] Om Rameshwar Gatla, Mai Zheng, Understanding the fault resilience of file system checkers, in: 9th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage, 2017.

[100] Runzhou Han, Om Rameshwar Gatla, Mai Zheng, Jinrui Cao, Di Zhang, Dong Dai, Yong Chen, Jonathan Cook, A study of failure recovery and logging of high-performance parallel file systems, in: ACM Transactions on Storage TOS, 2021.

[101] Runzhou Han, Duo Zhang, Mai Zheng, Fingerprinting the checker policies of parallel file systems, in: Proceedings of the 5th ACM/IEEE International Parallel Data Systems Workshop, PDSW, 2020.

BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100007

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

A parallel sparse approximate inverse preconditioning algorithm based on MPI and CUDA☆

Yizhou Wang, Wenhao Li, Jiaquan Gao *

Jiangsu Key Laboratory for NSLSCS, School of Computer and Electronic Information, Nanjing Normal University, Nanjing, 210023, China

ARTICLE INFO

Keywords:
Sparse approximate inverse
Preconditioning
CUDA
GPU
MPI

ABSTRACT

In this study, we present an efficient parallel sparse approximate inverse (SPAI) preconditioning algorithm based on MPI and CUDA, called HybridSPAI. For HybridSPAI, it optimizes a latest static SPAI preconditioning algorithm, and is extended from one GPU to multiple GPUs in order to process large-scale matrices. We make the following significant contributions: (1) a general parallel framework for optimizing the static SPAI preconditioner based on MPI and CUDA is presented, and (2) for each component of the preconditioner, a decision tree is established to choose the optimal kernel of computing it. Experimental results show that HybridSPAI is effective, and outperforms the popular preconditioning algorithms in two public libraries, and a latest parallel SPAI preconditioning algorithm.

1. Introduction

It has proved that sparse approximate inverse (SPAI) preconditioners can effectively accelerate the convergence rate of Krylov subspace methods, such as the generalized minimal residual method (GMRES) [1] and the biconjugate gradient stabilized method (BiCGSTAB) [2]. Moreover, compared with the incomplete factorization preconditioners [3–6] and the factorized sparse approximate inverse (FSAI) preconditioners [7–10], SPAI preconditioners neither require excessively sparse matrix–vector multiplication operations nor take care of the risk of breakdowns that can be encountered by FSAI preconditioners [11]. Consequently, SPAI preconditioners have attracted much attention [12–17].

In recent years, graphic processing units (GPUs) have become an important resource for scientific computing because of their many core structures and powerful computation efficiency, and have been used as tools for high-performance computation in a lot of fields [18–21]. As we know, the cost of constructing SPAI preconditioners is commonly very expensive for large-scale matrices, because the memory requirements to store them, and the computation requirements to calculate them are approximately the scale with the square to third power of the number of nonzeros in each row.

With the emerging of graphic processing units (GPUs), many studies have been conducted to accelerate the construction of SPAI preconditioners on the GPU architecture, and many parallel preconditioning algorithms [11,22–26] are proposed. Based on the degree of freedom used, SPAI preconditioner generation is classified as static (a priori) or adaptive. In this paper, we focus on optimizing a latest static SPAI preconditioning algorithm and extend it from one GPU to multiple GPUs. There has existed some work about static SPAI preconditioners on GPU [11,27], but the detailed implementations never be given and the source code is not public. Furthermore, He and Gao et al. propose two static SPAI preconditioning algorithms on GPU, called SPAI-Adaptive [28] and GSPAI-Adaptive [29], and give their implementation details. The two algorithms are verified to be effective for large-scale matrices. In this study, inspired by Gao's work, we further investigate how to highly optimize the static SPAI on multi-GPUs instead of only single GPU in this paper. We propose an optimized SPAI preconditioning algorithm based on MPI and CUDA, called HybridSPAI. Compared to a latest static SPAI preconditioning algorithm, the proposed algorithm has the following distinct characteristics. First, a general parallel framework based on MPI and CUDA is presented to optimize the static SPAI preconditioner, and is extended from one GPU to multiple GPUs. For each GPU, it operates same procedures as shown in Section 3.3, such as finding indices I and J, constructing the local submatrix, decomposing the local submatrix into QR, and solving the upper triangular linear systems. For MPI, it provides a simple and easy-to-use parallel controlling capability on multicore CPUs, which dedicates one thread for controlling one GPU. Second, when a sparsity pattern of the preconditioner is given, we use the thread-adaptive allocation strategy to choose the optimized number of threads for each column of the preconditioner, and construct the decision tree to choose the optimization kernel to calculate each one of components

☆ The research has been supported by the Natural Science Foundation of China under grant number 61872422, and the Natural Science Foundation of Jiangsu Province, China under grant number BK20171480.

* Corresponding author.
E-mail addresses: 1966224230@qq.com (Y. Wang), 917339495@qq.com (W. Li), springf12@163.com (J. Gao).

https://doi.org/10.1016/j.tbench.2021.100007
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 6 November 2021

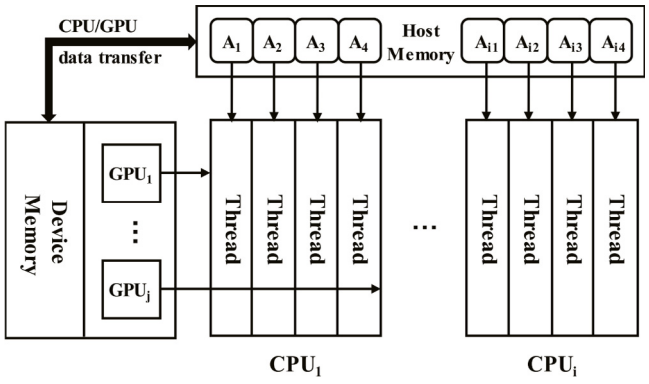


Fig. 1. A CPU–GPU hybrid parallel computing model based on MPI.

of the preconditioner. Experimental results show that HybridSPAI is effective, and is advantageous over the popular incomplete LU factorization algorithm in the CUSPARSE library [30], the static SPAI preconditioning algorithm in the ViennaCL library [24], and the latest GSPAI-Adaptive [29].

The main contributions in this paper are summarized as follows.

- A general parallel framework based on MPI and CUDA is presented for optimizing the static SPAI preconditioner, and is extended from one GPU to multiple GPUs, also the CPU and GPU tasks are designated.
- A strategy is presented to choose the optimal number of threads for each column of the preconditioner.
- On the basis of the parallel framework and proposed strategy, an optimization SPAI preconditioning algorithm based on MPI and CUDA, called HybridSPAI, is presented. In HybridSPAI, finding indices, constructing local submatrix, decomposing the local submatrix into *QR*, and solving the upper triangular linear system are computed in parallel, and the kernels of calculating them are selected by the decision tree optimization.

The rest of this paper is organized as follows. Section 2 describes the SPAI preconditioning algorithm, Section 3 gives the detailed implementation of HybridSPAI, Section 4 presents the experimental analysis and evaluation, and Section 5 contains our conclusions and points to our future research directions.

2. SPAI algorithm

The basic idea of the SPAI procedure [22] is described as follows: Use a sparse matrix M, known as the preconditioner, to approximate the inverse of A, and M is computed by the following formula:

$$\min \|AM - E\|_F^2. \tag{1}$$

Owing to the independence of the columns of M, the equation mentioned above can be separated into the following n independent least squares problems

$$\min_{m_k} \left\|Am_k - e_k\right\|_2^2, \quad k = 1, 2, \ldots, n \tag{2}$$

where e_k is the kth column of the identity matrix and m_k represents column k in matrix M. For a description of the implementation details of SPAI, we refer to the literature [27].

3. Optimizing SPAI on GPUs

We present an optimization SPAI preconditioning algorithm based on CPU–GPU platforms, called HybridSPAI. The hybrid parallel computing model is illustrated in Fig. 1. and the parallel framework of HybridSPAI is shown in Fig. 2, which includes the following three stages: *Pre-HybridSPAI* stage, *Compute-HybridSPAI* stage, and *Post-HybridSPAI* stage.

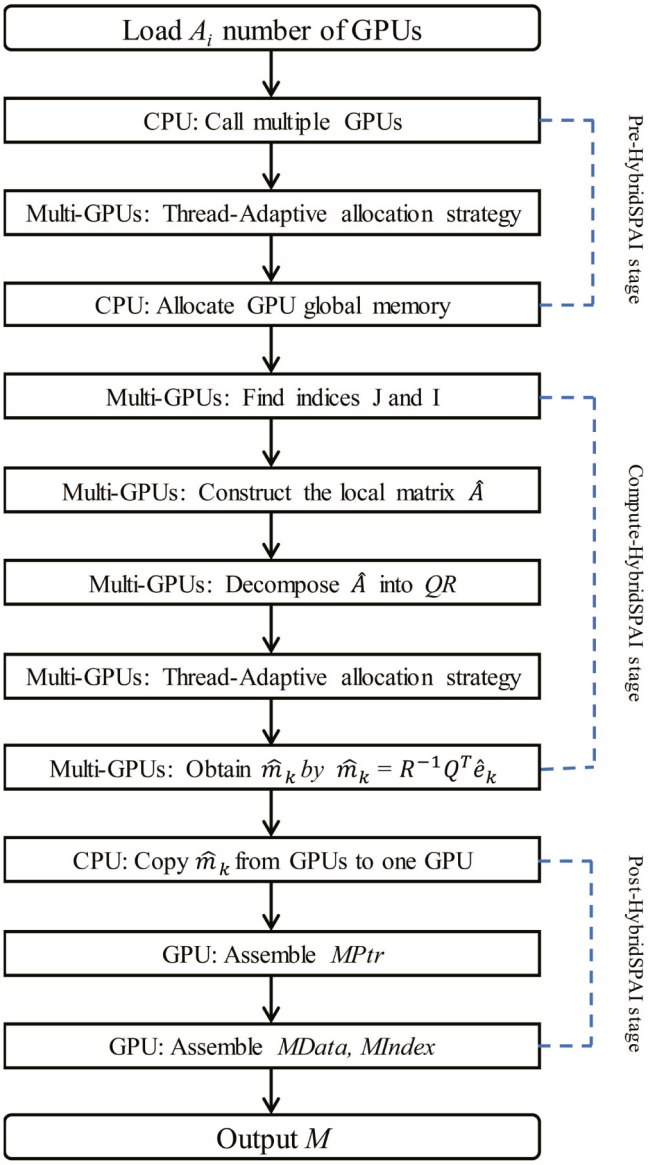


Fig. 2. Parallel framework of HybridSPAI.

3.1. Hybrid parallel programming based on MPI and CUDA

A hybrid parallel programming model must be designed for the architectures of GPU and CPU to improve the computing performance, and has the characteristics of extending to more devices. In our proposed model, as a device in CUDA, GPU can be controlled by each thread of multicore CPU, also can be controlled by each individual CPU. In addition, the data is transferred from the host memory to the GPU device memory, then the CPU launches the calculation process on the GPU by calling the kernel function.

MPI provides a simple and convenient parallel computing capability of multi-threads on multicore CPUs [31]. The hybrid parallel computing model is illustrated in Fig. 1, where A_1, A_2, …, A_{i3}, A_{i4},are submatrices which are stored in the host memory, and *Thread* are multi-threads which are assigned to cores of CPUs.

Note that when using this model, a computing matrix will be divided into multiple submatrices which corresponding with the number of calling threads of CPUs, so that these submatrices are assigned to each GPU to perform respectively.

Table 1
Arrays in HybridSPAI.

Array	Size	Type
AData	nonzeros	Double
AIndex	nonzeros	Integer
APtr	n	Integer
I	$ns \times n1max$	Integer
atmoic	n	Integer
J	$ns \times n2max$	Integer
jPTR	ns	Integer
$\hat{m}$	$ns \times n2max$	Double
$\hat{A}$	$ns \times n1max \times n2max$	Double
R	$ns \times n1max \times n2max$	Double
iPTR	ns	Integer

3.2. Pre-HybridSPAI Stage

In this paper, we summarize the sparsity of M in advance with the main method in [25]. *M(i,j)* is considered a nonzero if

$$|A(i,j)| > (1-\tau)\max_j |A(i,j)|, 0 \leqslant \tau \leqslant 1 \tag{3}$$

is satisfied, where τ is a user defined tolerance parameter (the main diagonal is always included).

Next, A is stored in host memory using the compressed sparse column(CSC) storage format, and M is also stored in columns. The dimensions of local submatrices ($n1_k$, $n2_k$) are usually distinct for different k, ($k = 1, 2, \ldots$,n). To simplify the accesses of data in memory and increasing the coalescence, the dimensions of all local submatrices are uniformly defined as ($n1max$, $n2max$), where $n1max = \max_k\{n1_k\}$ and $n2max = \max_k\{n2_k\}$.

Finally, the thread-adaptive allocation strategy is proposed. For any matrix, the number of threads z for each column of the preconditioner is calculated by the following formulas:

$$z = \min\left(2^l, nt\right) \tag{4}$$

$$\text{s.t. } 2^{l-1} < n2\ max \leqslant 2^l. \tag{5}$$

In Eqs. (4), nt is a fixed thread block size. z threads are grouped into a thread group, which is assigned to compute the kth column of M. The lowercase "L" in the Eqs. (4) was required to compute the suitable z threads. Note that we used a 1D array of the thread blocks to organize the compute grid in this paper, and used a 1D array of threads to organize the thread block as well.

3.3. Compute-HybridSPAI Stage

In the Compute-HybridSPAI stage, the allocations of every GPU global memory are shown in Table 1. Based on the characteristics of message interface, MPI is very convenient to scatter and gather data between the multiple threads of CPU. The following steps are implemented to compute M.

Finding J and I: In all blocks, each thread-group block size that is used to find J and I is same, and each thread group (warpSize threads) is assigned to find one subset of J and I, which making many subsets of J and I can be simultaneously obtained. Furthermore, parallelism is also exploited inside each thread group. For the kernel that finds J, the threads inside each warp (thread group) read one column of M in parallel, and store them to shared memory using atomic operation. For the kernel that finds I, a decision tree is established and for any given $n2max$ and $n1max$, this optimized kernel can be effective. Fig. 3 shows a segment of the decision tree for finding I. When 4 < n2max ≤ 8, cuFindIBySharedMemory kernel with shared memory of *sharedSize* size or cuFindI kernel with global memory will be selected according to different the $n1max$. Here *sharedSize* = number of computing columns of the preconditioner ×upper boundary closest to $n1max$. Fig. 4 shows the main procedure of cuFindIBySharedMemory kernel. Each thread group finds one subset of I, e.g., I_k. First, the row indices of the first column referenced in one subset of J, e.g., J_k are loaded to shared memory sI with the threads in the thread group. Then the index vectors of successive columns referenced by J_k are compared in parallel with values in sI and new indices are appended to sI by utilizing the atomic operations. Second, inside the thread group, the indices of sI are sorted in ascending order in parallel. Finally, the indices of sI are copied to I_k. When $n1max > 256$, cuFindI kernel is executed on global memory instead of shared memory, which is similar to cuFindIBySharedMemory kernel.

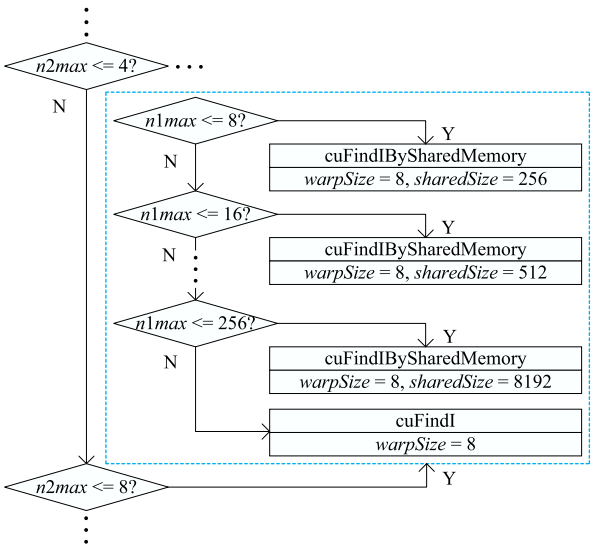


Fig. 3. A segment of the decision tree of find I.

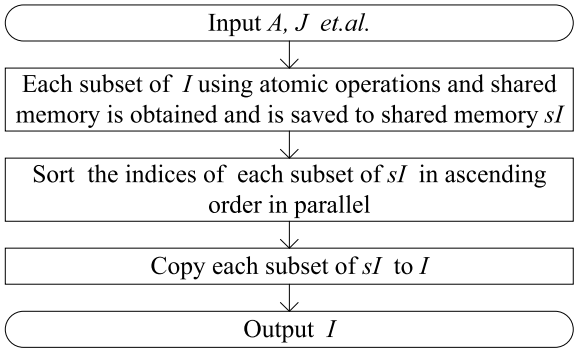


Fig. 4. Main procedure of cuFindIBySharedMemory kernel.

Constructing the local submatrix: Using J and I obtained above, the local matrix set $\hat{A}$, is computed by kernel with shared memory or kernel with global memory according to the established decision tree. Fig. 5 shows a segment of the decision tree for constructing $\hat{A}$. When 4 < n2max ≤ 8, cuComputeTildeABySharedMemory kernel with shared memory of *sharedSize* size or cuComputeTildeA kernel with global memory will be selected according to different $n1max$. Fig. 6 shows the main procedure of cuComputeTildeABySharedMemory kernel. For the thread group on each GPU that calculates $\widehat{A_k}$, all threads in the thread group first read values in I_k into shared memory sI in parallel, and $\widehat{A_k}$ is constructed on global memory by loading columns indexed in J_k and matching them to I_k in parallel. When $n1max > 256$, cuComputeTildeA kernel is executed on global memory instead of shared memory, which is similar to cuComputeTildeABySharedMemory kernel.

Decomposing the Local Submatrix into *QR*:The thread-group size of decomposing the local submatrix into *QR* is same in all blocks. Being similar with above two steps, the constructed decision tree is used again

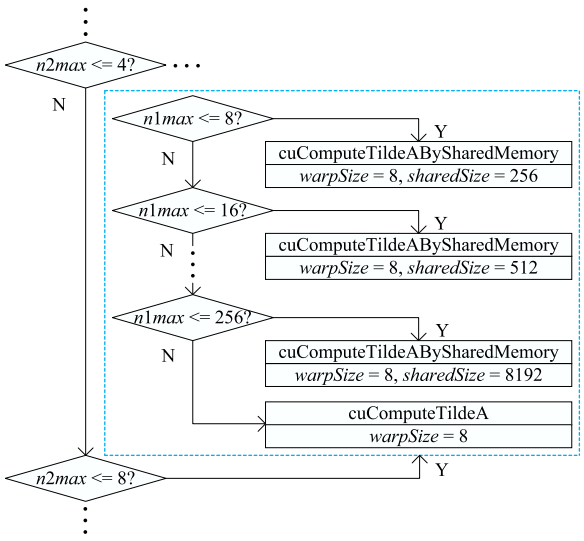


Fig. 5. A segment of the decision tree to construct $\hat{A}$.

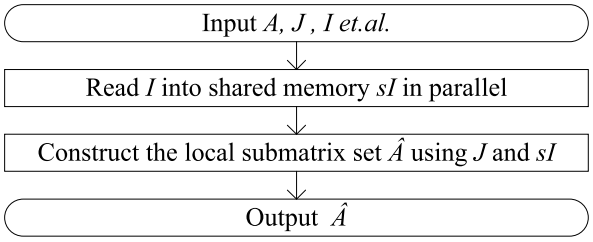


Fig. 6. Main procedure of cuComputeTildeABySharedMemory kernel.

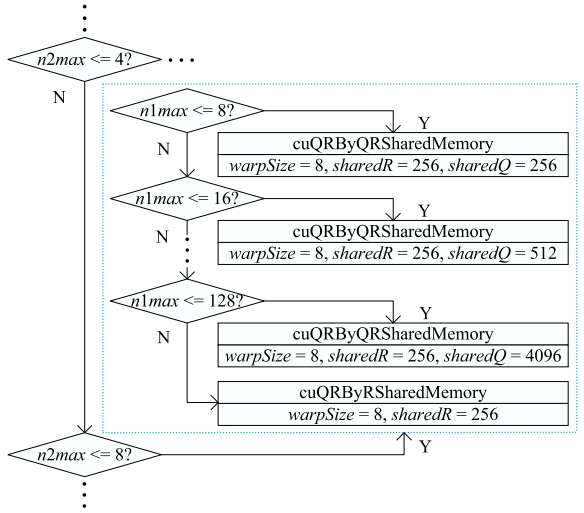


Fig. 7. A segment of the decision tree to decompose the local submatrix into *QR*.

to decompose local submatrix. Fig. 7 shows a segment of the decision tree for decomposing the local submatrix into *QR*. When 4 < n2max ≤ 8, cuQRByQRSharedMemory kernel with shared memory of *sharedSize* size and *sharedQ* size or cuQRByRSharedMemory kernel with shared memory of *sharedR* size will be selected according to different *n1max*. Fig. 8 shows the main procedure of cuQRByQRSharedMemory kernel. In addition, Eeach thread group is responsible for one *QR* decomposition. For a description of its detailed implementation, please refer to the literature [25]. In a thread group, the local submatrix, e.g., $\widehat{A_k}$, is decomposed into *QR* by the following four steps at each iteration *i*. In

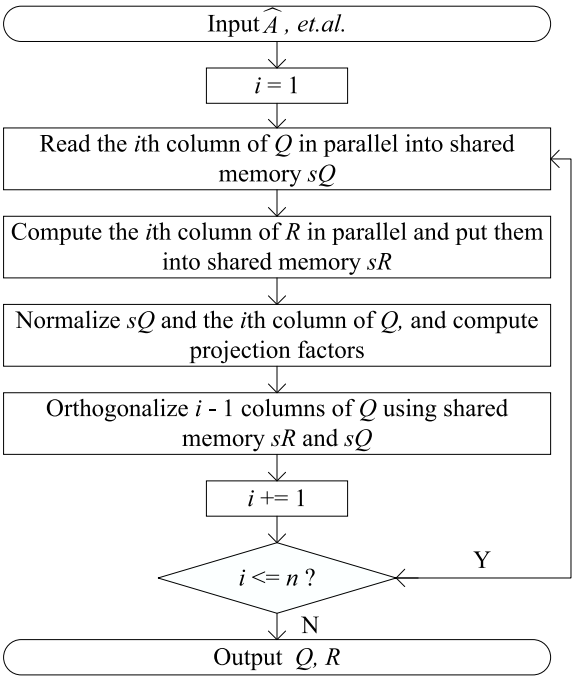


Fig. 8. Main procedure of cuQRByQRSharedMemory kernel.

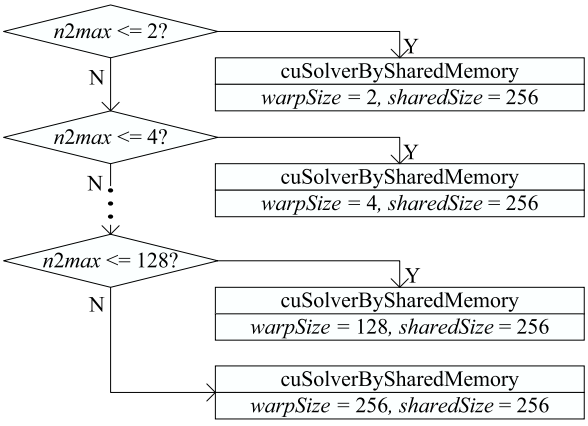


Fig. 9. A segment of the decision tree to solve the upper triangular linear system.

the first step, the *i*th column of Q_k are read into shared memory *sQ* in parallel. In the second step, the threads computed the *i*th row of the upper triangle matrix R_k in parallel and put into shared memory *sR*. In the third step, the column *i* of Q_k and *sQ* are concurrently normalized, and the projection factors R_k and *sR* are calculated. In the fourth step, the values of all columns of Q_k are updated by using shared memory *sQ* and *sR* in parallel. When *n1max* > 128. cuQRByRSharedMemory kernel is executed by utilizing shared memory *sR* instead of shared memory *sQ*, which is similar to cuQRByQRSharedMemory kernel.

Solving the Upper Triangular Linear System: In this section, one subset of $\hat{m_k} = R_k^{-1} Q_k^T \hat{e_k}$ are computed by solving an upper triangular linear system. Fig. 9 shows a segment of the decision tree for solving an upper triangular linear system. For any given *n2max* value, cuSolverBySharedMemory with shared memory of 256 size and thread-group size of *warpSize*, is chosen. For example, when 4 < n2max ≤ 8, cuSolverBySharedMemory kernel with shared memory of 256 size and thread-group size of 8 is selected. Fig. 10 shows the main procedure of cuSolverBySharedMemory kernel. For each thread group,T the steps to compute $\hat{m}$, e.g., $\widehat{m_k}$, include: (1) $Q_k^T \hat{e_k}$ is calculated in parallel and saved to the shared memory xE, and (2) the values of $\hat{m_k}$ are obtained

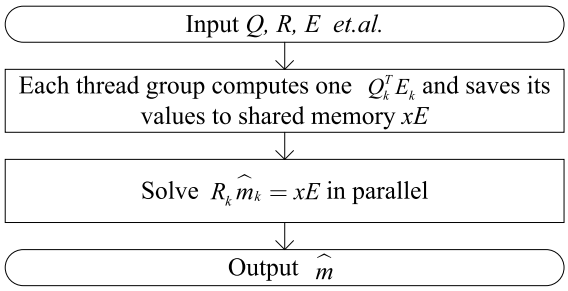


Fig. 10. Main procedure of cuSolverBySharedMemory kernel.

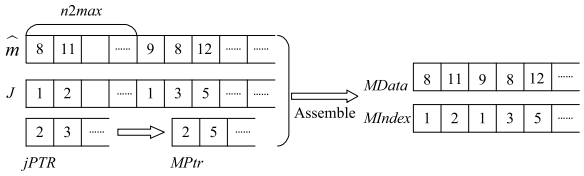


Fig. 11. Assemble *M*.

Table 2
Descriptions of test matrices.

Name	Kind	Rows	Nonzeros	*avg*	*max*	*min*
venkat01	CFD sequence	62,424	1,717,792	27.52	44	16
imagesensor	Semiconductor device	118,758	1,446,396	12.18	21	2
cfd2	CFDproblem	123,440	3,085,406	25.00	30	8
apache2	Structural	715,176	4,817,870	6.74	8	4
t2em	Electronmagnetics	921,632	4,590,832	4.98	5	1
thermal2	Thermal	1,228,045	8,580,313	6.99	11	1
G3_circuit	Circuitsimulation	1,585,478	7,660,826	4.83	6	2

by solving the upper triangular linear system, $R_k \hat{m_k} = xE$, in parallel using shared memory.

3.4. Post-HybridSPAI Stage

The Post-HybridSPAI Stage is to assemble *M* in the CSC storage format from multiple GPUs. Fig. 11 illustrates the procedure of assembling *MPtr*, *MIndex* and *MData* arrays on each GPU. First, *MPtr* is assembled utilizing *jPTR*. Second, $\hat{m}$ and *J* are utilized to assemble *MIndex* and *MData*. Finally, *MData* arrays on each GPU are transferred to the respective threads of CPU according to the device ID of GPUs. On the CPU, each thread utilize the function MPI_Gatherv() of MPI to gather the *MData* into a complete array in parallel.

4. Evaluation and analysis

We evaluate the performance of HybridSPAI in this section. The test matrices in Table 2 are used to evaluate the performance of NVIDIA GTX 1080 Ti GPUs, which are selected from University of Florida Sparse Matrix Collection. The source codes are compiled and executed using the CUDA toolkit 10.1.

4.1. Effectiveness analysis

For each test matrix, GPUPBICGSTAB are called to solve *Ax=b* on GTX 1080 Ti, where all values of *b* are 1 and the produced *M* is used as the preconditioner. They stop when the residual error is less than $1e^{-7}$, or the number of iterations exceeds 10,000. Table 3 shows the results, and the time unit is second (*s*).

In addition, we take GTX 1080 Ti to investigate the effort of single GPU and increasing the number of *threads* on the execution time of HybridSPAI and GPUPBICGSTAB with HybridSPAI. Table 4 demonstrates

Table 3
Iterations and execution time of GPUBICGSTAB on GTX 1080 Ti.

Matrix	GPUBICGSTAB		GPUPBICGSTAB	
	Iterations	Execution time	Iterations	Execution time
venkat01	10000	/	35	1.312
imagesensor	10000	/	52	1.036
cfd2	7768	5.167	1613	3.518
apache2	5813	8.061	1106	3.032
t2em	1661	3.122	768	2.338
thermal2	4095	9.771	2584	9.748
G3_circuit	10000	/	475	2.53

Table 4
Execution time of HybridSPAI and GPUPBICGSTAB.

Matrix	GPU	1 thread	2 thread	4 thread	8 thread
venkat01	0.506	0.501	0.262	0.151	0.102
	0.806	0.771	0.761	0.736	0.715
	1.312	1.272	1.023	0.887	0.817
imagesensor	0.228	0.227	0.179	0.103	0.104
	0.808	0.785	0.767	0.745	0.713
	1.036	1.012	0.946	0.848	0.817
cfd2	1.187	1.191	0.631	0.356	0.224
	2.331	2.231	2.294	2.178	2.101
	3.518	3.422	2.925	2.534	2.325
apache2	0.226	0.219	0.126	0.101	0.133
	2.806	2.761	2.746	2.734	2.838
	3.032	2.980	2.872	2.835	2.971
t2em	0.075	0.070	0.060	0.064	0.103
	2.263	2.253	2.241	2.231	2.268
	2.338	2.323	2.301	2.295	2.371
thermal2	0.332	0.329	0.201	0.165	0.164
	9.416	9.443	9.367	9.369	9.187
	9.748	9.772	9.568	9.534	9.351
G3_circuit	0.167	0.156	0.113	0.094	0.115
	2.363	2.302	2.321	2.329	2.290
	2.530	2.458	2.434	2.423	2.405

the execution time of this. For each matrix and given number of *threads*, the first row and second row are respectively the computing time of HybridSPAI and GPUPBICGSTAB, and the third row is the sum of time of the first two row. GPUPBICGSTAB stops while the residual error is less than $1e^{-7}$. The minimum values of the second and third rows for each matrix both are marked in the red font. In addition, we observe that when the time of computing the preconditioner keeps less than 228 ms on single GPU, increasing the number of GPU cannot provide significant acceleration.

4.2. Performance comparison

We test the HybridSPAI performance by comparing it with a popular preconditioning algorithms: CSRILU0 in CUSPARSE (denoted by CSRILU) [32], a static sparse approximate inverse preconditioning algorithm in ViennaCL (denoted by SSPAI-VCL) [33], and a latest parallel SPAI preconditioning algorithm(denoted by GSPAI-Adaptive) [29]. Table 5 demonstrate the comparison results on GTX 1080 Ti GPUs. For each matrix and the preconditioner, the first row is the computing time of these four preconditioning algorithms, and the second row and the third row are respectively the execution time and the number of iterations of GPUPBICGSTAB while the residual error is less than $1e^{-7}$. Note that "/" represents the number of iterations for HybridSPAI exceeds 10,000, and all the other rows for each matrix will be denoted except that the third row is denoted by "> 10000". The minimum value of the fourth row for each matrix is marked in the red font.

From Table 5, we observe that on GTX 1080 Ti, the total time of HybridSPAI and GPUPBICGSTAB with HybridSPAI is the smallest among all algorithms for any matrices. This displays that HybridSPAI outperforms CSRILU and SSPAI-VCL, and is advantageous over GSPAI-Adaptive.

Table 5
Execution time of all preconditioning algorithms and GPUPBICGSTAB on GTX 1080 Ti.

Matrix	CSRILU	SSPAI-VCL	GSPAI-Adaptive	HybridSPAI
venkat01	1.835	38.856	0.506	0.102
	1.574	0.036	0.806	0.715
	11	48	35	35
	3.427	38.892	1.312	0.817
imagesensor	/	/	0.228	0.104
	/	/	0.808	0.713
	10000	10000	52	52
	/	/	1.036	0.817
cfd2	/	/	1.187	0.224
	/	/	2.331	2.101
	10000	10000	1613	1613
	/	/	3.518	2.325
apache2	3.386	43.532	0.226	0.101
	6.776	2.995	2.806	2.734
	475	2503	1106	1106
	10.162	46.527	3.032	2.835
t2em	19.884	/	0.075	0.064
	2998.63	/	2.263	2.231
	427	10000	768	768
	3018.514	/	2.338	2.295
thermal2	5.502	/	0.332	0.164
	45.008	/	9.416	9.187
	1619	10000	2584	2584
	50.510	/	9.748	9.351
G3_circuit	5.245	/	0.167	0.115
	12.475	/	2.363	2.290
	257	10000	475	475
	17.720	/	2.530	2.405

5. Conclusion

We present an efficient parallel sparse approximate inverse preconditioning algorithm on multi-GPUs in this paper, which is based MPI and CUDA, called HybridSPAI. In our proposed HybridSPAI, a general parallel framework is embraced for optimizing the static SPAI on multi-GPUs, and a decision tree is established to choose the optimal kernel for computing it. The experimental results demonstrate a noticeable performance and high effectiveness of our proposed HybridSPAI.

References

[1] Y. Saad, M.H. Schultz, GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems, SIAM J. Sci. Stat. Comput. 7 (1986) 856–869.
[2] H.A: Bi-CGSTAB: a fast and smoothly converging variant of BiCG for the solution of nonsymmetric linear systems, SIAM J. Stat. Comput. 13 (2) (1992) 631.
[3] Y. Saad, Iterative Methods for Sparse Linear Systems, second version, SIAM, Philadelphia, PA, 2003.
[4] J. Gao, R. Liang, J. Wang, Research on the conjugate gradient algorithm with a modified incomplete cholesky preconditioner on GPU, J. Parallel Distr. Com. 74 (2) (2014) 2088–2098.
[5] S.C. Rennich, D. Stosic, T.A. Davis, Accelerating sparse Cholesky factorization on GPUs, Parallel Comput. 59 (2016) 140–150.
[6] H. Anzt, M. Gates, J. Dongarra, M. Kreutzer, G. Wellein, M. Kohler, Preconditioned Krylov solvers on GPUs, Parallel Comput. 68 (2017) 32–44.
[7] L.Y. Kolotilina, A.Y. Yeremin, Factorized sparse approximate inverse preconditioning I. theory, SIAM J. Matrix Anal. Appl. 14 (1) (1993) 45–58.
[8] M. Benzi, C.D. Meyer, M. Tuma, A sparse approximate inverse preconditioner for the conjugate gradient method, SIAM J. Sci. Comput. 17 (5) (1996) 1135–1149.
[9] M. Ferronato, C. Janna, G. Pini, A generalized block FSAI preconditioner for nonsymmetric linear systems, J. Comput. Appl. Math. 256 (2014) 230–241.
[10] V.A.P. Magri, A. Franceschini, M. Ferronato, C. Janna, Multilevel approaches for FSAI preconditioning, Numer. Linear Algebr. (2018) e2183, http://dx.doi.org/10.1002/nla.2183.
[11] M.M. Dehnavi, D.M. Fern´andez, J.L. Gaudiot, D.D. Giannacopoulos, Parallel sparse approximate inverse preconditioning on graphic processing units, IEEE T. Parall. Distr. 24 (9) (2013) 1852–1861.
[12] Z. Jia, B. Zhu, A power sparse approximate inverse preconditioning procedure for large sparse linear systems, Numer. Linear Algebr. 16 (4) (2009) 259–299.
[13] J.D.F. Cosgrove, J.C. Diaz, A. Griewank, Approximate inverse preconditioning for sparse linear systems, Int. J. Comput. Math. 44 (1–2) (1992) 91–110.
[14] M. Grote, T. Huckle, Parallel preconditioning with sparse approximate inverses, SIAM J. Sci. Comput. 18 (3) (1997) 838–853.
[15] E. Chow, Y. Saad, Approximate inverse preconditioners via sparse-sparse iterations, SIAM J. Sci. Comput. 19 (3) (1998) 995–1023.
[16] E. Chow, A priori sparsity patterns for parallel sparse approximate inverse preconditioners, SIAM J. Sci. Comput. 21 (5) (2000) 1804–1822.
[17] E. Chow, A. Patel, Fine-grained parallel incomplete LU factorization, SIAM J. Sci. Comput. 37 (2) (2015) C169–C193.
[18] J. Gao, Z. Li, R. Liang, G. He, Adaptive optimization l1-minimization solvers on GPU, Int. J. Parallel Program. 45 (3) (2017) 508–529.
[19] K. Li, W. Yang, K. Li, A hybrid parallel solving algorithm on GPU for quasitridiagonal system of linear equations, IEEE Trans. Parallel Distrib. Syst. 27 (10) (2016) 2795–2808.
[20] J. Gao, Y. Zhou, G. He, Y. Xia, A multi-GPU parallel optimization model for the preconditioned conjugate gradient algorithm, Parallel Comput. 63 (2017) 1–16.
[21] G. He, J. Gao, J. Wang, Efficient dense matrix–vector multiplication on GPU, Concurr. Comput. Pract. Exp. 30 (19) (2018) e4705, http://dx.doi.org/10.1002/cpe.4705.
[22] J. Gao, K. Wu, Y. Wang, P. Qi, G. He, GPU-accelerated preconditioned GMRES method for two-dimensional Maxwell's equations, Int. J. Comput. Math. 94 (10) (2017) 2122–2144.
[23] M. Lukash, K. Rupp, S. Selberherr, Sparse approximate inverse preconditioners for iterative solvers on GPUs, in: Proceedings of the 2012 Symposium on High Performance Computing, Society for Computer Simulation, San Diego, CA, USA, 2012, pp. 1–8.
[24] K. Rupp, R. Tillet, F. Rudolf, J. Weinbub, A. Morhammer, T. Grasser, A. Jungel, S. Selberher, ViennaCL-linear algebra library for multi-and many-core architectures, SIAM J. Sci. Comput. 38 (5) (2016) S412–S439.
[25] G. He, R. Yin, J. Gao, An efficient sparse approximate inverse preconditioning algorithm on GPU, Concurr. Comput.-Pract. Exp. 32 (7) (2020) e5598.
[26] J. Gao, Q. Chen, G. He, A thread-adaptive sparse approximate inverse preconditioning algorithm on multi-GPUs, Parallel Comput. 101 (2021) 102724, http://dx.doi.org/10.1016/j.parco.2020.102724.
[27] J. Gao, K. Wu, Y. Wang, P. Qi, G. He, GPU-accelerated preconditioned GMRES method for two-dimensional Maxwell's equations, Int. J. Comput. Math. 94 (10) (2017) 2122–2144.
[28] G. He, R. Yin, J. Gao, An efficient sparse approximate inverse preconditioning algorithm on GPU, Concurr. Comput.-Pract. Exp. 32 (7) (2020) e5598, http://dx.doi.org/10.1002/cpe.5598.
[29] J. Gao, Q. Chen, G. He, A thread-adaptive sparse approximate inverse preconditioning algorithm on multi-GPUs, Parallel Comput. 101 (2021) 102724, http://dx.doi.org/10.1016/j.parco.2020.102724.
[30] NVIDIA, Cusparse library, 2019, v10.1, https://docs.nvidia.com/cuda/cusparse/index.html.
[31] Yang. C.T., Huang. C.L., Lin. C.F., Hybrid CUDA, OpenMP, and MPI parallel programming on multicore GPU clusters, Comp. Phys. Commun. 182 (1) (2011) 266–269, http://dx.doi.org/10.1016/j.cpc.2010.06.035.
[32] Cusparse library, v10.1. https://docs.nvidia.com/cuda/cusparse/index.html.
[33] K. Rupp, et al., ViennaCl–linear algebra library for multi-and many-core architectures, SIAM J. Sci. Comput. 38 (5) (2016) S412–S439.

BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100008

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

MVDI25K: A large-scale dataset of microscopic vaginal discharge images

Lin Li [a,b], Jingyi Liu [c], Fei Yu [a], Xunkun Wang [a,*], Tian-Zhu Xiang [d,*]

[a] *QINGDAO HUA JING BIOTECHNOLOGY CO., LTD., No. 77 Keyun Road, Qingdao, China*
[b] *Ocean University of China, Qingdao, China*
[c] *Qingdao University of Science and Technology, Qingdao, China*
[d] *Inception Institute of Artificial Intelligence, United Arab Emirates*

ARTICLE INFO

Keywords:
Vaginal discharge detection
Object segmentation
Object detection
Benchmark dataset
Medical images

ABSTRACT

With the widespread application of artificial intelligence technology in the field of biomedical images, the deep learning-based detection of vaginal discharge, an important but challenging topic in medical image processing, has drawn an increasing amount of research interest. Although the past few decades have witnessed major advances in object detection of natural scenes, such successes have been slow to medical images, not only because of the complex background and diverse cell morphology in the microscope images, but also due to the scarcity of well-annotated datasets of objects in medical images. Until now, in most hospitals in China, the vaginal diseases are often checked by observation of cell morphology using the microscope manually, or observation of the color reaction experiment by inspectors, which are time-consuming, inefficient and easily interfered by subjective factors. To this end, we elaborately construct the first large-scale dataset of **m**icroscopic **v**aginal **d**ischarge **i**mages, named ***MVDI25K***, which consists of 25,708 images covering 10 cell categories related to vaginal discharge detection. All the images in *MVDI25K* dataset are carefully annotated by experts with bounding-box and object-level labels. In addition, we conduct a systematical benchmark experiments on *MVDI25K* dataset with 10 representative state-of-the-art (SOTA) deep models focusing on two key tasks, *i.e.*, object detection and object segmentation. Our research offers the community an opportunity to explore more in this new field.

1. Introduction

Obstetrics and gynecology infectious diseases, such as vaginitis, cervicitis, and endometritis, often trouble women's health. It is reported in [1] that, the incidence of obstetrics and gynecology infectious diseases accounts for about 40% of the female population in China. The vaginal discharge examination is the most direct and effective way to detect the above diseases. For instance, the presence of trichomonas in the secretions can be used to determine whether a patient has trichomonas vaginitis, the presence of clue cells indicates that the patient has bacterial vaginosis, and the presence of candida albicans determine whether the patient has vulvovaginal candidiasis. As mentioned in [2], an increase in the number of leukocytes in vaginal secretions is a strong predictor of bacterial vaginosis or cervical infection. Besides, whether there are epithelial cells in the secretions is also a sign to judge whether the secretion sampling is qualified.

For a long time, manual inspection methods, observing the smear through a high-power microscope to conduct the diagnose, have dominated. As it is known, however, they suffer from some defects, such as time-consuming, labor-intensive, inefficient, and easy to be interfered by subjective factors. Recently, deep learning has prospered in object detection of natural scenes, indicating its great potential in the detection of vaginal discharge. However, there is a long way to go, which can be attributed to two key aspects. Firstly, there are considerable differences in the morphology, number, and distribution of cells in vaginal secretions, due to the differences between not only individuals but also different life stages of the same person. Obviously, it poses numerous difficulties for automatic and robust vaginal discharge detection, such as complex background, scale variations, extremely nonuniform object densities, large aspect ratios, and nonrigid changes of cell shape, as shown in Fig. 1. Most importantly, deep learning greatly relies on the large-scale well-annotated datasets, which has long been lacking in the medical community and thus hinder further research in this field.

To facilitate the study of vaginal discharge detection, we provide two contributions. First, we elaborately constructed a novel large-scale dataset of microscopic vaginal discharge images, called *MVDI25K*, which contains 25,708 microscope images covering 10 object classes of cells related to vaginal discharge detection. To the best of our knowledge, *MVDI25K* is the first large-scale dataset for vaginal discharge detection. It has several distinctive features:

* Corresponding authors.
E-mail addresses: maksimljc@163.com (X. Wang), tianzhu.xiang19@gmail.com (T.-Z. Xiang).

https://doi.org/10.1016/j.tbench.2021.100008
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 11 November 2021

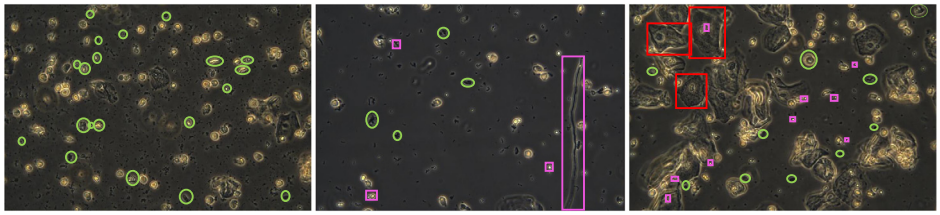


Fig. 1. Various examples of challenging images from *MVDI25K* dataset. The objects labeled with green bounding boxes are all impurities, including cell debris (such as epithelial cell debris), naked nucleus (no cytoplasm and cell membrane), drugs, crystals, starch granules, oil drop, *etc*. In (a), excessive impurities make the complex background, and bring various interferences to object detection. (b) shows the objects with different aspect ratios. Pink bounding boxes label the four types of candida, *i.e.*, candida1, candida2, candida3 and hyphae, among which the longest is hyphae. In (c), the epithelial cells labeled with red bounding boxes show the changeable cell morphology, and the large scale variations compared to the candida cells with pink bounding boxes.

- *Hierarchical categories*. All objects in the microscopic images are labeled into ten cell classes related to vaginal discharge detection, *e.g.*, epithelial cell, clue cell, leukocyte, and lactobacillus, *etc*. Specially, considering the diverse morphology of candida, we annotated them into four sub-classes according to morphological differences. The hierarchical categories could benefit the accurate and fine-grained object detection.
- *Diverse annotations*. The objects in *MVDI25K* dataset are hierarchically annotated with category labels, bounding boxes labels, and object-level masks, which can greatly facilitate different medical image processing tasks, such as object localization, object detection and object/cell segmentation.
- *High quality*. All the images in the dataset are collected by Leica and Olympus phase-contrast microscopes and megapixel dedicated medical Basler cameras with the size from 1536 × 1536 to 2064 × 3088. The phase-contrast microscope facilitates the acquisition of clearer and more realistic cell images. Moreover, cross checking by multiple experts and volunteers is conducted to maintain accuracy, reliability and consistency during the whole annotation process. These high-quality data and annotations could help providing deeper insight into the performance of algorithms.

Second, based on the established *MVDI25K*, we present a comprehensive study on 10 state-of-the-art baselines for vaginal discharge detection. We provide detailed experimental analyses in two scenarios, *i.e.*, object detection and object segmentation. Based on the evaluation results, we find that vaginal discharge detection is very challenging and still far from being solved, leaving much room for improvement. We hope that our research will give a strong boost to growth in this new field.

The remainder of the paper is organized as follows. We review the current medical datasets, medical object detection, and medical object segmentation in Section 2. In Section 3, we present details on the proposed *MVDI25K* dataset, including collection manner, annotation pipeline, and data statistics. Then, we describe benchmark experiments from the aspects of object detection and object segmentation, and provide both quantitative and qualitative experimental analysis in Section 4. Finally, we draw conclusions in Section 5.

2. Related work

In this section, we briefly review some closely related works, including current medical datasets, medical object detection, and medical object segmentation.

2.1. Medical image dataset

In general, X-rays, Computed Tomography (CT), Magnetic Resonance Imaging (MRI), and Positron Emission Computed Tomography (PET) are the four most widely used image-assisted means to help clinicians diagnose diseases, assess prognosis, and plan operations. Thus, a series of corresponding medical image datasets are constructed. Table 1 summarizes their details.

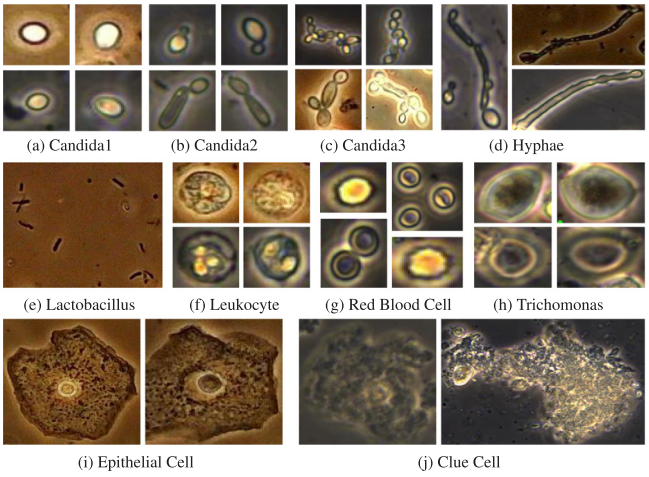


Fig. 2. Examples of 10 classes of cell images in *MVDI25K* dataset.

As is known to all, the well-annotated dataset plays an important role in data-driven medical image processing research. However, to the best of our knowledge, there is few datasets collected from microscope imaging for vaginal discharge research, which may hinder further research in this field. To this end, in this paper, we constructed the first large-scale dataset of vaginal discharge images providing professional annotations. Compared with Peng's dataset [11], the proposed *MVDI25K* provides more images with diverse and rich annotations. It is worth noting that collecting the microscopic image dataset is more difficult than datasets of other medical imaging equipment, because the image quality of microscope imaging is greatly affected by various factors, e.g., focus adjustment.

2.2. Deep models for medical object detection

Object detection, to identify and locate objects in an image or video, is a longstanding problem in computer vision. Recently, with the development of deep learning, many researchers in the medical image processing community have adapted deep object detectors developed for natural images to medical images.

As suggested in [12,13], object detection can be roughly divided into two categories: two-stage algorithms, such as R-CNN [14] and its variants, and one-stage algorithms, such as YOLO [15] and SSD [16]. Due to its high efficiency and good performance, YOLO and its variants have attracted extensive attention in the medical imaging community.

Table 1
Medical image datasets.

Dataset	Object	Year	Number	X-ray	CT	MRI	PET	MImg	BBox.	Obj.	Cate.
ABIDE [3]	Brain	2013	1,112*			✓				✓	✓
OASIS-3 [4]	Brain	2019	3,776			✓	✓			✓	✓
DDSM [5]	Breast	2000	10,239	✓						✓	✓
MURA [6]	Upper Limb	2018	40,561	✓							✓
LIDC-IDRI [7]	Lung	2006	244,527	✓	✓					✓	✓
LUNA16[a]	Lung	2016	888		✓					✓	✓
NSCLC [8]	Lung	2018	1,355		✓		✓			✓	✓
DeepLesion [9]	Lung *etc.*	2018	928,020		✓	✓	✓		✓		✓
ChestX-ray14 [10]	Chest	2017	112,120	✓					✓		✓
Peng' Dataset [11]	Vaginal Dis.	2021	3,645					✓			✓
MVDI25K (Ours)	Vaginal Dis.	2021	25,708					✓	✓	✓	✓

BBox.: Bounding-box annotation. Obj.: Object-level annotation. Cate.: Categories. MImg: Microscopic image. Vaginal Dis.: Vaginal discharge. *: number of sub-datasets.
[a]https://luna16.grand-challenge.org/.

Based on the chest CT scans from Lung Image Database Consortium, YOLO-based model was applied to efficiently and accurately identify lung nodules in [17]. The high-quality gallstone CT image dataset was established in [18], and YOLO-v3 achieved good performance on the detection of granular gallstones and muddy gallstones. To effectively fighting against COVID-19, an improved model based on YOLOv2 and ResNet-50 was designed to detect medical masks with high accuracy [19].

2.3. Deep models for medical object segmentation

In the past few years, convolutional neural networks have been the most commonly-used architecture in state-of-the-art models for medical image segmentation, such as FCN [20], U-Net [21], and Deeplab series [22].

Among them, the U-net plays an important role and has been applied to numerous fields, such as using NAS-Unet to segment Magnetic Resonance Imaging (MRI), Computed Tomography (CT), and ultrasound with high quality [23], and medical object segmentation including liver, brain and lung tissue and tumor segmentation [24–26], cell segmentation [27], optic disc segmentation [28], *etc*.

Another related topic that deserves attention is camouflage object detection [29,30], which is to segment objects which have a similar pattern (e.g. texture, color and direction) to their natural or man-made environment. In our vaginal discharge detection, trichomonas has a strong camouflage compared to other cell morphologies. Thus camouflage object detection could shed new light on trichomonas detection.

3. Proposed dataset

Our *MVDI25K* dataset contains 25,708 microscopic images belonging to ten object/cell classes related to vaginal discharge detection. The images are carefully selected to cover diverse challenging cases, *e.g.*, complex background, large-scale variation, and nonuniform object density. Examples can be found in Figs. 1 and 2. We will describe the details of *MVDI25K* in terms of three aspects, *i.e.*, data collection, annotation pipeline, and data statistics, as follows.

3.1. Data collection

We build a high-quality dataset, *MVDI25K*, images of which are collected from the *HJ500* Discharge Analysis Workstation with two sources. One is directly photographed from the fresh samples collected by many hospitals across the country, and the other is captured by ourselves using the specimens we collected from other partner hospitals. The images are acquired by Leica and Olympus phase-contrast microscope and the megapixel dedicated medical Basler camera. Our dataset covers *315* hospitals distributed in more than *20* provinces in China, and 26 of them are tertiary hospitals including Beijing Tiantan Hospital and Hubei Maternity and Child Health Hospital. The entire collection work last nearly 9 weeks. We just use the devices to collect the image data independently, and do not collect any patient information. The images are free from copyright and loyalties and will be available at: https://zenodo.org/record/5523661.

Besides, the images are acquired by two types of microscopes, Leica and Olympus microscope. Both microscopes adopt a phase contrast field of view, which is more suitable for microscopy examination than ordinary optical microscopes, and even unstained cells can be observed more clearly and brighter.

3.2. Professional annotation

To facilitate the study of vaginal discharge detection, we provide bounding-box and object-level annotations for each image in our *MVDI25K* dataset. We hire seven professional annotators, and six of them are divided into three groups. Each group is responsible for the annotation and meantime they need to cross-check the label results from other groups. After finishing the annotation process, the team leader (the seventh annotation expert) will carefully conduct the final validation to ensure high-quality annotation.

3.2.1. Categories

We establish a hierarchical taxonomic system for the proposed dataset. We first choose seven major cell categories such as *epithelial cell*, *clue cell*, *leukocyte*, *candida*, *red blood cell*, *lactobacillus*, and *trichomonas*. Considering the large morphology differences of candida cells, shown in the first row of Fig. 2, we divided the candida into four sub-classes, namely *candida1*, *candida2*, *candida3*, and *hyphae*. Finally, we integrate these classes into 10 cell/ object classes. The taxonomic structure of our *MVDI25K* is given in Fig. 3(a). The example of the word cloud is shown in Fig. 3(b). We believe that the fine-grained classification of candidas would play a positive role in accurate vaginal discharge detection.

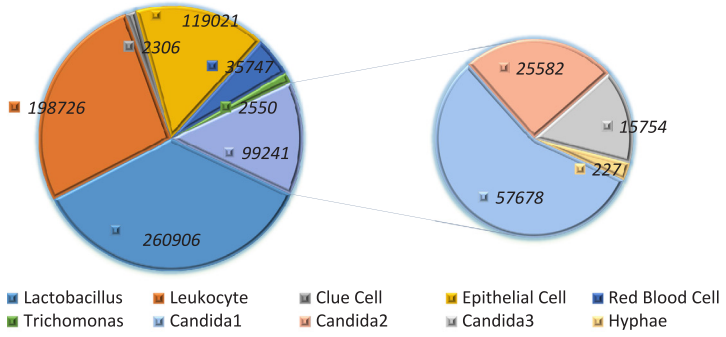
3.2.2. Bounding-box annotation

Bounding box is widely used in object detection and localization. To extend *MVDI25K* for the object proposal task, we carefully annotate the bounding boxes around the objects in each image. Finally, we obtained total 718,497 object instances from 25,708 microscopic images. Some examples of annotated patches are shown in Fig. 4(a).

3.2.3. Object-level annotation

High-quality pixel-level annotations are necessary for *MVDI25K* dataset. Here we focus on the *Trichomonas* category, the disease who causes is one of the most common obstetrics and gynecology infectious diseases.[1] Besides, the Trichomonas cell is usually active and thus difficult to be observed clearly from vaginal secretion microscope sample,

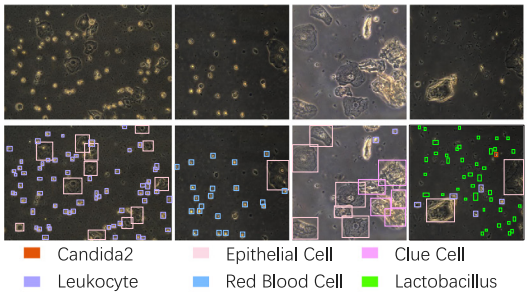
[1] https://news.un.org/en/story/2019/06/1039891.



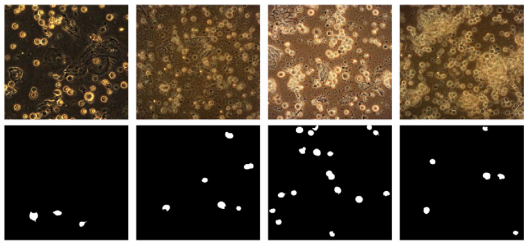
(a) The 10 cell/object classes in *MVDI25K* and the number of labels in each class.

(b) Word cloud of *MVDI25K*.

Fig. 3. Categories of *MVDI25K*.



(a) Examples of bounding-box annotations in the proposed *MVDI25K* dataset. As can be seen, the labeled objects are with diverse scales, nonuniform distributions, and nonrigid shape changes, which bring challenges for object detection.



(b) Examples of object-level annotations for Trichomonas in the proposed *MVDI25K* dataset. As can be seen, the labeled Trichomonas are often small, and with some occlusion or overlap. In addition, they are generally easily concealed in their background, that is, very similar to their background, which may largely confuse the algorithms.

Fig. 4. Annotations of *MVDI25K*.

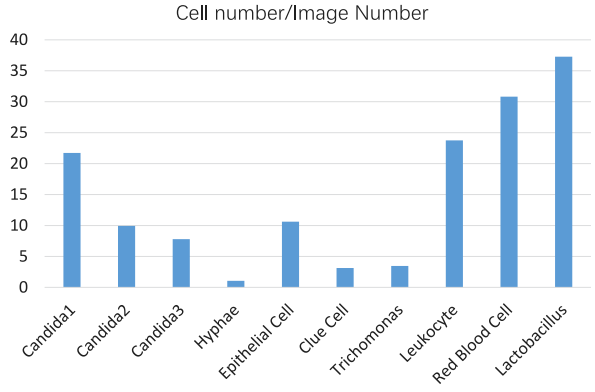


Fig. 5. Multiple Objects. The number of each object in each image.

which poses a challenge for data collection. Through observation, we found that most of the Trichomonas cells seem to have a similar pattern, *e.g.*, texture, color and shape, to other cells in the background, that is, they have a certain degree of concealment, which is easy to cause confusion to the detector. What is more, by data annotation, it can be seen that the Trichomonas cell generally has the following challenging attributes: (1) dense objects: more than 10 objects in a single image; (2) small object: too small size compared with its large background; (3) occlusion/overlap: incomplete object contour due to the occlusion of other cells or impurities and multiple cells overlap; (4) irregular shape: cell contains tiny parts (*e.g.* small tails). As a result, it may become a hot potato when using deep models to detect this type of cells.

To this end, Trichomonas detection deserves more effort and thus we provide meticulous object-level annotations for Trichomonas. We adopt *Photoshop* as the annotation tool to label the object-level masks. In this way, we obtain a total of 2,550 object-level annotations from 912 Trichomonas images. Some examples can be seen in Fig. 4(b).

3.3. Dataset features and statistics

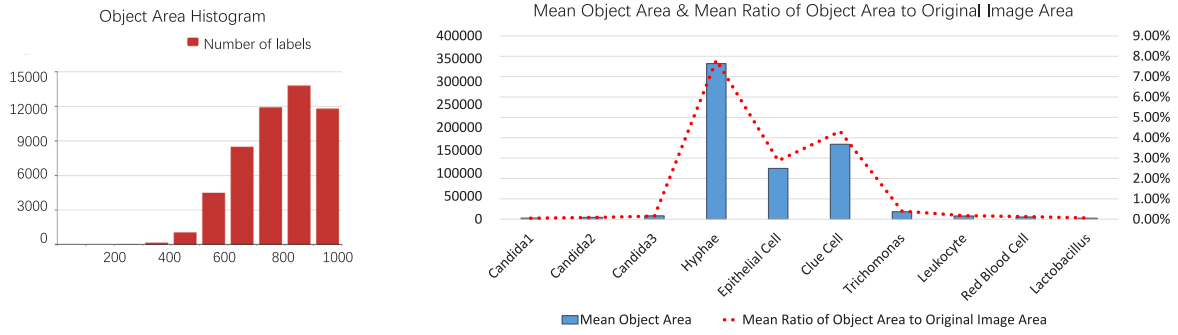
To provide deeper insights into our *MVDI25K*, we present its several important characteristics in below.

3.3.1. Multiple objects

In this paper, we define multiple objects as cells of the same type in one image with a number equal to or greater than two. Note that the multi-object value is the total number of a certain type of object divided by the number of images containing this object. As shown in Fig. 5, hypha is with the low multi-object attribute value 1.07, while the other cell classes are larger than 3. The top-3 is the Lactobacillus, red blood cell, and Leukocyte, which are 37.27, 30.82, and 23.75, respectively.

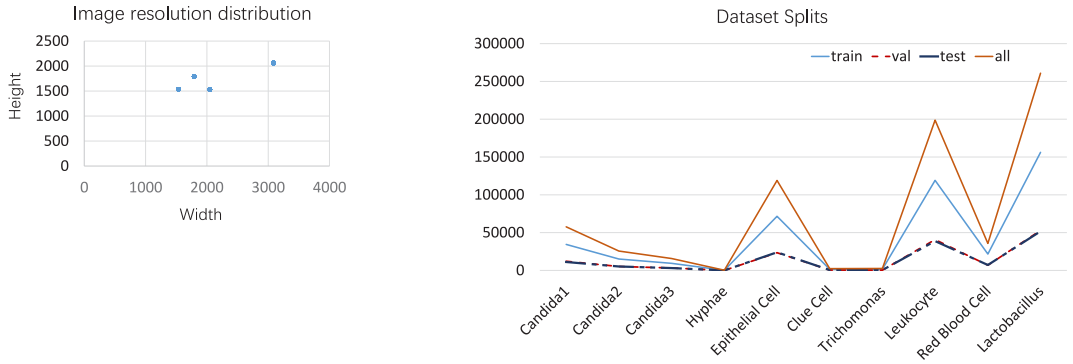
3.3.2. Small object

Small objects, As we know, is defined as (a) the objects whose absolute size is less than 32×32, or (b) the objects whose width and height are less than 1/10 of the width and height of the whole image. Generally speaking, the small object is easily overwhelmed by the noisy background. In addition, for deep models, its feature information will disappear when the network gradually goes deeper, which lets many deep models be cast into the shade. Thus, detection of small objects is a challenging issue.



(a) Number of pixels per object/cell. (b) Object pixel ratio.

Fig. 6. The attribution analysis of small object in the proposed *MVDI25K*.



(a) Image resolution distribution of *MVDI25K* dataset (unit for the axis: pixel). (b) Dataset Splits

Fig. 7. Annotations of *MVDI25K*.

We counted these two small target attributes in the *MVDI-25K* dataset, shown in Fig. 6(a). It shows the number of cells whose absolute pixel is less than or equal to 1000. A total of 51,549 small objects are labeled in the proposed dataset, accounting for 7.17%. Fig. 6(b) shows the mean area of each type of cells and the mean ratio of their area in one images. Obviously, except for hyphae, clue cell and epithelial cell, the area ratio of other classes of cells are far less than 1%, the threshold of small objects.

3.3.3. Resolution distribution

High-resolution images generally provide more object details for deep model training and thus facilitate to yield outstanding detection performance when testing [31]. When collecting data, we carefully adjusted the microscope settings to obtain high-resolution images. Fig. 7(a) shows the resolution distribution of our dataset. Specifically, the four resolutions of the images are: 1536×1536, 1536×2048, 1792×1792, 2064×3088, and their proportions are: 30.03%, 0.47%, 42.94%, 26.56%.

3.3.4. Dataset splits

To provide a large quantity of training data for learning-based approaches, we divided 25,708 images into training set, validation set and test set, with a ratio of 6:2:2. It should be noted that it is impossible to split the dataset based on cell class and then select randomly from each class, because each image contains at least 2 or 3 classes of cells. In order to ensure the same distribution of the training set, the validation set, and the test set, we first select the images with the least number of cells (hyphae), and then select them randomly. Then the images containing the second-fewest cell classes (Trichomonas) are selected at random. Follow this rule until all types of cells have been split. Fig. 7(b) presents the final split results of different cell categories. Consequently, the dataset split satisfies the same distribution of training set, validation set and test set.

4. Benchmark experiments

Based on the established *MVDI25K*, we systematically benchmark 10 representative models on two key tasks, object detection and object segmentation. From the evaluation results, we conduct some in-depth analysis and present several insightful conclusions which may inspire further research.

4.1. Object detection experiments

4.1.1. Dataset settings

Based on the data split rules described in Section 3.3.4, we split the whole images into 15,428 training set, 5,143 validation set, and 5,137 test set, respectively, with corresponding bounding box ground-truth.

4.1.2. Training protocols

In this benchmark experiment, we collect the released codes of three representative object detection models, that is YOLOv5-s, YOLOv5-m, and YOLOv5-x [32], and re-train these models with the training set of *MVDI25K*. The images are set to 640×640 as model input. The initial learning rate is 1e-2, and GIoU loss gain is 0.05, and class loss gain is 0.5. For optimizer, the momentum is set to 0.937, and weight decay is set to 0.0005. The batch size of YOLOv5-s, YOLOv5-m and YOLOv5-x are set to 36, 24 and 8 respectively. The total training is 300 epoch on a NVIDIA GeForce RTX 2080Ti with 11 GB memory.

4.1.3. Evaluation metrics

We apply three widely-used metrics to evaluate object detection performance. These metrics include precision (P), recall (R), and mean average precision (mAP).

P is the accuracy rate, that is the percentage of the correct positive classes account for all positive classes detected, *i.e.*, $P=TP/(TP+FP)$.

Table 2
Quantitative results of object detection on our *MVDI25K* dataset. Cls1-10: Candida1, Candida2, Candida3, Hyphae, Epithelial Cell, Clue Cell, Trichomonas, Leukocyte, Red Blood Cell, and Lactobacillus. The first three categories with the worst performance are shown in red, blue, and green fonts.

Models	Metric	All	*Cls*1	*Cls*2	*Cls*3	*Cls*4	*Cls*5	*Cls*6	*Cls*7	*Cls*8	*Cls*9	*Cls*10
YOLOv5-s [32]	P	0.464	0.480	0.407	0.409	0.198	0.690	0.267	0.473	0.596	0.643	0.480
	R	0.894	0.966	0.942	0.950	0.476	0.982	0.811	0.915	0.985	0.961	0.956
	mAP@.5	0.801	0.882	0.853	0.860	0.307	0.964	0.572	0.825	0.942	0.936	0.872
	mAP@.5:.95	0.560	0.520	0.540	0.557	0.185	0.809	0.417	0.599	0.715	0.754	0.502
YOLOv5-m [32]	P	**0.502**	0.507	0.449	0.458	0.202	0.747	0.287	0.536	0.648	0.678	0.513
	R	0.905	0.971	0.959	0.967	0.548	0.976	0.831	0.908	0.980	0.957	0.957
	mAP@.5	0.820	0.891	0.897	0.899	0.370	0.965	0.573	0.840	0.945	0.939	0.879
	mAP@.5:.95	0.588	0.541	0.590	0.613	0.243	0.816	0.430	0.618	0.732	0.773	0.521
YOLOv5-x [32]	P	0.493	0.512	0.476	0.495	0.183	0.717	0.263	0.505	0.629	.657	0.487
	R	**0.919**	0.977	0.971	0.972	0.619	0.979	0.843	0.915	0.983	0.962	0.967
	mAP@.5	**0.837**	0.901	0.920	0.919	0.434	0.966	0.605	0.851	0.947	0.941	0.883
	mAP@.5:.95	**0.607**	0.561	0.617	0.640	0.266	0.819	0.465	0.650	0.741	0.782	0.532

Noted *TP* means that both the predicted value and the true value are both 1, and *FP* denotes that the true value is 0 and the predicted value is 1.

R is the recall rate, that is the percentage of the correct positive class accounts for all the true positive classes, *i.e.*, *R=TP/(TP+FN)*. Noted *FN* means that the true value is 1 and the predicted value is 0.

AP is the area under the *P-R* curve, and *mAP* is the average of *AP* of each category. We adopt *mAP@.5* where ".5" indicates the threshold for judging *IoU* as a positive or negative sample, and *mAP@0.5:0.95* which means the *AP* average under a series of thresholds that start at 0.5 and increase to 0.95 in steps of 0.05.

4.1.4. Quantitative evaluation

As can be seen in Table 2, from the "all" column, the overall missed detection rate of the outstanding YOLOv5-x model is only 8.1%(*i.e.*, 1-91.9%), but its cost exceeds that of the YOLOv5-m model. False positive samples accounted for 50.7% (*i.e.*, 1-49.3%), which means that the number of false positive cells detected is almost the same as the number of true cells. In fact, the requirements for accurate cell identification in medical images are relatively high. Even if *R* and mAP have reached a high level, the low *P* metrics means too many false positive samples, which is a very important but still challenging problem in medical image detection.

As shown in Table 2, in all evaluation items, the fourth (Hyphae) and sixth classes (Clue Cell) are the two worst performing categories. Objectively speaking, in our *MVDI25K* dataset, the numbers of images containing hyphae and clue cells are very small, especially only 213 images contain hyphae. For the seventh classes cell (Trichomonas), due to the small number of images, and sometimes the similar appearance to leukocyte, it is also a difficult class in object detection. As we all know, different types of objects are inherently unevenly distributed in nature. It can also be seen from this experiment that the object detection of unbalanced categories is a very important but challenging issue, which is worthy of further study.

4.1.5. Qualitative evaluation

Five representative detection results are shown in Fig. 8. In the first line, the epithelial cell in the middle left is disturbed by the complex background, causing the failed detection for three models. In addition, three models misjudged its left cell as Trichomonas. The second row contains two types of typical red blood cells, dark side and bright side, as well as intact and broken white blood cells. At the same time, leukocytes in complete and fragmented form. YOLOv5-m model is the best for detecting multiple leukocytes in the upper left corner of the image. The image in the third row is a typical environment where a large number of lactobacillus exist. From the perspective of confidence, YOLOv5-m performed slightly better than YOLOv5-x. The difficulty in the identification of the fourth line of pictures is that the morphology of Trichomonas and some leukocytes are very similar, especially the three adjacent cells in the upper right corner. None of the three models can completely accurately identify leukocytes and Trichomonas. The last image contains intact epithelium and deformed epithelium. It is difficult to identify deformed epithelium, and all three models are missed.

4.2. Object segmentation experiments

4.2.1. Dataset settings

In this benchmark experiment, our dataset provides a total of 2550 object-level annotations from 912 Trichomonas images. We split these images into 730 images for training and 182 for testing, with corresponding object-level ground-truth.

4.2.2. Training protocols

In this study, we evaluate eight representative, recently published and state-of-the-art deep models for object segmentation or concealed object detection, including BASNet [33], CPD [34], SCRN [35], U^{2}Net [36], F3Net [37], GateNet [38], PraNet [39], and SINet [29]. We collect the released codes of these models and re-train them on our dataset with 50 epochs on a NVIDIA GeForce RTX 2080Ti GPU. During the training stage, the batch size is set to 20, and the maximum learning rate is 0.05. For the Adam optimizer, the momentum is 0.9 and the weight decay is 5e-4. When the memory is insufficient, the batch size and epoch are changed to 10 and 100, respectively.

4.2.3. Evaluation metrics

To provide a comprehensive evaluation, six widely-used metrics are employed to quantitatively compare the eight deep models for object segmentation on the proposed *MVDI25K*, with the evaluation toolbox provided by [39], including structural similarity measure (S_α, with α =0.5) [40], enhanced-alignment measure ($meanE_\phi$ and $maxE_\phi$), and F_β measure (wF_β, $meanF_\beta$ and $maxF_\beta$).

- S-Measure calculates the structure similarities from the object-aware and region-aware aspects, between objects in ground-truth (GT) maps and predicted maps:

$$S_\alpha = \alpha * S_o + (1-\alpha) * S_r, \tag{1}$$

where

$$S_o = \frac{2 * E(pre)}{E(pre)^2 + 1 + \sigma + e}, \tag{2}$$

S_r indicates that the four regions are cut into four regions according to the position of the center of gravity, and the area of the entire image occupied by the pixels of the four regions is used as the weight, and the weighted average of the structure similarities of the four regions is calculated.

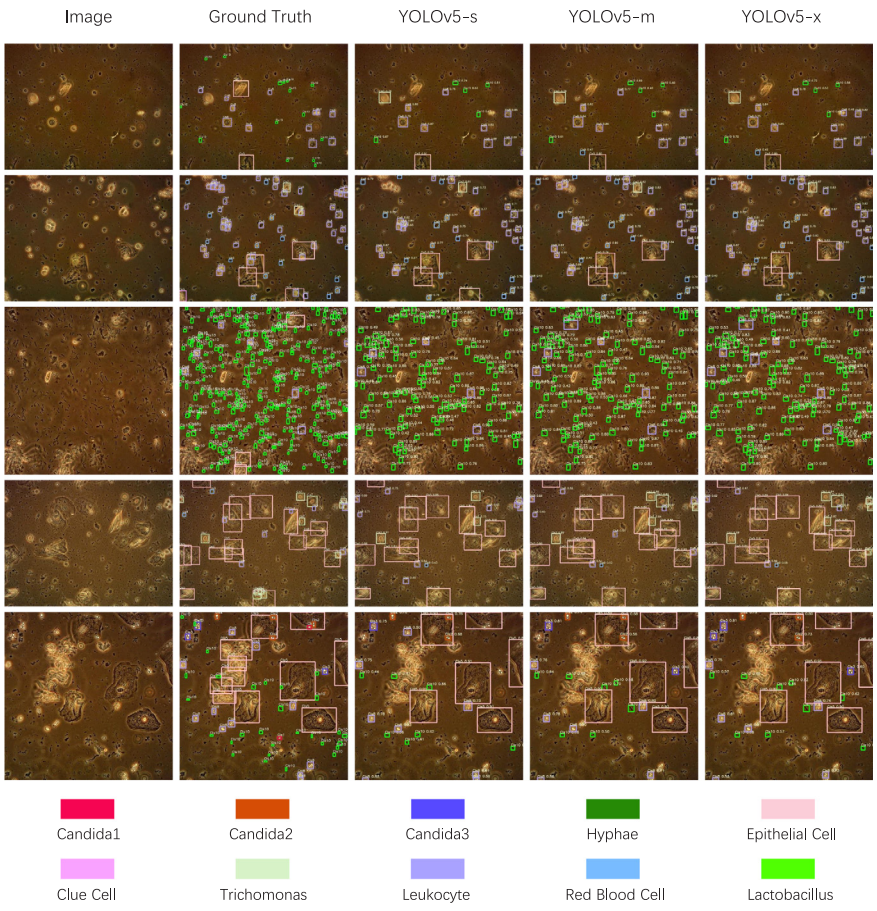


Fig. 8. Qualitative examples of object detection with three representative YOLO models evaluated on our *MDVI25K* dataset.

- E-Measure is a cognitive vision-inspired metric, which measures both the local and global similarities between two binary maps [41]. It combines local pixel values and image-level average values to capture both image-level statistics and local pixel matching information. Specifically, it is defined as:

$$E_\phi = \frac{1}{w * h} \sum_{x=1}^{w} \sum_{y=1}^{h} \phi FM(x, y) \tag{3}$$

where w and h are the width and the height of the map, respectively. $\phi FM = f(\xi FM)$, the value of ξFM depends on the similarity between feature map and ground truth, $f(x) = \frac{1}{4}(1 + x)^2$. The rest of the specific derivation procedure is given by [41]. Here, we introduce mean/maximal E-measure, *i.e.*, $meanE_\phi$ and $maxE_\phi$, to provide a more comprehensive evaluation.

- F-measure is essentially a region-based similarity metric, which is based on weighted precision and recall values.

$$F_\beta = \frac{(1 + \beta^2)R * P}{R + \beta^2 P}, \tag{4}$$

where β^2 is a parameter to trade-off recall and precision, and it is usually set to 0.3. Here, Here, we introduce 3 variants of this metric, namely wF_β, $meanF_\beta$ and $maxF_\beta$, for a comprehensive evaluation.

4.2.4. Quantitative evaluation

Table 3 shows the evaluation results of all models on our dataset. Overall, PraNet is the best performing models compared to the others. PraNet obtained the best results on four metrics, S_α, wF_β, $meanF_\beta$ and $maxF_\beta^w$, especially on S_α. Its value for the S_α metric is 0.159 above the mean (0.648) and 0.014 above the second place (0.793). BASNet and U^{2}Net performed the best on metrics $meanE_\phi$ and $maxE_\phi$, respectively. The $meanE_\phi$ of BASNet was 0.158 higher than the average (0.711) and the $maxE_\phi$ of U^{2}Net was 0.043 higher compared to the average (0.852).

Although some deep models have achieved seemingly-good results on some metrics, they are still far from obtaining satisfactory performance. In addition, trichomonas is extremely active in microscope samples, so when observed with a microscope, trichomonas usually has two states of blur and clarity. Our current dataset is mainly focused on labeling clear trichomonas. We will continue to label the active and non-clearly trichomonas to construct an increasingly challenging dataset.

4.2.5. Qualitative evaluation

Fig. 9 shows seven representative results, including original images, ground truth, and object segmentation results of deep models. Specifically, our images contain multiple dense objects (first row), transgressions (second row), overlaps (third row), partial occlusion (4th and 5th rows), complex shapes (5th row), small objects and cases with extremely high similarity between objects and backgrounds, which bring many difficulties for object segmentation. From the segmentation map, it can be seen that the these models are in a less favorable situation for trichomonas edge recognition.

Obviously, the segmentation results in the first row are angular and ambiguous. For trichomonas that are partially occluded by other objects (more than 30% of their own area), there is a tendency to miss detection. For example, the trichomonas near the middle-left position in the fourth row, all deep models failed to detect it. Meanwhile, most of the above models cannot accurately capture the complex shape of the object. Trichomonas in the upper left corner of the fourth row and the middle right of the fifth row cannot be clearly and completely identified and all models miss their small tails or flagella. In addition, for objects

Table 3
Quantitative results of object segmentation on our *MVDI25K* dataset. "↑" indicates the higher the score the better. The best results are in boldface.

Models	$S_\alpha \uparrow$	$meanE_\phi \uparrow$	$maxE_\phi \uparrow$	$wF_\beta \uparrow$	$meanF_\beta \uparrow$	$maxF_\beta \uparrow$
2019 BASNet [33]	0.793	**0.868**	0.890	0.585	0.634	0.653
2019 CPD [34]	0.616	0.608	0.716	0.253	0.328	0.400
2019 SCRN [35]	0.641	0.771	0.893	0.217	0.517	0.587
2020 U^2Net [36]	0.691	0.739	**0.895**	0.344	0.489	0.612
2020 F3Net [37]	0.791	0.856	0.881	0.557	0.623	0.661
2020 PraNet [39]	**0.807**	0.856	0.880	**0.613**	**0.658**	**0.677**
2020 SINet [29]	0.520	0.726	0.868	0.044	0.472	0.578

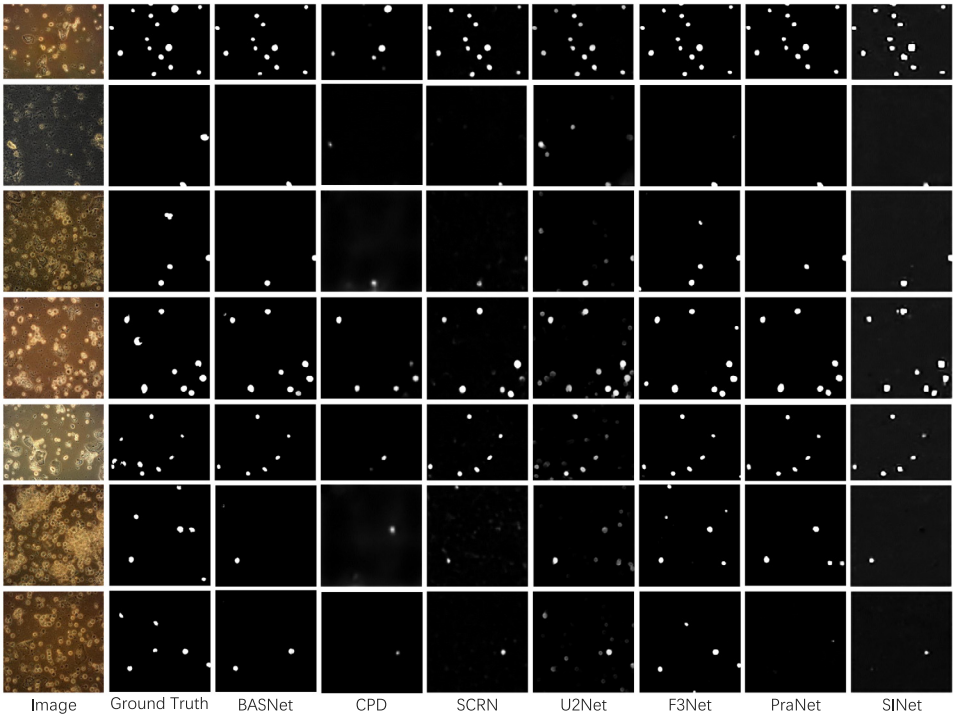


Fig. 9. Qualitative examples of object segmentation with the existing representative models evaluated on our *MDVI25K* dataset.

with similar colors and shapes (such as leukocytes and trichomonas), most of models tend to confuse them.

5. Conclusion

In this paper, we have presented the first comprehensive benchmark study on vaginal discharge detection. Specifically, we have constructed the first large-scale and challenging dataset of microscopic vaginal discharge, *MVDI25K*, containing 25,708 images with diverse and high-quality annotations. Then, we conducted a systematical benchmark experiments on 10 representative SOTA deep models on two key tasks, *i.e.*, object detection and object segmentation, and provided some insightful discussions. The benchmark indicates that vaginal discharge detection is far from being solved. We hope the studies presented in this work would facilitate the development of this field.

References

[1] R. Chen, Q. Liao, Attach importance to female reproductive tract infection and vaginal microecological diagnosis and treatment in China, Chin. J. Lab. Med. 41 (4) (2018) 251–253.
[2] W. Geisler, S. Yu, M. Venglarik, J. Schwebke, Vaginal leucocyte counts in women with bacterial vaginosis: relation to vaginal and cervical infections, Sex. Transm. Infect. 80 (5) (2004) 401–405.
[3] C. Craddock, Y. Benhajali, C. Chu, F. Chouinard, A. Evans, A. Jakab, B.S. Khundrakpam, J.D. Lewis, Q. Li, M. Milham, et al., The neuro bureau preprocessing initiative: open sharing of preprocessed neuroimaging data and derivatives, Front. Neuroinformatics 7 (2013).
[4] P.J. LaMontagne, T.L. Benzinger, J.C. Morris, S. Keefe, R. Hornbeck, C. Xiong, E. Grant, J. Hassenstab, K. Moulder, A. Vlassenko, et al., OASIS-3: longitudinal neuroimaging, clinical, and cognitive dataset for normal aging and alzheimer disease, MedRxiv (2019).
[5] K. Bowyer, D. Kopans, W. Kegelmeyer, R. Moore, M. Sallam, K. Chang, K. Woods, The digital database for screening mammography, in: Third International Workshop on Digital Mammography, Vol. 58, pp. 27.
[6] P. Rajpurkar, J. Irvin, A. Bagul, D. Ding, T. Duan, H. Mehta, B. Yang, K. Zhu, D. Laird, R.L. Ball, et al., Mura: Large dataset for abnormality detection in musculoskeletal radiographs, 2017, arXiv preprint arXiv:1712.06957.
[7] S.G. Armato III, G. McLennan, L. Bidaut, M.F. McNitt-Gray, C.R. Meyer, A.P. Reeves, B. Zhao, D.R. Aberle, C.I. Henschke, E.A. Hoffman, et al., The lung image database consortium (LIDC) and image database resource initiative (IDRI): A completed reference database of lung nodules on CT scans, Med. Phys. 38 (2) (2011) 915–931.
[8] S. Bakr, O. Gevaert, S. Echegaray, K. Ayers, M. Zhou, M. Shafiq, H. Zheng, J.A. Benson, W. Zhang, A.N. Leung, et al., A radiogenomic dataset of non-small cell lung cancer, Sci. Data 5 (1) (2018) 1–9.
[9] K. Yan, X. Wang, L. Lu, R.M. Summers, DeepLesion: automated mining of large-scale lesion annotations and universal lesion detection with deep learning, J. Med. Imaging 5 (3) (2018) 036501.
[10] X. Wang, Y. Peng, L. Lu, Z. Lu, M. Bagheri, R.M. Summers, Chestx-ray8: Hospital-scale chest x-ray database and benchmarks on weakly-supervised classification and localization of common thorax diseases, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2017, pp. 2097–2106.
[11] S. Peng, H. Huang, M. Cheng, Y. Yang, F. Li, Efficiently recognition of vaginal micro-ecological environment based on convolutional neural network, in: 2020 IEEE International Conference on E-Health Networking, Application & Services, HEALTHCOM, IEEE, 2021, pp. 1–6.
[12] L. Liu, W. Ouyang, X. Wang, P. Fieguth, J. Chen, X. Liu, M. Pietikäinen, Deep learning for generic object detection: A survey, Int. J. Comput. Vis. 128 (2) (2020) 261–318.

[13] Z. Zou, Z. Shi, Y. Guo, J. Ye, Object detection in 20 years: A survey, 2019, arXiv preprint arXiv:1905.05055.
[14] R. Girshick, J. Donahue, T. Darrell, J. Malik, Rich feature hierarchies for accurate object detection and semantic segmentation, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 580–587.
[15] J. Redmon, S. Divvala, R. Girshick, A. Farhadi, You only look once: Unified, real-time object detection, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 779–788.
[16] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, A.C. Berg, Ssd: Single shot multibox detector, in: European Conference on Computer Vision, Springer, 2016, pp. 21–37.
[17] J. George, S. Skaria, V. Varun, et al., Using YOLO based deep learning network for real time detection and localization of lung nodules from low dose CT scans, in: Medical Imaging 2018: Computer-Aided Diagnosis, Vol. 10575, International Society for Optics and Photonics, 2018, p. 105751I.
[18] S. Pang, T. Ding, S. Qiao, F. Meng, S. Wang, P. Li, X. Wang, A novel YOLOv3-arch model for identifying cholelithiasis and classifying gallstones on CT images, PLoS One 14 (6) (2019) e0217647.
[19] M. Loey, G. Manogaran, M.H.N. Taha, N.E.M. Khalifa, Fighting against COVID-19: A novel deep learning model based on YOLO-v2 with ResNet-50 for medical face mask detection, Sustainable Cities Soc. 65 (2021) 102600.
[20] J. Long, E. Shelhamer, T. Darrell, Fully convolutional networks for semantic segmentation, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 3431–3440.
[21] O. Ronneberger, P. Fischer, T. Brox, U-net: Convolutional networks for biomedical image segmentation, in: International Conference on Medical Image Computing and Computer-Assisted Intervention, Springer, 2015, pp. 234–241.
[22] L.-C. Chen, G. Papandreou, F. Schroff, H. Adam, Rethinking atrous convolution for semantic image segmentation, 2017, arXiv preprint arXiv:1706.05587.
[23] Y. Weng, T. Zhou, Y. Li, X. Qiu, Nas-unet: Neural architecture search for medical image segmentation, IEEE Access 7 (2019) 44247–44257.
[24] V. Cherukuri, P. Ssenyonga, B.C. Warf, A.V. Kulkarni, V. Monga, S.J. Schiff, Learning based segmentation of CT brain images: application to postoperative hydrocephalic scans, IEEE Trans. Biomed. Eng. 65 (8) (2017) 1871–1884.
[25] W. Li, et al., Automatic segmentation of liver tumor in CT images with deep convolutional neural networks, J. Comput. Commun. 3 (11) (2015) 146.
[26] Y. Onishi, A. Teramoto, M. Tsujimoto, T. Tsukamoto, K. Saito, H. Toyama, K. Imaizumi, H. Fujita, Multiplanar analysis for pulmonary nodule classification in CT images using deep convolutional neural network and generative adversarial networks, Int. J. Comput. Assist. Radiol. Surg. 15 (1) (2020) 173–178.
[27] T.-H. Song, V. Sanchez, H. EIDaly, N.M. Rajpoot, Dual-channel active contour model for megakaryocytic cell segmentation in bone marrow trephine histology images, IEEE Trans. Biomed. Eng. 64 (12) (2017) 2913–2923.
[28] H. Fu, J. Cheng, Y. Xu, D.W.K. Wong, J. Liu, X. Cao, Joint optic disc and cup segmentation based on multi-label deep network and polar transformation, IEEE Trans. Med. Imaging 37 (7) (2018) 1597–1605.
[29] D.-P. Fan, G.-P. Ji, G. Sun, M.-M. Cheng, J. Shen, L. Shao, Camouflaged object detection, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 2777–2787.
[30] L. Li, B. Dong, E. Rigall, T. Zhou, J. Donga, G. Chen, Marine animal segmentation, IEEE Trans. Circuits Syst. Video Technol. (2021).
[31] Y. Zeng, P. Zhang, J. Zhang, Z. Lin, H. Lu, Towards high-resolution salient object detection, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2019, pp. 7234–7243.
[32] G. Jocher, K. Nishimura, T. Mineeva, R. Vilariño, Yolov5, Code Repos. (2020) https://github.com/ultralytics/yolov5.
[33] X. Qin, Z. Zhang, C. Huang, C. Gao, M. Dehghan, M. Jagersand, Basnet: Boundary-aware salient object detection, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 7479–7489.
[34] Z. Wu, L. Su, Q. Huang, Cascaded partial decoder for fast and accurate salient object detection, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 3907–3916.
[35] Z. Wu, L. Su, Q. Huang, Stacked cross refinement network for edge-aware salient object detection, in: The IEEE International Conference on Computer Vision (ICCV), 2019.
[36] X. Qin, Z. Zhang, C. Huang, M. Dehghan, O.R. Zaiane, M. Jagersand, U2-net: Going deeper with nested U-structure for salient object detection, Pattern Recognit. 106 (2020) 107404.
[37] J. Wei, S. Wang, Q. Huang, F^3Net: Fusion, feedback and focus for salient object detection, in: Proceedings of the AAAI Conference on Artificial Intelligence, Vol. 34, 07, 2020, pp. 12321–12328.
[38] X. Zhao, Y. Pang, L. Zhang, H. Lu, L. Zhang, Suppress and balance: A simple gated network for salient object detection, in: European Conference on Computer Vision, Springer, 2020, pp. 35–51.
[39] D.-P. Fan, G.-P. Ji, T. Zhou, G. Chen, H. Fu, J. Shen, L. Shao, PraNet: Parallel reverse attention network for polyp segmentation, in: International Conference on Medical Image Computing and Computer-Assisted Intervention, Springer, 2020, pp. 263–273.
[40] D.-P. Fan, M.-M. Cheng, Y. Liu, T. Li, A. Borji, Structure-measure: A new way to evaluate foreground maps, in: Proceedings of the IEEE International Conference on Computer Vision, 2017, pp. 4548–4557.
[41] D.-P. Fan, C. Gong, Y. Cao, B. Ren, M.-M. Cheng, A. Borji, Enhanced-alignment measure for binary foreground map evaluation, in: Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18), 2018, pp. 698–704.

BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100009

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Latency-aware automatic CNN channel pruning with GPU runtime analysis

Jiaqiang Liu, Jingwei Sun *, Zhongtian Xu, Guangzhong Sun

School of Computer Science and Technology, University of Science and Technology of China, Hefei, China

ARTICLE INFO

Keywords:
GPU runtime analysis
Inference latency
Channel pruning
Convolutional neural network

ABSTRACT

The huge storage and computation cost of convolutional neural networks (CNN) make them challenging to meet the real-time inference requirement in many applications. Existing channel pruning methods mainly focus on removing unimportant channels in a CNN model based on rule-of-thumb designs, using reduced floating-point operations (FLOPs) and parameter numbers to measure the pruning quality. The inference latency of pruned models is often overlooked. In this paper, we propose a latency-aware automatic CNN channel pruning method (LACP), which aims to search low latency and accurate pruned network structure automatically. We evaluate the inaccuracy of measuring pruning quality by FLOPs and the number of parameters, and use the model inference latency as the direct optimization metric. To bridge model pruning and inference acceleration, we analyze the inference latency of convolutional layers on GPU. Results show that the inference latency of convolutional layers exhibits a staircase pattern along with channel number due to the GPU tail effect. Based on that observation, we greatly shrink the search space of network structures. Then we apply an evolutionary procedure to search a computationally efficient pruned network structure, which reduces the inference latency and maintains the model accuracy. Experiments and comparisons with state-of-the-art methods on three image classification datasets show that our method can achieve better inference acceleration with less accuracy loss.

1. Introduction

Convolutional Neural Networks (CNNs) have demonstrated state-of-the-art achievements in various tasks, such as image classification [1], object detection [2], and image segmentation [3]. Such a success is built upon a large number of model parameters and convolutional operations. As a result, the huge storage and computation cost make these models difficult to be deployed on resource-constrained devices, such as phones and robots. To address this problem, a common approach is to use model compression techniques, including quantization [4], distillation [5], and pruning [6–9]. Among them, neural network pruning has been recognized as one of the most effective tools for compressing CNNs.

Neural network pruning methods aim to remove redundant weights in a dense model. According to the pruning granularity, these methods can be categorized into either weight pruning or channel pruning. In weight pruning, individual weights are zeroed out, leaving a sparse set of weight tensors. Weight pruning can significantly reduce the model size, but it also introduces irregular memory access, leading to very limited or even negative speedups on general-purpose hardware (e.g. CPU, GPU) [10]. Differing from weight pruning, channel pruning methods remove entire channels to compress the model. Since channel pruning only changes the dimension of weight tensors, the pruned model still adopts a dense format, which is well-suited to general-purpose hardware and off-the-shelf libraries. As a result, channel pruning can achieve better acceleration on inference performance than weight pruning.

Due to the promising performance improvement in model compression, channel pruning methods have been widely studied for many years. Existing methods use the reduced floating-point operations (FLOPs) and parameter numbers to measure the pruning quality by default. However, the inference latency of neural network is influenced by many factors, such as the network architecture, the implementation of operators, and the hardware property. Therefore, using FLOPs or the number of parameters as a proxy for inference latency is insufficient, and may lead the algorithm to sub-optimal result. For instance, Fig. 1 shows the relationship between FLOPs, model size, and inference latency of VGG16 network. We randomly prune channels in convolutional layers, then measure the pruned model's FLOPs, number of parameters, and inference latency. Results show that FLOPs or parameter reduction does not necessarily result in latency reduction. For example, the pruned model A has smaller FLOPs than model B, but shows larger inference latency. The same for model C and model D, the smaller model C shows larger inference latency. This observation motivates us to investigate a latency-aware channel pruning method, instead of only focusing on FLOPs or parameter numbers.

* Corresponding author.
E-mail addresses: jqliu42@mail.ustc.edu.cn (J. Liu), sunjw@ustc.edu.cn (J. Sun), xuzt@mail.ustc.edu.cn (Z. Xu), gzsun@ustc.edu.cn (G. Sun).

https://doi.org/10.1016/j.tbench.2021.100009
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 3 November 2021

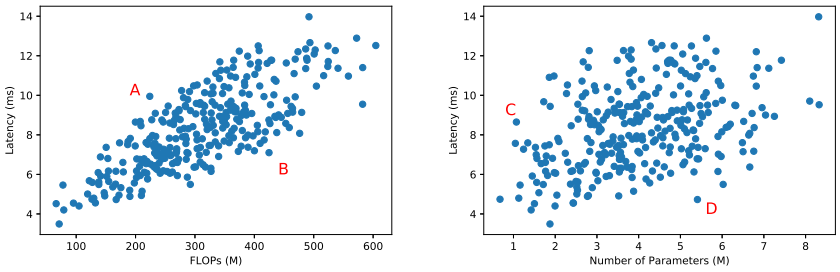


Fig. 1. The relationship between FLOPs, number of parameters, and inference latency of pruned models.

Another motivation of this work is that conventional channel pruning methods crucially rely on human expert knowledge and hand-crafted designs, and focus on selecting unimportant channels. Li et al. [9] take $l1$-norm as significance criteria to determine which channels will be pruned. Luo et al. [11] use the input of $(i+1)$-th layer to guide the pruning of ith layer. Lin et al. [12] rank channels with high rank of feature maps, then prunes the least important channels. However, Liu et al. [13] find that the pruned network can achieve the same accuracy no matter it inherits the weights in the original network or not. This study inspires us that the essence of channel pruning lies in finding optimal channel numbers in each layer, instead of selecting unimportant channels based on rule-of-thumb designs. Following that idea, Lin et al. [14] use artificial bee colony algorithm to search optimal pruned network structure. However, like many conventional channel pruning methods, Lin et al. [14] use the reduced FLOPs and parameter numbers to measure the pruning quality, the latency speedup of pruned model cannot be guaranteed.

In this paper, we propose a latency-aware automatic channel pruning (LACP) method. Differing from conventional methods, we take channel pruning in an automatic manner. Our method aims to search the optimal pruned network structure, i.e., the channel number in convolutional layers, instead of selecting important channels. An intuitive challenge in finding optimal network structure is that it is impractical to exhaustively searching all the possible combinations of pruned network structures. To make the algorithm feasible, effective shrinkage on search space is necessary. We first analyze the inference latency of pruned convolutional layers on GPU. Results show that the inference latency of convolutional layers presents a staircase pattern with the number of channels, which means the inference latency of a convolutional layer changes suddenly at certain channel number intervals. Based on this observation, we greatly shrink the search space of pruned structures. Then we apply an evolutional procedure to efficiently search low-latency and accurate network structure. For each candidate structure, we encode it to a vector $C = [c_1, c_2, c_3, \ldots, c_i]$, where c_i represents the channel numbers in ith convolutional layer. The fitness of candidate pruned network structure is measured in both model accuracy and inference latency. At each population, K candidates with highest fitness will survive to next population, crossover and mutation will take place in these survived structures to generate new structures. Finally, the best candidate is selected as the optimal pruned network structure.

Overall, the main contributions of this paper are as follows:

- We propose a latency-aware automatic channel pruning method LACP. Compared to conventional methods, LACP does not require hand-crafted designs on selecting unimportant channels. It focus on the inference latency speedup, instead of the FLOPs reduction.
- We analyze the inference latency of convolutional layers on GPU. Based on the analysis results, we greatly shrink the search space of pruned network structures, which enables efficient search of low-latency and accurate network structure.
- We conduct a detailed evaluation to compare the proposed method and existing methods on standard datasets. Results show that our method can achieve more latency reduction with less accuracy loss.

The rest of this paper is organized as follows. Section 2 reviews related works. Section 3 presents the proposed latency-aware automatic channel pruning method in detail. Section 4, show the experimental results and analysis. Finally, we draw the paper to a conclusion in Section 5.

2. Related work

Deep neural networks are usually over-parameterized [15,16], leading to huge storage and computation cost. There are extensive studies on compressing and accelerating neural networks. We classify current related research works into two major types: network pruning methods and neural architecture search (NAS) methods.

Pruning methods reduce the storage and computation cost by removing unimportant weights from the origin network. Existing pruning algorithms can be categorized into weight pruning and channel pruning. In weight pruning, individual weights are zeroed out. LeCun et al. [6] present the early work about network pruning using second-order derivatives as the pruning criterion. Han et al. [7] first propose iterative pruning, which prunes individual weights below a monotonically increasing threshold. Guo et al. [17] and Mocanu et al. [18] point out that some previously unimportant weights may tend to be important later. Inspired by this idea, LIU et al. [19] propose a trainable mask-based method to dynamically get sparse network during the training phase. Dettmers and Zettlemoyer [20] propose sparse momentum that used the exponentially smoothed gradients as the criterion for pruning and regrowth. A fixed percentage of parameters are pruned at each pruning step. Weight pruning can significantly reduce the model size. However, the non-structured random connectivity in DNN introduces irregular memory access. It adversely affects practical acceleration in hardware platforms [10]. Differing from weight pruning, channel pruning methods focus on removing the entire redundant channels. Li et al. [9] use $l1$-norm to determine the importance of channels. He et al. [8] formulate channel pruning as an optimization problem, which selects the most representative channels to recover the accuracy of pruned network with minimal reconstruction error. Luo et al. [11] use the next layer's input to guide the pruning of the previous layer. Lin et al. [12] use the feature map rank as sensitivity metric to prune the least important channels. Differing from these magnitude-based or sensitivity-based channel pruning methods, our work performs channel pruning in an automatic manner.

Although network pruning methods have achieved great success, they crucially rely on human expert knowledge and hand-crafted designs. Automatically optimizing the neural network architecture has been widely studied in recent years, known as neural architecture search (NAS). Prior works mainly sample a large number of networks from search space and train them from scratch to obtain a supervision signal, e.g. validation accuracy, for optimizing the sampling agent with reinforcement learning [21–23] or updating the population with an evolutionary algorithm [24]. Bender et al. [25] and Pham et al. [26] introduce weight-sharing paradigm in NAS to boost search efficiency, where all candidate sub-networks share the weights in a single one-shot model that contains every possible architecture in the search space. Liu et al. [27] relax the search space to be continuous with architecture

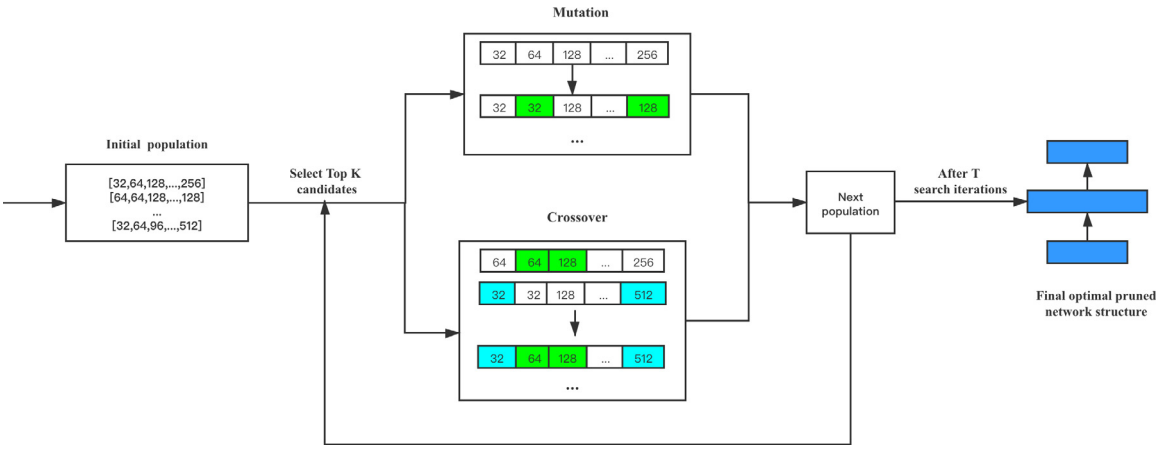


Fig. 2. The overall framework of LACP algorithm.

parameters and then efficiently optimized model parameters and architecture parameters together via gradient descent. Most prior NAS based pruning methods are implemented in a bottom-up and layer-by-layer manner. In contrast, our work mainly focuses on the optimal channel number of each layer.

3. Methodology

3.1. Overview

Cheng et al. [28] point out that convolutional layers take up most of the computation cost in convolutional neural networks. Our work focuses on reducing the channel number of convolutional layers to effectively compress the neural network. Fig. 2 presents the overall framework of our LACP algorithm. Consider a CNN model N that contains L convolutional layers. We refer to $C = [c_1, c_2, \ldots, c_L]$ as the network structure of N, where c_i is the channel number of the ith convolutional layer. We regard channel pruning as an optimal network structure search process, rather than manually designed strategies to remove unimportant channels. The algorithm aims to find a thinner network structure than the unpruned model, meanwhile, keeping a comparable accuracy. We adopt an evolutionary algorithm to achieve the goal of our search algorithm. A certain number of candidate network structures make up a population, the candidate network structures are evaluated using fitness. At each population, the best K candidate network structures are survive to the next population, and those survival candidates will produce new network structures through crossover and mutation. In the end, the best candidate network structure in the whole process is selected to be the optimal pruned network structure, we then fine-tune it to restore the accuracy. Formally, the algorithm is equivalent to solve an optimization problem as Eq. (1) shows.

$$C_{optimal} = \arg\max_{S} \quad F(C, W, D_{train}, D_{test}) \tag{1}$$

S is the search space of pruned network structures. $C \in S$ is the candidate network structure. W is the weight of pruned network, which is assigned from the pre-trained model. D_{train} and D_{test} represent the training data and testing data, respectively. The function F evaluates the fitness of candidate network structure to decide whether keeping current candidate in next population. The effectiveness and efficiency of the search algorithm mostly rely on the fitness evaluation and the search space definition. To find a low-latency and accurate pruned model, the fitness function should consider both model inference latency and test accuracy. For a convolutional neural network that contains L convolutional layers, the possible combination of pruned network structure can be $\prod_{i=1}^{L} c_i$, where c_i represents the channel numbers of ith convolutional layer in original dense model. It is impractical to exhaustively searching all the possible network structures, therefore, effective constraints on the search space are necessary. To solve these problems, we further describe the detailed implementation of our method in the following sections.

3.2. Search space definition

Exhaustively searching every possible pruned network structure is impractical. To make the search algorithm feasible, we need to shrink the search space. In this section, we conduct analysis on inference latency of convolutional layers to find an efficient search space design.

Convolutional layers are widely used in modern neural networks. A convolutional layer consists of a certain number of channels to extract data features. To reduce the computation cost of convolutional layer, channel pruning aims to remove a portion of channels. Intuitively, with the decrease of the channel number, the FLOPs of a convolutional layer will decrease linearly. However, due to the complex nature of convolutional layer's execution environment, its inference latency does not vary linearly with the FLOPs. To better understand the execution mechanism convolutional layer, we analyze how channel pruning affects the inference latency of convolutional layer. As Fig. 3 illustrates, the inference latency of convolutional layers shows a staircase pattern with different number of channels, which means with increasing a certain number of channels, there will be a significant step increase in latency. By analyzing the intrinsic mechanism of DNN deployment on GPU, this phenomenon can be explained. The computation of a convolutional layer is parallelized using multiple threads. These threads are first grouped into different blocks, then loaded to streaming multiprocessors (SMs) on a GPU. The maximum number of blocks loaded on one SM is determined by GPU's physical capacity. If the number of thread blocks in need exceeds the GPU capacity, then GPU will divide these thread blocks into multiple consecutive waves, and run these waves in sequence. Since the SMs are executed in parallel, one wave takes the same amount of time, no matter it is fully occupied or not. This phenomenon is called "GPU tail effect". For different channel number settings of a convolutional layer, their execution time can be very similar if they need the same amount of waves to compute. Therefore, with the increase of channel number, the computation cost of convolutional layer will increase. Once a critical point is exceeded, an extra wave is needed to finish the computation, which leads to a significant step increase in latency. Then, the inference latency of convolutional layer will change slowly, until the last wave is fully occupied.

Inspired by the "GPU tail effect" phenomenon, we can greatly shrink the search space of pruned network structure. Since the inference latency shows a staircase pattern, which means under a certain range

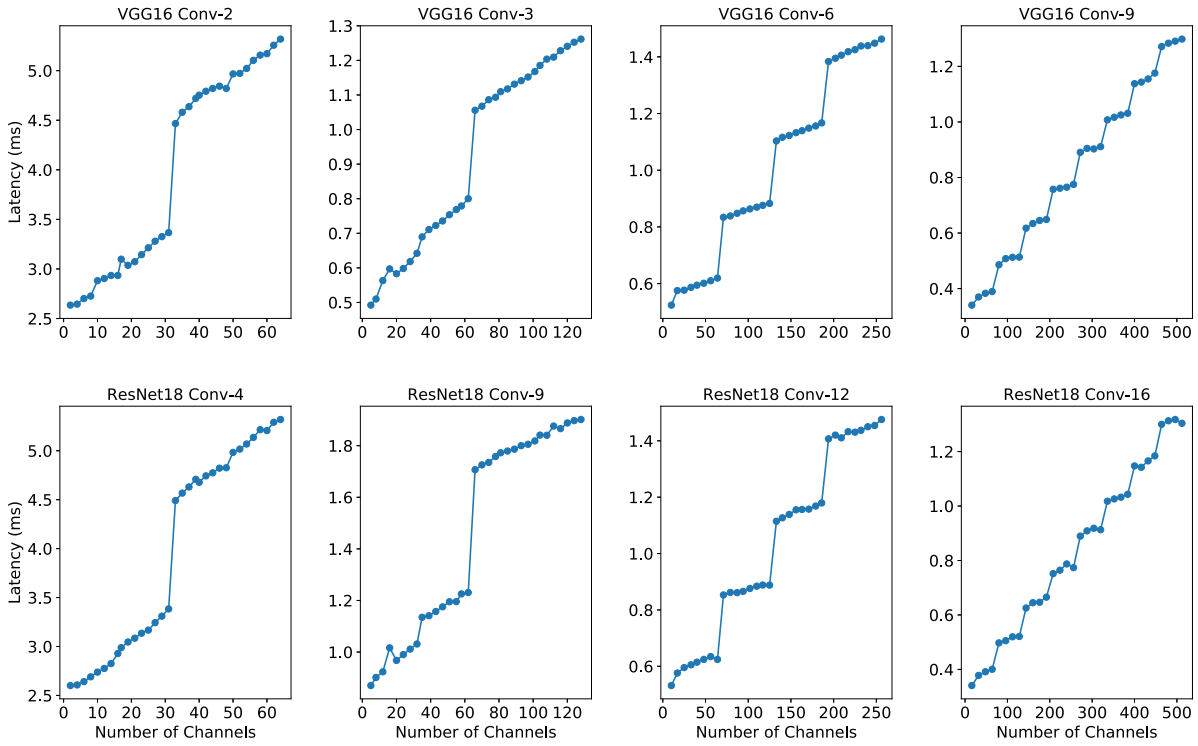


Fig. 3. Inference latency of convolutional layers with varying number of channels.

of channel number settings, the inference latency changes very slowly. Within a staircase step, set the channel number to the right endpoint, then we can maximize the representational capacity of network with a little latency cost.

Algorithm 1 Latency-aware Automatic Channel Pruning Algorithm

Input: Search Cycles: S, Population Size: N, Number of Mutation: M, Number of Crossover: C, Target latency: T
Output: Optimal pruned network structure C^*

G_0 = Random(N)
$G_{topK} = \emptyset$
for $i = 0; i < S; i++$ **do**
  G_{best} = Top1(G_i)
  if G_{best} better than C^* **then**
    $C^* = G_{best}$
  end if
  G_{topK} = TopK(G_i)
  $G_{mutation}$ = Mutation(G_{topK},M)
  $G_{crossover}$ = Crossover(G_{topK},C)
  $G_{i+1} = G_{mutation} + G_{crossover}$
end for
return C^*

We analyze the inference latency variation of different convolutional layers in VGG and ResNet. Results show that the width of the staircase step is a multiple of 32. For the first few convolutional layers, the width of the staircase step is 32. As the layers deepen, the max-pooling operation or the down-sampling operation makes the feature map smaller, thus a single GPU wave can compute more convolution operation. As a result, in the subsequent convolutional layers, the width of the staircase step can increase to 64 or 128. Heuristically, for each convolutional layer, we set its possible channel number in pruned network structure to a multiple of 32. Taking VGG16 as an example, the possible channel number in the sixth convolutional layer is $[32, 64, 96, 128, 160, 192, 224, 256]$, where the initial number of channels is 256. The other convolutional layers are also set up in the same way.

3.3. Optimal network structure search

In this section, we describe the detailed implementation of our LACP method. As Algorithm 1 shows, our method adopts evolutionary search as the overall framework. In the beginning, the initial population is randomly generated from the search space. Each sample in the population represents a pruned network structure, formalized as $C = [c_1, c_2, \ldots, c_L]$, where c_i represents the channel number in ith convolutional layer. At each population, the fitness of every candidate pruned network structure is evaluated as below:

$$fitness(C) = Acc(C) \times \left[\frac{Latency(C)}{T}\right]^w \tag{2}$$

$$w = \begin{cases} 0, & if \quad Latency(C) < T, \\ -1, & otherwise. \end{cases} \tag{3}$$

As Eq. (2) shows, both accuracy and inference latency are considered in the fitness evaluation, where $Acc(C)$ represents the test accuracy of the pruned network. $Latency(C)$ is the inference latency of the pruned network and T is the target latency, which is specified before running the algorithm. To measure the test accuracy of a network structure, it is very time-consuming to completely train and test the pruned model. In our implementation, we initialize the candidate pruned network with the pre-trained model, for a pruned network $C = [c_1, c_2, \ldots, c_L]$, the ith convolutional layer is initialized with c_i channels in the corresponding ith convolutional layer in the pre-trained model, which have larger $l1$-norm value. Then we train the pruned model with 2 epochs and evaluate its test accuracy. Besides, we add a latency constraint in the fitness function. Given a target latency, if the inference latency of pruned network is less than the target latency T, we simply use the test accuracy as the fitness value, otherwise, we penalize the fitness value with a coefficient less than 1. In such a mechanism, the algorithm will tend to select the model whose inference latency reaches the target latency constraint.

At each population, K candidate pruned network structures with largest fitness will survive to next population. Crossover and mutation will take place in these K candidate structures to generate new

structures. The objective of the crossover operation is to integrate excellent information from the parents. For example, given two preserved network structures:

[**32**, **32**, 128, 96, **32**, **192**, 224, 192], [64, 64, **96**, **64**, 160, 96, **320**, **288**]

one new structure will be generated by combining pieces of two parent structures:

[32, 32, 96, 64, 32, 192, 320, 288]

Mutation operation is used to promote population diversity. For example, given a network structure:

[**32**, 32, **128**, 96, **32**, 192, 224, 192]

its fragments are randomly changed, generating a new network structure:

[64, 32, 160, 96, 128, 192, 224, 192]

The preserved K candidate network structures and the structures generated by crossover and mutation form the next population. The algorithm will repeat such search iteration for S times. In the end, the best candidate is selected to be the optimal pruned network structure. We then fine-tune it to restore the accuracy.

4. Evaluation

In this section, we conduct experiments on standard datasets with different models to evaluate the performance of our algorithm.

4.1. Experimental settings

We implement our algorithm with Pytorch 1.5.0. All the experiments are run on NVIDIA GeForce RTX 2080 Ti GPU, which is made up of 4352 CUDA Cores and 68 SMs. We choose three standard image classification datasets (CIFAR-10, CIFAR-100, and Tiny-ImageNet) to evaluate our method. CIFAR-10 dataset consists of 60,000 colored images, which are classified into 10 classes. Each class has 5000 training images and 1000 testing images. Similar to CIFAR-10, CIFAR-100 contains 100 classes of images. Each class has 500 training images and 100 testing images. Tiny-ImageNet contains 100,000 images of 200 classes (500 for each class) colored images. Each class has 500 training images, 50 validation images, and 50 test images.

We use two kinds of models in our experiments: VGG and ResNet. VGG is a single-path network. The 16-layer model is adopted for compression. ResNet consists of a series of blocks, and there is a shortcut between two adjacent blocks. For dimensional matching in the pruned network, the last convolutional layer in each block will not be pruned. Two different depths of ResNet are adopted, including ResNet18 and ResNet34.

For each group of experiments, we report test accuracy, the reduction of network inference latency, the reduction of FLOPs, the reduction of parameter numbers, and the reduction of channel numbers as the performance metrics. We use the PyTorch expansion package thop to count the FLOPs and parameter numbers of network. To measure inference latency of network, we run the model 10 times for GPU warm up, then run the model 300 times with input batch size 128, and take the average inference time.

For each pre-trained model used in our experiments, we train it with 200 epochs using Stochastic Gradient Descent with momentum 0.9, and the batch size is set to 128, the initial learning rate is set to 0.1, which decays by 10 every 50 epochs. The weight decay is set to 1e-4.

4.2. Comparative methods

We compare our method with three representative algorithms to show its effectiveness.

- PFEC [9] is a representative traditional magnitude-based channel pruning method. PFEC calculates and sorts the $l1$-norm value of channels. Channels with smaller $l1$-norm value are less important, then those channels and corresponding feature maps are pruned.
- Thinet [11] formulates channel pruning as an optimization problem, and prunes channels of current layer based on statistics information computed from its next layer.
- ABCPruner [14] is a state-of-the-art automatic channel pruning method. It adopts artificial bee colony algorithm to search optimal pruned network structures. For ith convolutional layer of unpruned model that contains c_i channels, ABCPruner defines its search scope to $\{10\%c_i, 20\%c_i, 30\%c_i, \ldots, \alpha\%c_i\}$, where the maximum preserve percent α is used to restrict the width of pruned network, so that the FLOPs and parameter numbers can be reduced.

4.3. Evaluation results

We conduct our experiments on CIFAR-10, CIFAR-100 and Tiny-ImageNet datasets with VGG and ResNet models. To search for optimal pruned network structures, we set the number of search cycles to 10 and the population size is 30, so LACP searches 300 pruned network structures in the whole process. In each population, the numbers of new pruned network structures that generated from mutation and crossovers are both set to 15. In the end, we fine-tune the best pruned network structure for 200 epochs with a learning rate of 0.1, which is divided by 10 every 50 epochs. The weight decay is set to 1e-4. All algorithms use the same pre-trained model, and the number of fine-tuning epoch is set to 200. For a fair comparison with ABCPruner, we set its maximum searching number of pruned network structures to the same 300.

The experimental results are shown in Table 1, compared with PFEC, Thinet and ABCPruner, our method achieves better model inference acceleration, while maintaining similar or higher accuracy. It is worth noting that, as we have discussed before, more FLOPs or parameter reduction does not necessarily lead to better inference acceleration. Take CIFAR-100 dataset experiments as an example, ABCPruner-50% prunes VGG16 with 87.29% FLOPs reduction and 88.22% parameter reduction, while LACP-0.5 prunes 69.03% FLOPs and 81.14% parameters. However, LACP-0.5 achieves more latency reduction and a significantly higher accuracy than ABCPruner-50%. Another drawback of ABCPruner can be observed from the experimental results. ABCPruner compresses the model by limiting the maximum preserve channel number of convolutional layers. As a result, once the max preserve percent is small, the width of the pruned network is limited and the representational capacity of the pruned model is thus limited. To verify that point, we show a case study in Fig. 4. As shown in the figure, compared with ABCPruner, our method achieves less accuracy loss, while reducing the same percent of inference latency. As a supplementary analysis, we compare the pruned network structure of LACP and ABCPruner, result shows that our method preserves more channels in the first several convolutional layers, which is more important for neural network to extract feature information. On the contrary, the pruned network of ABCPruner has a narrower head structure due to the maximum preserve setting, leading to more accuracy loss.

5. Conclusion

In this paper, we propose a novel latency-aware automatic CNN channel pruning method. Differing from conventional channel pruning methods, our method get rid of selecting unimportant channels based on hand-crafted design, and search for optimal pruned network

Table 1
The experiment results on three datasets for different models. We compare LACP with three other methods.

Dataset	Model	Algorithm	Accuracy (%)	+/- (%)	Latency reduction (%)	FLOPs reduction (%)	Parameter reduction (%)	Channel reduction (%)
CIFAR10	VGG16	dense	93.02	0	0	0	0	0
		PFEC	92.52	−0.5	26.18	47.01	44.57	25
		LACP-0.7	**92.74**	**−0.28**	**33.63**	46.19	53.88	36.36
		Thinet	91.51	−1.51	39.18	61.06	57.88	33.55
		ABCPruner-90%	92.41	−0.61	46.59	68.82	78.39	51.23
		LACP-0.5	**92.62**	**−0.4**	**48.97**	70.08	72.42	50
		ABCPruner-50%	90.11	−2.91	59.18	87.55	87.81	67.57
		LACP-0.4	**91.62**	**−1.4**	**59.56**	85.04	94.46	72.73
	ResNet18	dense	94.28	0	0	0	0	0
		PFEC	92.01	−2.27	17.56	35.26	48.51	18.67
		ABCPruner-90%	94.02	−0.26	0.19	25.44	30.39	12.48
		ABCPruner-50%	93.59	−0.69	18.61	60.27	60.07	24.98
		LACP-0.5	**94.34**	**+0.06**	**22.29**	44.96	69.14	22.33
		Thinet	94.36	+0.08	17.98	37.84	37.51	15.29
		LACP-0.7	**94.37**	**+0.09**	**21.3**	41.15	61.71	20.67
CIFAR100	VGG16	dense	69.78	0	0	0	0	0
		PFEC	69.45	−0.33	26.14	47	44.43	25
		LACP-0.7	**70.54**	**+0.76**	**30.23**	50.72	63.17	39.39
		Thinet	69.32	−0.46	40.63	63.42	60.01	35.27
		ABCPruner-90%	69.06	−0.72	28.82	54.11	72.41	41.5
		ABCPruner-50%	65.95	−3.93	39.14	87.29	88.22	66.17
		LACP-0.5	**69.42**	**−0.36**	**49.4**	69.03	81.14	55.3
	ResNet34	dense	74.86	0	0	0	0	0
		PFEC	69.52	−5.34	20.91	39.63	48.93	21.05
		Thinet	74.59	−0.27	19.94	38.09	37.75	16.96
		ABCPruner-90%	74.58	−0.28	25.05	63.65	61.15	27.1
		ABCPruner-50%	74.18	−0.68	23.56	67.92	68.33	30.06
		LACP-0.5	**74.59**	**−0.27**	**27.91**	56.91	69.94	29.32
Tiny-ImageNet	ResNet18	dense	57.87	0	0	0	0	0
		PFEC	56.49	−1.38	17.68	35.26	48.09	18.67
		Thinet	56.53	−1.34	17.81	37.45	37.19	15.29
		ABCPruner-90%	56.55	−1.32	2.69	38.9	34.37	15.27
		ABCPruner-50%	55.23	−2.64	22.35	67.94	71.77	28.08
		LACP-0.7	**56.87**	**−1**	**22.15**	50.16	68.08	24.67
	ResNet34	dense	59.19	0	0	0	0	0
		PFEC	58.98	−0.21	21.24	39.64	48.81	21.05
		Thinet	59.05	−0.14	18.57	37.91	37.66	16.96
		LACP-0.7	**59.02**	**−0.17**	**24.45**	52.28	67.1	27.07
		ABCPruner-90%	58.58	−0.61	10.44	42.21	52.34	20.84
		ABCPruner-50%	57.96	−1.23	24.17	68.09	73.76	31.58
		LACP-0.5	**58.64**	**−0.55**	**27.12**	61.89	76.58	31.58

Note: LACP-α means we set the target latency to $\alpha \times L$, where L is the unpruned model's inference latency. ABCPruner-β means the maximal preserved channel number in each convolutional layer is $\beta \times C$, where C is the original channel number in that layer.

structure automatically. By analyzing the inference latency of pruned networks, we indicate that neither FLOPs nor the number of parameters can accurately represent the real inference acceleration. Besides, we analyze the execution mechanism of convolutional layers on GPU. Results show that the inference latency of convolutional layers presents a staircase pattern with different number of channels. Based on this observation, we greatly shrink the combinations of network structure, enabling efficient search of low-latency and accurate pruned network. We conduct extensive evaluations to compare our method with existing studies on public datasets, and report the real latency metric. Experimental results show that our method can achieve better inference acceleration, while maintaining higher accuracy.

Although we have achieved desired pruning effect on our experiments, our method can be further improved. As we discussed before,

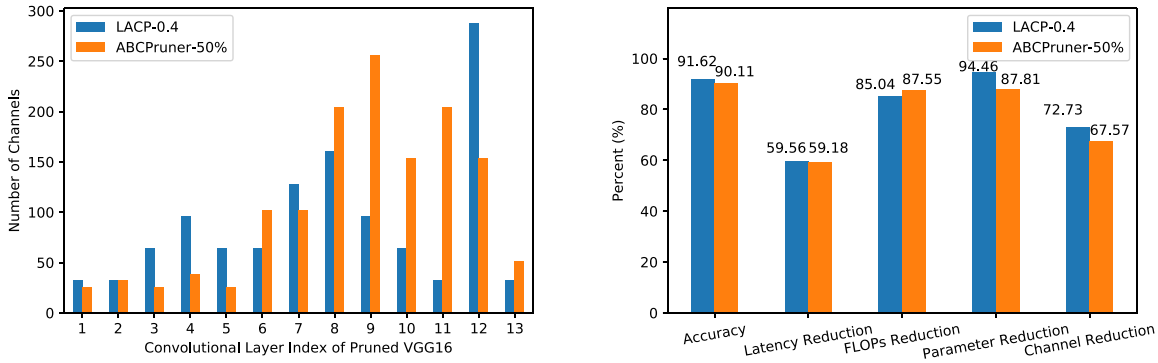


Fig. 4. The pruning results of LACP and ABCPruner for VGG16 on CIFAR-10 dataset.

we shrink the search space of pruned network structure through the analysis of the GPU tail effect. However, our analysis is based on empirical profiling. A more thorough and general investigation of the GPU tail effect could be helpful. Besides, how to generalize our method to different hardware platforms is also worth studying in future work.

Acknowledgments

This work is supported by National Natural Science Foundation of China (No. 61772485). It is also funded by Youth Innovation Promotion Association of Chinese Academy of Sciences (CAS) and JD AI research. Experiments in this study were conducted on the supercomputer system in the Supercomputing Center of University of Science and Technology of China.

References

[1] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.

[2] Ross Girshick, Jeff Donahue, Trevor Darrell, Jitendra Malik, Rich feature hierarchies for accurate object detection and semantic segmentation, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 580–587.

[3] Jonathan Long, Evan Shelhamer, Trevor Darrell, Fully convolutional networks for semantic segmentation, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2015, pp. 3431–3440.

[4] Itay Hubara, Matthieu Courbariaux, Daniel Soudry, Ran El-Yaniv, Yoshua Bengio, Quantized neural networks: Training neural networks with low precision weights and activations, J. Mach. Learn. Res. 18 (1) (2017) 6869–6898.

[5] Chenglin Yang, Lingxi Xie, Chi Su, Alan L. Yuille, Snapshot distillation: Teacher-student optimization in one generation, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 2859–2868.

[6] Yann LeCun, John S Denker, Sara A Solla, Richard E Howard, Lawrence D Jackel, Optimal brain damage, in: NIPs, Vol. 2, Citeseer, 1989, pp. 598–605.

[7] Song Han, Jeff Pool, John Tran, William Dally, Learning both weights and connections for efficient neural network, in: C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, R. Garnett (Eds.), Advances in Neural Information Processing Systems, Vol. 28, Curran Associates, Inc., 2015.

[8] Yihui He, Xiangyu Zhang, Jian Sun, Channel pruning for accelerating very deep neural networks, in: Proceedings of the IEEE International Conference on Computer Vision, 2017, pp. 1389–1397.

[9] Hao Li, Asim Kadav, Igor Durdanovic, Hanan Samet, Hans Peter Graf, Pruning filters for efficient convnets, 2016, arXiv preprint arXiv:1608.08710.

[10] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, Hai Li, Learning structured sparsity in deep neural networks, in: Proceedings of the 30th International Conference on Neural Information Processing Systems, 2016, pp. 2082–2090.

[11] Jian-Hao Luo, Jianxin Wu, Weiyao Lin, Thinet: A filter level pruning method for deep neural network compression, in: Proceedings of the IEEE International Conference on Computer Vision, 2017, pp. 5058–5066.

[12] Mingbao Lin, Rongrong Ji, Yan Wang, Yichen Zhang, Baochang Zhang, Yonghong Tian, Ling Shao, Hrank: Filter pruning using high-rank feature map, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2020, pp. 1529–1538.

[13] Zhuang Liu, Mingjie Sun, Tinghui Zhou, Gao Huang, Trevor Darrell, Rethinking the Value of Network Pruning, in: International Conference on Learning Representations, 2018.

[14] Mingbao Lin, Rongrong Ji, Yuxin Zhang, Baochang Zhang, Yongjian Wu, Yonghong Tian, Channel Pruning via Automatic Structure Search, in: Proceedings of the International Joint Conference on Artificial Intelligence, IJCAI, 2020, pp. 673–679.

[15] Emily Denton, Wojciech Zaremba, Joan Bruna, Yann LeCun, Rob Fergus, Exploiting linear structure within convolutional networks for efficient evaluation, in: Proceedings of the 27th International Conference on Neural Information Processing Systems-Volume 1, 2014, pp. 1269–1277.

[16] Jimmy Ba, Rich Caruana, Do deep nets really need to be deep? in: NIPS, 2014.

[17] Yiwen Guo, Anbang Yao, Yurong Chen, Dynamic network surgery for efficient DNNs, in: NIPS, 2016.

[18] Decebal Constantin Mocanu, Elena Mocanu, Peter Stone, Phuong H Nguyen, Madeleine Gibescu, Antonio Liotta, Scalable training of artificial neural networks with adaptive sparse connectivity inspired by network science, Nature Commun. 9 (1) (2018) 1–12.

[19] Junjie LIU, Zhe XU, Runbin SHI, Ray C. C. Cheung, Hayden K.H. So, Dynamic sparse training: Find efficient sparse network from scratch with trainable masked layers, in: International Conference on Learning Representations, 2020.

[20] Tim Dettmers, Luke Zettlemoyer, Sparse networks from scratch: Faster training without losing performance, 2019, arXiv preprint arXiv:1907.04840.

[21] Bowen Baker, Otkrist Gupta, Nikhil Naik, Ramesh Raskar, Designing neural network architectures using reinforcement learning, in: International Conference on Learning Representations, 2018.

[22] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, Quoc V Le, Learning transferable architectures for scalable image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2018, pp. 8697–8710.

[23] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, Mark Sandler, Andrew Howard, Quoc V Le, Mnasnet: Platform-aware neural architecture search for mobile, in: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 2019, pp. 2820–2828.

[24] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Jie Tan, Quoc V Le, Alexey Kurakin, Large-scale evolution of image classifiers, in: International Conference on Machine Learning, PMLR, 2017, pp. 2902–2911.

[25] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, Quoc Le, Understanding and simplifying one-shot architecture search, in: International Conference on Machine Learning, PMLR, 2018, pp. 550–559.

[26] Hieu Pham, Melody Guan, Barret Zoph, Quoc Le, Jeff Dean, Efficient neural architecture search via parameters sharing, in: International Conference on Machine Learning, PMLR, 2018, pp. 4095–4104.

[27] Hanxiao Liu, Karen Simonyan, Yiming Yang, Darts: Differentiable architecture search, in: International Conference on Learning Representations, 2018.

[28] Jian Cheng, Pei-song Wang, Gang Li, Qing-hao Hu, Han-qing Lu, Recent advances in efficient computation of deep convolutional neural networks, Front. Inf. Technol. Electron. Eng. 19 (1) (2018) 64–77.

BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100010

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Fallout: Distributed systems testing as a service

Matt Fleming *, Guy Bolton King, Sean McCarthy, Jake Luciani, Pushkala Pattabhiraman

DataStax Inc., United States of America

ARTICLE INFO

Keywords:
Distributed systems
Databases
Performance
Apache Cassandra, Pulsar

ABSTRACT

All modern distributed systems list performance and scalability as their core strengths. Given that optimal performance requires carefully selecting configuration options, and typical cluster sizes can range anywhere from 2 to 300 nodes, it is rare for any two clusters to be exactly the same. Validating the behavior and performance of distributed systems in this large configuration space is challenging without automation that stretches across the software stack. In this paper we present Fallout, an open-source distributed systems testing service that automatically provisions and configures distributed systems and clients, supports running a variety of workloads and benchmarks, and generates performance reports based on collected metrics for visual analysis. We have been running the Fallout service internally at DataStax for over 5 years and have recently open sourced it to support our work with Apache Cassandra, Pulsar, and other open source projects. We describe the architecture of Fallout along with the evolution of its design and the lessons we learned operating this service in a dynamic environment where teams work on different products and favor different benchmarking tools.

1. Introduction

Building databases and distributed systems with high performance requires thorough testing and benchmarking. The earlier that performance testing can be done in the development process, the cheaper issues are to fix [1].

Software teams are now expected to use techniques such as CI/CD [2] to deliver frequent releases to users. For many types of products, including distributed systems and databases, users also expect the systems to be resilient, never lose data, and always achieve high performance. Strong automated testing tools are required to reduce development time and deliver stable products.

Automating the testing of complex distributed systems requires tightly controlling every aspect of the software: from operating system configurations to application-level tuning. Fallout evolved into a full-stack orchestration system, enabling us to test and tweak all aspects of the distributed system under test. Fallout is a service that deploys hardware resources, configures the operating system and distributed application, runs a workload or benchmark on the cluster and gathers the results for analysis. Through a rich YAML-based configuration, every aspect of the system and application can be detailed and parameterized.

We use Fallout to run a mixture of manual and automated testing and Fallout executes around 200 tests every day. These tests have been used to verify the performance of new features and optimizations, uncover functional and performance regressions before they have shipped to customers, and reproduce issues that were discovered in the field. Recently, we have added support for chaos testing too. Automated testing is driven by Jenkins which is the CI tool of choice for the majority of our teams. The rest of this paper is organized as follows. In Section 2 we discuss our rationale for building Fallout along with the existing tools at the time. In Section 3 we present a high-level overview of the Fallout design and dive down into the details in Section 4. Section 5 illustrates how Fallout test run results are displayed for users. Lessons learned, related work, and conclusions are covered in Sections 6–8.

2. Background

Five years ago, we had a server-based performance testing and comparison tool named cstar_perf that could bootstrap Apache Cassandra onto an already provisioned cluster, run a workload against it, and plot the performance results on a web page. The workload was composed via a web UI and used cassandra-stress [3] to generate load on the cluster. cstar_perf gave us some flexibility in that the Cassandra installation could be configured in a number of ways but it also came with many limitations. The size of the cluster was fixed and could not be changed. The workload consisted of a number of linear steps, each of which could invoke one of a small number of tools. This gave us neither the modularity we needed to support diverse teams with

* Corresponding author.
E-mail addresses: matt@codeblueprint.co.uk (M. Fleming), guy@waftex.com (G.B. King), sean.mccarthy@datastax.com (S. McCarthy), jake@datastax.com (J. Luciani), pushkala.pattabhiraman@datastax.com (P. Pattabhiraman).

https://doi.org/10.1016/j.tbench.2021.100010
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 4 November 2021

different preferences for benchmarks, tools, and workloads, nor the parallelism required to run multiple tests at once.

Fallout was conceptualized to address these limitations. There was a clear need to create a system that could seamlessly stitch together a plethora of tools and systems built by internal teams so they could be made to work together while remaining tool agnostic. It was also desired to provide the ability to support any testing environment, be it public or private cloud. Since Fallout needed to test distributed systems, it needed to support scenarios involving multiple server/client clusters and a myriad topology configurations as well as tools that disrupt normal operation such as throttling the network bandwidth and deleting cluster data. While cstar_perf gave us the ability to analyze performance for a single test run we also wanted the ability to generate better insights into results by gathering artifacts from those clusters. To encourage adoption from a diverse set of stakeholders, Fallout was required to be intuitive, simple, and self-documenting. The target user group ranged from seasoned database engineers to non-developers. Hence, Fallout needed to use a declarative language that was simple for non-developers to write tests in. The artifacts involved in Fallout were required to be persisted and versioned for future reference. All of the test configurations, results, and artifacts were to be stored in a single place so that everything could be trivially shared within our organization.

In summary, Fallout addresses the following engineering challenges:

- Build a testing service that provides a single interface for multiple teams to run test and benchmarking tools
- Use simple test configuration files to deploy tests into distributed systems that accurately reflect real-world configurations
- Extract and preserve test run artifacts for later analysis
- Ease of use for both developers and non-developers.

The initial version of Fallout used Jepsen [4] as the workload execution tool. This was largely a pragmatic choice since Jepsen was well-known in the original Fallout team and using it avoided the need to reinvent the wheel by creating a brand new tool. Fallout extended Jepsen's correctness testing features by creating operation logs during test runs and allowing pass/fail checks to be run on test completion. Over time, Fallout has evolved into a more performance-focussed service but still retains a couple of the original Jepsen concepts such as Checkers and operation logs.

3. Architecture

Fallout runs as a single service and exposes a REST API which is accessible via a Python client API and command-line application, and a web UI which users can access using a web browser. Fallout supports multiple concurrent users while enabling each user to store and execute tests independently. Read-only access of test configurations is granted for other user's test configurations which is especially handy when multiple engineers are working on the same test since they can clone the test configuration and collaborate. The Python client API and command-line application are used by Continuous Integration tools to submit tests to Fallout's job queue. Once a job reaches the front of the queue and hardware resources become available, Fallout deploys and configures the test's infrastructure (setup), runs the workload, then collects test artifacts and tears down the infrastructure once the test is complete. Results are published to a central server for analysis. Fallout maintains logs of all the operations involved in each step of a test. An overview of Fallout's architecture is given in Fig. 1.

3.1. Cluster deployment

Test jobs are submitted to Fallout which internally schedules them based on the available hardware resources in the infrastructure. To provision the cluster in DataStax's data center (private and public), Fallout relies on a proprietary infrastructure tool, *ctool*. The open-source version of Fallout includes support for using Google Kubernetes Engine (GKE) to manage clusters. *ctool* is cloud provider agnostic and abstracts the provisioning and deployment steps of Fallout tests so that users only need to specify high-level requirements such as cloud provider, instance type and region in a YAML test config file. Fallout handles provisioning machines with GKE using the gcloud tool [5] and includes logic for configuring resources that might be required for the test. For example, Fallout will automatically add persistent storage to the Kubernetes cluster so that test run artifacts can be downloaded from the cluster once the test completes. Users can also specify custom manifests in their test config files which configure cluster resources. Fallout monitors all logs from the cluster and can display them in real time via the Fallout web UI. Once the test completes and the cluster is torn down, those logs are permanently stored on the Fallout server for offline analysis.

Running performance tests against clusters requires applying workloads and benchmarks. Fallout also handles provisioning and configuring client nodes that generate these workloads. Metrics and statistics are gathered for all the client and server nodes via a dedicated observer instance that is configured for the test run in exactly the same way as both client and server: via the test config. In each test, the observer instance operates for the duration of the test run and allows Fallout users to monitor metrics from the client and server in real time. Watching the live observer node is frequently important when re-running a configuration that is known to exhibit performance issues and the observer can be used to detect when a cluster has entered a bad state of performance. At the end of the test, the observer metrics are archived and saved locally to the Fallout server and available on the test run web page. This enables analysis after the test execution has completed. Lastly, Fallout tears down the infrastructure after the test completes thereby returning the allocated resources to the cloud.

3.2. Application installation, configuration, and execution

The specific method used to install applications such as Apache Cassandra and Pulsar varies between releases and engineers are often unaware of the differences. Fallout automatically handles installation and system configuration no matter which version is specified for the test. Installation involves extracting tarballs on each node and updating the *cassandra.yaml* config file to use the additional larger disks from the deployment phase — Fallout also needs to handle configuration of each individual node to work in the cluster. For instance, Apache Cassandra requires the IP addresses of seed nodes in a cluster to be known and listed in every node's config file.

Benchmarking tools including profilers and metrics collection agents are installed on the client nodes by Fallout. Fallout supports a wide variety of tools though only a few of them are currently available in the open source version. We plan on contributing more in the future. Each benchmark can be configured using the same YAML interface and individual options contained in the config will be specific to each benchmark. As Fallout has gained popularity, more and more benchmarks have been added since it is common for different teams to favor different benchmarks. For example, YCSB is a popular open source benchmark often used to compare relative performance of NoSQL database management systems. The DataStax Stargate team use YCSB to benchmark Stargate's Document API performance for every release.

Fallout was designed to accommodate this heterogeneity while still providing the same interface to users. This has an added benefit — because the complexity of supporting multiple benchmarks is primarily hidden inside of Fallout, external services that use Fallout can automatically work with any benchmark, reducing the effort required to support new teams and new tools.

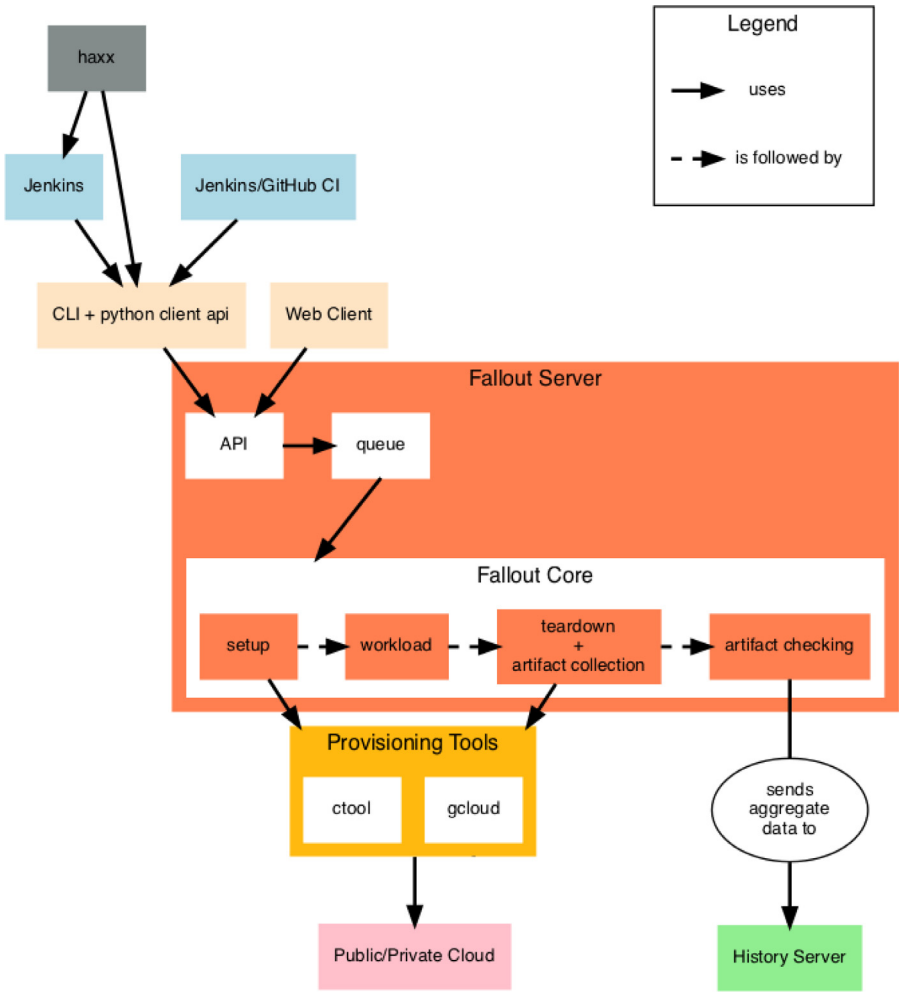


Fig. 1. Fallout architecture.

3.3. Artifact collection and analysis

To aid with post-run analysis, Fallout saves a range of logs and other artifacts locally on the central server so that they can be inspected after the test run has finished. This is the most common situation for analyzing metrics and other benchmark data collected as part of a manual test run. For automated test analysis, Fallout will push the archived metrics to a central Grafana server where other tools run further analysis on them, including Hunter, our statistical significance detection tool that uses change point detection [6]. Fallout uses artifact checkers to inspect the logs for specific error or warning messages and allows the test run to be marked as failed if any are present. Other artifact checkers are used to post-process files. For example, the hdrtool artifact checker merges HDR files [7] retrieved from multiple clients and produces aggregated metrics.

Even when a performance regression is automatically detected by Hunter, engineers might need to look at the metrics that were collected during the test run to understand the cause of the performance issue. When a user needs to check the observer metrics they can simply download the archived artifact from Fallout, extract it to their machine and use a docker image containing Grafana to display the metrics.

3.4. Integration with CI

Automated testing with Fallout is primarily driven via Jenkins. Jenkins uses the Fallout API to launch test runs whenever a pull request from GitHub is successfully built. We have configured Jenkins so that it links directly to the Fallout test run for a given job (GitHub pull request). Being able to navigate from the Jenkins job to the Fallout test run acts as a breadcrumb trail and simplifies post-test run analysis.

We also run nightly and weekly performance tests that are scheduled outside of the GitHub PR-merge workflow but still rely on Jenkins to call the Fallout REST APIs. Haxx is a git repository that acts as a central location for storing Fallout test configs since Fallout itself does not provide any kind of version control other than A) storing a read-only copy of the YAML file from previous test runs and B) the most recent version. Haxx also provides templating for Fallout YAML files where common configuration snippets, such as optimal Apache Cassandra configuration options, can be stored in template files and reused across test configs. This allows us to significantly cut down on the boiler plate code required to support a large number of tests where only the machine size, version of Apache Cassandra, or benchmark config is different. Better still, templates allow users to take advantage of known-good performance options which ensures that they do not waste their time analyzing performance issues that were the results of poorly configured tests.

4. Implementation

Since Fallout was originally created as a wrapper around Clojure, Fallout had to be written in another JVM language to make development easier and Java was selected as the target language. Despite Fallout development primarily being the responsibility of a very small team, Fallout has benefited from a large number of contributors and

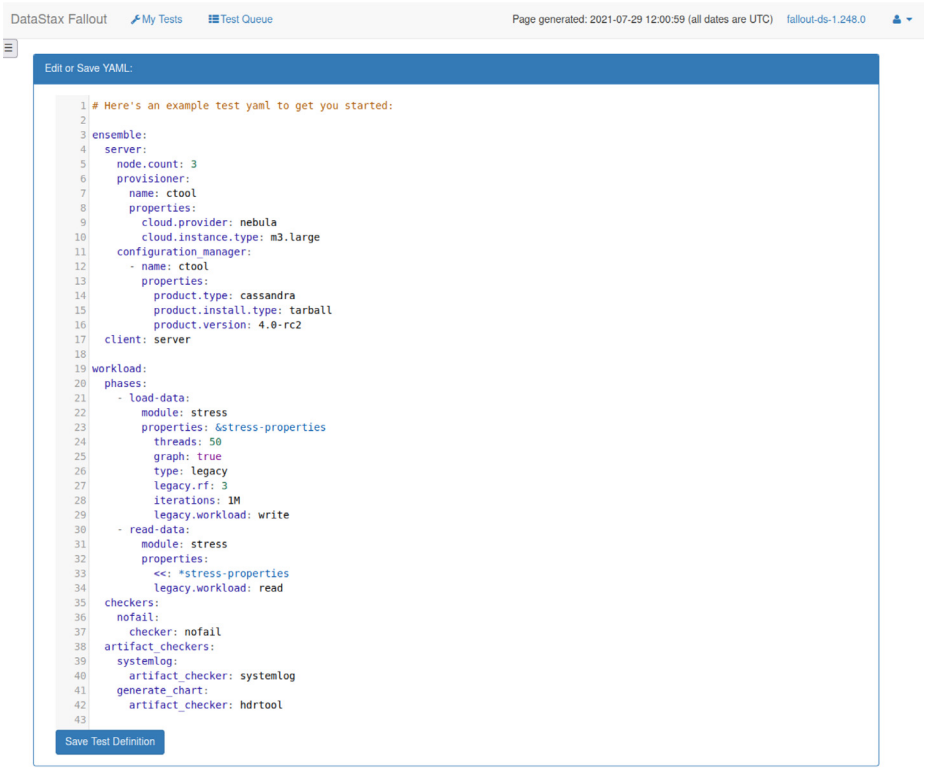


Fig. 2. Example Fallout test configuration.

since Java is widely used inside of DataStax the choice of programming language is no doubt a contributing factor.

A similar desire to make the configuration interface as welcoming for users as possible led to the decision to use YAML for the configuration files. YAML syntax is easy to learn for new users and YAML syntax highlighting is readily available in IDEs and editors. Fallout's web UI provides a built-in YAML editor with syntax checker for creating and modifying test configurations.

4.1. Test configuration files

Fallout test runs are driven by a single YAML configuration file that has a number of required entries. Tests describe machines and services running on those machines. A node is a resource with services running on it. An example of a node is a single Apache Cassandra node within a multi-node cluster. NodeGroups are collections of nodes. An example of a NodeGroup is an Apache Cassandra cluster. An ensemble is a set of NodeGroups with a specified role and test run configuration files expose this concept to the user. The list of ensemble roles is:

- Server: A distributed server or cluster such as Apache Cassandra
- Client: A benchmark or workload
- Observer: A monitoring server such as graphite
- Controller: An external controller such as Jepsen.

Fig. 2 shows an example of a Fallout test configuration file.

Workloads are built from one or more phases which are the basic unit of concurrency in Fallout. Each phase can run one or more modules and specifying more than one module executes them in parallel. Phases are always run sequentially and a phase will not start executing until the previous phase completes.

4.2. Test provisioning lifecycle

Each NodeGroup in a test transitions through a number of states when the test executes. There are three types of states: Unknown, Transitional, and Runlevel. Transitional states are entered when a NodeGroup moves from one state to another. Runlevel states represent steady states where a NodeGroup is not currently transitioning and are modeled on the UNIX runlevel concept — NodeGroups progress to higher levels where each level has more capabilities than the previous one. State transitions perform provisioning and configuration actions on the NodeGroup and the current state of a NodeGroup is used by Fallout to guarantee only legal transitions between states can occur. Using the state machine, it is impossible for Fallout to configure a NodeGroup before it is provisioned. If any errors are encountered during a transition, for example if Fallout fails to install the distributed application, the NodeGroup will enter the FAILED state and the entire test run will fail.

A transition diagram is presented in Fig. 3. The oval states on the left and right represent Transitional states, and the rectangular states in the center represent runlevel states.

4.3. Modules, providers, and configuration managers

Adding support for a new benchmark or tool to Fallout requires adding 3 new components to the Fallout code base: a module, a provider, and a configuration manager. Providers allow access to a service or tool via an API and these are invoked by the Fallout test harness to run commands on the node. For example, the *NoSqlBenchPodProvider* is responsible for executing the nosqlbench [8] benchmark on a Kubernetes pod. Providers can also have dependencies on other Providers which makes it possible to express that a benchmark should only be available when running on a Kubernetes cluster, for example. Fallout supports Chaos Mesh [9], a tool for running chaos experiments on a cluster, however since it is only available on Kubernetes Fallout will refuse to deploy it into any environment that does not meet the Kubernetes Provider dependency.

Configuration Managers are responsible for configuring and unconfiguring software running on nodes as well as starting and stopping services. Additionally, Configuration Managers register Providers with

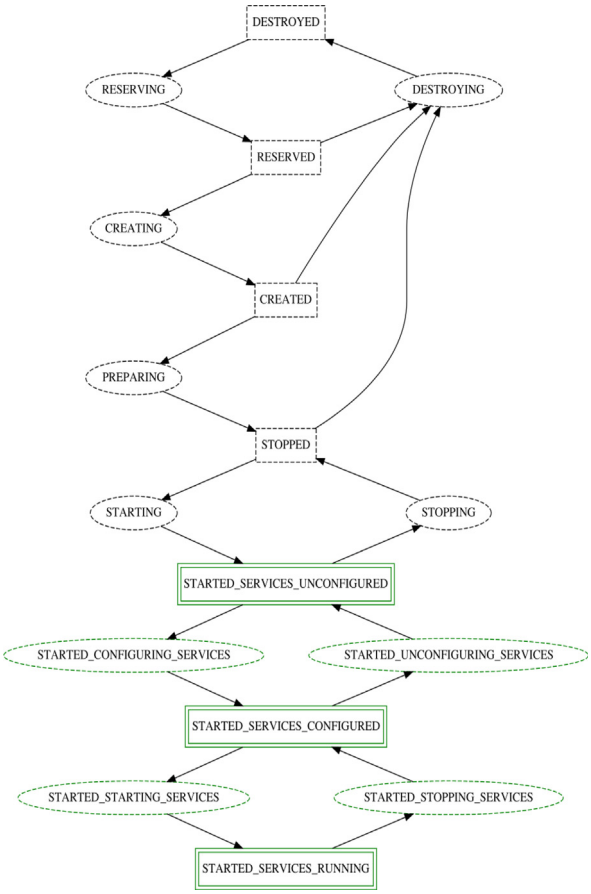


Fig. 3. NodeGroup transition diagram.

nodes, making the associated services available to Modules in a test workload.

Finally, Modules are the user-facing component of benchmarks. Modules define the supported keywords and parameters that can be passed to the benchmark via YAML configuration files. Since this provides a layer of indirection between the test config and the benchmark itself, it is common for only a subset of the parameters supported by the benchmark to be supported in Fallout, though if users want maximum flexibility there is usually an *args* parameter that passes through parameters without any kind of filtering.

While Fallout supports a number of different benchmarks, one lesson we have learned is that users need some kind of back-stop module that allows them to manually run benchmarks for which no support currently exists. A bash module is provided to fill the gap where users need to run a simple script or download a benchmark to a node and run it manually. Extended use of the bash module is frowned upon because we have seen it lead to difficult to understand shell scripts that are copied between test configs.

4.4. Checkers and artifact checkers

Once a test has completed, Fallout needs a way to validate that the system under test behaved correctly for the duration of the test. Checkers are the component in Fallout responsible for ensuring that no errors occurred during the test that might invalidate the results. This is important for performance tests even though the checkers do not perform any kind of performance analysis themselves — any performance results from tests that fail basic checks are likely to be invalid because the test was not run under real-world conditions. *NoFailChecker* is an example of a very basic checker that simply checks that none of the Fallout operations that ran during a test failed. The history of operations is passed to checkers as an argument so that they can run arbitrary checks against it. There is no limit to the number of checkers that can be included in a Fallout test and a test will only pass if all checkers pass.

A related concept is the artifact checker which performs the same kind of validation process on artifacts that are collected after the test run completes. A frequently used artifact checker for Apache Cassandra tests is *SystemLogChecker* which checks Cassandra's *system.log* for the presence of user-specified patterns such as log messages containing "ERROR" or "WARN".

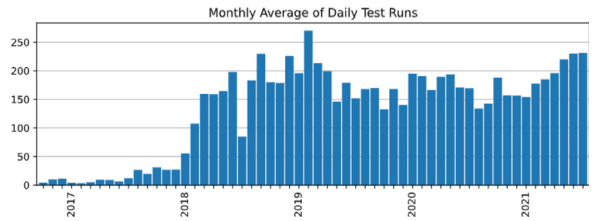


Fig. 4. Average daily test runs by month.

Table 1
Test run statistics.

Year	Total	Mean	Min	Max
2016	759	8	0	44
2017	5512	15	0	101
2018	58625	160	0	562
2019	64633	177	0	361
2020	62616	171	0	421
2021	39945	197	34	349

4.5. Test queue

When Fallout was first launched, test runs were executed as soon as they were submitted. As Fallout grew in popularity, contention for VMs on our internal infrastructure resulted in tests failing. A simple queueing mechanism was added to fix this that checked for VM availability before attempting to submit a claim for resources. It has been tweaked over time to become more robust and fair. For example, it now favors users with fewer running test runs to prevent anyone monopolizing the system. With this in place, Fallout now handles over 200 test runs a day. Fig. 4 shows the mean number of daily test runs per month. Table 1 shows additional yearly statistics for this time period.

4.6. REST API

The Fallout command-line client is built using a Python library for accessing the Fallout REST API. Making this API available instead of only providing access to Fallout via the web UI has helped many other services leverage Fallout's test running capabilities and has no doubt led to Fallout's rise in popularity at DataStax. Recently, we have used Fallout's API and Python library to drive Fallout tests using *pytest* [10] for a new project.

5. Results

Once one or more benchmarks have been run on a cluster, we use multiple tools to display benchmark and OS metrics. Fallout includes a built-in way to display client-side benchmark metrics as part of the web UI but we usually collect many more metrics for runs such as Apache Cassandra and OS metrics. We use a central Grafana server, known as the history server, to display all of the historical metrics that are accumulated during test runs.

5.1. Performance reports

Fallout can generate performance reports which visualizes the metrics gathered from a single test run. Performance reports are built on top of HdrHistogram datasets [7]. The HdrHistogram format is a de facto standard for histogram data and implementations are available for many benchmarking tools. A feature that we use heavily is the ability to merge HdrHistogram data across multiple clients which makes it possible to split load across nodes, collect individual HDR files, and combine them to summarize the total load on the cluster. Finally, HDR files capture both throughput and latency in a single file format. Fig. 5 shows an example of a performance report.

Metrics are displayed using time series data which is invaluable for database workloads where the workload does not have a consistent behavior, e.g. where it changes as memory-resident data structures fill up and are flushed to disk. Being able to see metrics for the entire test run duration makes it easier for users to spot situations where the test hits an unexpected state. The metric data used to create graphs can be altered by selecting an item from the drop-down menu on the right of the page and in this example in Fig. 5 each phase of the test run records a separate set of metrics. Digesting time-series metrics into a single number is impossible to do manually, so we also provide summary metrics that list throughput, mean, median, and percentiles for the test run though these metrics are missing from Fig. 5 above due to lack of space.

Performance reports are globally readable for all logged in Fallout users and we have used this feature to share test runs across teams that were collaborating on investigating performance issues — having a single location to refer to for a test run's performance helped everyone to agree what work needed to be done next.

Individual performance reports can be grouped together into one report which allows users to look for differences in performance between test runs. Fig. 6 shows an example of a grouped performance report.

Graphed metrics for each run are displayed in the group report using different colors and details of the runs are included below the chart in a key which is not included in Fig. 6 again due to lack of space. The group performance reports are particularly useful for comparing different versions of Apache Cassandra or different configuration options on either the server or client side. When performance reports started appearing in Jira tickets to illustrate performance improvements and regressions, we knew that this feature had become successful as a way of quickly visualizing the performance of benchmarks. Over time, these links to performance reports have become even more useful as engineers have been able to refer back to previous benchmarking with ready-to-run tests they can reuse to troubleshoot new issues.

5.2. History server

Though performance reports offer a helpful way to look at the performance of a small number of test runs for comparison, the fact that all of the metrics from a test run are presented in a time-series chart makes it unsuitable for analyzing historical trends. When we need to understand how the performance of our automated tests have changed over the past few days or weeks we use a central Grafana server we call the history server. This server aggregates OS and application metrics from both clients and servers for historical analysis and is one of the ways that release engineers assess the quality of DataStax products. Aggregated metrics are very coarse grained to reduce disk space usage and calculate simple summary statistics — each metric is reduced to a single data point per run regardless of the duration of the test run.

Given that the history server is a central component of quality engineering for releases, it may be surprising that the hardware resources used to run it are extremely modest. The original version of the history server ran on a virtual machine with 1 CPU, 4 GB of RAM and a 20 GB hard disk drive. The current configuration uses 2 CPUs, 4 GB of RAM and an 80 GB hard disk drive. We believe that the reason the history server has survived for many years without any kind of downtime and without exhausting its small disk space is due to the aggressive graphite retention policy we apply to all metrics. The default metric namespace, *temporary*, has a retention policy of 1 h:15d which works well for one-off investigations because metrics can be updated once per hour and are automatically deleted after 15 days. We use a separate namespace, *performance_regressions*, to retain metrics for much longer but with a reduced frequency: daily metrics are recorded at most once a day, weekly metrics are recorded once a week, and both are retained for 10 years. Graphite's design requires that disk space for all configured metrics be allocated up front and storage for a single metric is 12 bytes, so we can calculate that storing one metric in *performance_regressions* every day for a full year only consumes 4.3 KB of disk space.

Fig. 7 shows one of the Grafana dashboards from the history server which includes panels for throughput, error count, and percentile metrics.

6. Lessons learned

Fallout has evolved over many years of development and we have found that while some of our initial design choices were correct and have stood the test of time, others were wrong and needed reassessing. And some problems we never even anticipated.

6.1. Configuration files should be short and expressive

The more lines a test configuration file has the greater the chance of introducing a bug. One of the goals for Fallout has been to provide enough support in the test and benchmark modules that common use cases only need small test run configuration files which reduces the probability that a user will make a mistake. This is still an on-going effort as it takes time for common usages to emerge when support for new modules is added but the end result is happier users with greater confidence in Fallout. This goal has served us well in creating a useful configuration language that is easy to understand.

6.2. Templating for configuration files encourages reuse

As Fallout amassed more users and the number of test run configurations increased, we noticed that many users began copying and pasting YAML across config files. A common situation where this happens is when users need to run the same test across multiple versions of an app, e.g. running the same benchmark against Apache Cassandra 3.11 and 4.0 to compare performance. We added support in Fallout's YAML parser for mustache [11] templates which allow users to use templates in their YAML files and provide specific values either on the Fallout test run web page or as parameters via the REST API.

Even with mustache templating, we found that users wanted to separate out common chunks of YAML into different, smaller files and include them in multiple configuration files. Additionally, users wanted to be able to store these files in a version control system. Fallout does not support either of these features so the haxx project was created which uses Jinja [12] templating to allow composition of test fragments and to provide version control via a git repository.

6.3. Tests need access to external files

A feature that we failed to anticipate early on was that tests, benchmarks, and tools would need the ability to access external files, e.g. configuration files. We initially worked around this limitation by either extending the test module to fetch the external file from a GitHub gist or by generating the test config file at runtime based on the keys and values in the Fallout YAML config. This approach did not scale as we added new modules and it is now possible to use a unified method to access external files with the ≪*file:filename*≫ syntax regardless of the module used in the test run config.

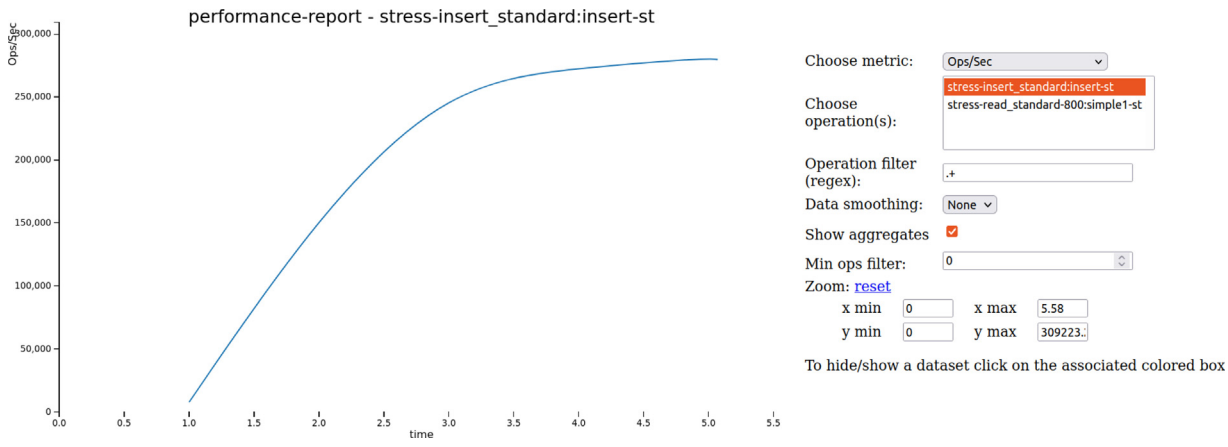


Fig. 5. Example performance report.

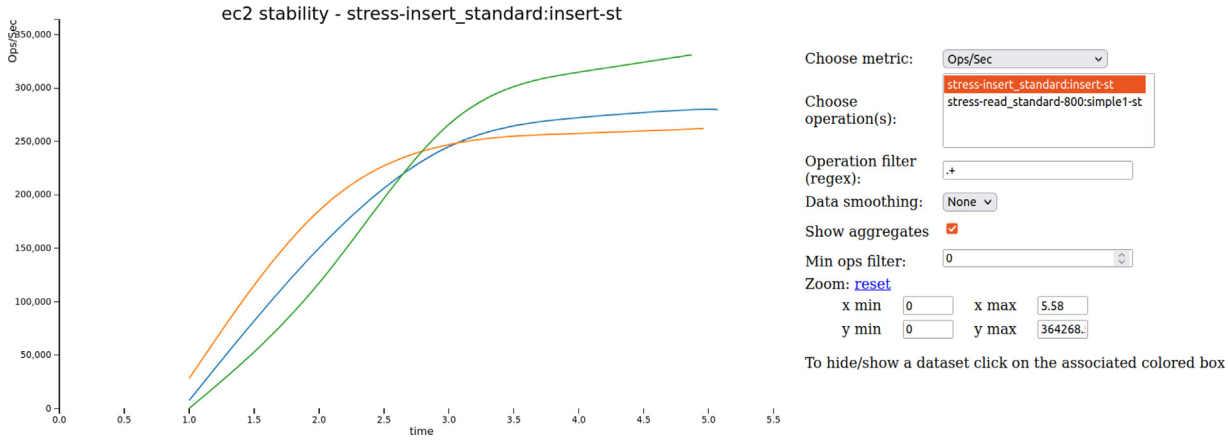


Fig. 6. Example grouped performance report.

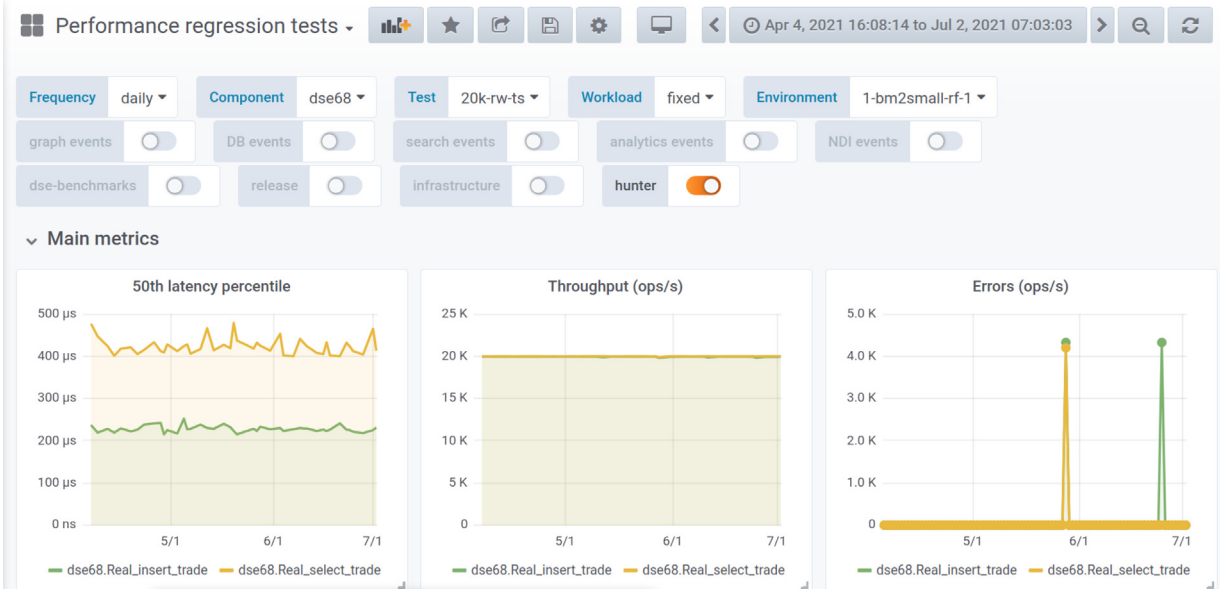


Fig. 7. Grafana dashboard.

6.4. *Long-running tests benefit from semantic checks and idempotency*

It is very straightforward to check YAML files for syntactic errors and there are numerous Java libraries available to do that, such as SnakeYaml [13] which is the library that Fallout uses. However, syntactic errors are only one source of problems afflicting users. Since most of the YAML values in a test config are consumed by tools other than Fallout, it is challenging to validate that the semantics of those values

behave as expected. We have encountered situations where a single mistyped character in a NoSQL table name caused all subsequent test phases to fail and was only triggered after the test had been running for an hour.

Additionally, re-running Fallout tests sometimes requires the infrastructure to be torn down and brought back up if Fallout cannot determine the runlevel of the cluster. Other deployment tools, such as Terraform [14] solve this problem with idempotency which allows the same deployment steps to be applied repeatedly without causing any changes to the underlying machine if the corresponding configuration for those steps has not changed. Fallout does make an attempt to detect the current cluster runlevel and skip unnecessary configuration steps but the detection is imperfect. This detection is used in Fallout's cluster-reuse mechanism, which is triggered by naming a cluster and requesting that it be left in a specific runlevel at the end of a test run; subsequent test runs with the same test definition will find the named cluster, detect its runlevel, and continue from there. This makes it possible to iterate on test creation a little bit faster, and—in some specialized cases – skip slow data loading steps for big-data tests. However, in our experience most users do not encounter situations where they need to use these features.

7. Related work

Automated testing, which includes running benchmarks, is a vital part of ensuring quality for software projects [15]. Integrating benchmarking into a continuous deployment pipeline is discussed in [16] which focuses on using performance metrics with thresholds to decide whether changes should be allowed into production. Since we use Fallout to test software that will ultimately be deployed to a variety of environments, ranging from the cloud to on-premise, there is no built-in functionality for gating deployments based on performance change thresholds. Instead, statistically significant changes are detected using change point detection and a developer is required to make the deployment decision. Automating deployments with Fallout is one of our future goals.

MockFog 2.0 [17] enables fog applications experiments by emulating fog infrastructure in the cloud and has a very similar design to Fallout. Both MockFog and Fallout provision infrastructure, configure and deploy applications, run tests and benchmarks, and even use states (Action states and NodeGroup states, respectively) to define legal transitions for the internal state machine. However, MockFog uses Docker to manage applications whereas Fallout supports both native and Kubernetes-based applications which more closely aligns with typical deployments of Apache Cassandra and Apache Pulsar. MockFog also uses Ansible to configure infrastructure which provides the idempotent state updates that are partly missing from Fallout's implementation.

Adelphi [18] is an open-source QA tool that runs on top of Kubernetes and allows users to run data integrity and performance tests against Apache Cassandra. It is packaged as a helm chart and includes a limited number of benchmarks and testing tools so that users can compare two clusters against one another. Adelphi takes care of executing the tests but does not provide facilities to create and terminate the underlying Kubernetes clusters or present the benchmark and test results for analysis.

MongoDB's Distributed Systems Infrastructure (DSI) [19] was developed at approximately the same time as Fallout though the two projects were not known to each other. DSI shares many things in common with Fallout including components to provision virtual machines, configure database servers and benchmarks, collect results for automated and visual inspection, and finally teardown the infrastructure when the test completes. Both Fallout and DSI use YAML configuration files to control test runs. However, Fallout differs from DSI in a number of ways. Fallout is written in Java and DSI is written in Python. While DSI primarily targets Amazon EC2, Fallout can currently launch tests on Google Cloud Platform, Amazon EC2, Microsoft Azure, as well as our internal OpenStack-based private cloud. Because ctool already existed when Fallout was created, Fallout has a very modular architecture and relies on other tools and components to do certain tasks whereas most of the corresponding functionality for DSI is built into the service. Lastly, as far as the authors are aware, DSI does not expose an API for other tools to call.

Work on reducing the cost of testing very large distributed systems by running many virtual machines on top of fewer physical servers is discussed in [20]. This work targets network services with thousands of nodes which are much larger than typical Apache Cassandra or Pulsar clusters.

RocksDB includes tools for running benchmarks and analyzing the results but no project exists to handle the setting up and tearing down of hardware to run the benchmarks [21]. Likewise, SAP has published work that shows how they integrate performance testing into their CI process [22] but no details are included on the way that tests are deployed on their testing infrastructure.

8. Conclusion

Fallout is a distributed systems testing service capable of automatically provisioning clients and servers, installing, configuring and executing distributed apps and workloads, and centrally collecting results for later analysis. We use Fallout internally at DataStax and it drives the entire performance and testing ecosystem for both our Apache Cassandra and Apache Pulsar products. Fallout started life with a very specific purpose and has evolved after years of engineering effort to be the backbone of performance and quality for us and it provides our engineering teams with fully-automated end-to-end testing for distributed systems. Fallout's REST API has been essential for new teams to leverage Fallout's distributed testing and has encouraged the birth of numerous tools and services that complement Fallout. Our Fallout server executes around 200 tests every day, and on busy days runs closer to 400 tests.

Since each of our engineering teams have their own preferences for the kinds of benchmarks, cluster configurations, and cloud infrastructure, all of these components are configurable in Fallout which has been designed with modularity in mind. We have extended this modularity to allow tests and benchmarks to load external files and added templating so that users can reuse test config fragments without copying and pasting.

We have released Fallout as an open-source project with the hope that the open-source community can benefit from our investment and the lessons we have learned running Fallout in production for over 5 years.

Acknowledgments

We would like to thank the anonymous reviewers for their valuable feedback and suggestions. Fallout was created by Jake Luciani, Joel Knighton, and Philip Thompson and we are thankful for their decision to create a new tool to solve the complex problem that is distributed systems testing. The history server was created by Pierre Laporte and it is stability is illustrated by the fact that it has been the component that has required the fewest updates in the whole Fallout ecosystem. Christopher Lambert was largely responsible for integrating ctool support to Fallout and James Trevino continues to maintain and improve Fallout. Ulises Cerviño Beresi created haxx. Shaunak Das contributed numerous test modules. Many more people have contributed to Fallout and related testing services and we are grateful for all of their efforts.

The open-source version of Fallout owes a great deal of gratitude to Jake Luciani and Jonathan Ellis for championing the project internally at DataStax.

References

[1] B. Boehm, Software engineering, IEEE Trans. Comput. C-25 (1976) 1226–1241.

[2] J. Humble, D. Farley, Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation, Addison-Wesley, 2010.

[3] Apache Cassandra, Cassandra stress, 2021, URL https://cassandra.apache.org/doc/latest/cassandra/tools/cassandra_stress.html, Accessed: 2021-08-05.

[4] K. Kingsbury, Distributed systems safety research, jepsen, 2021, URL https://jepsen.io/, Accessed: 2021-07-29.

[5] Google Cloud SDK documentation, Gcloud tool overview, 2021, URL https://cloud.google.com/sdk/gcloud, Accessed: 2021-10-07.

[6] D. Daly, W. Brown, H. Ingo, J. O'Leary, D. Bradford, The use of change point detection to identify software performance regressions in a continuous integration system, in: Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering(ICPE '20), 2020, http://dx.doi.org/10.1145/3358960.3375791.

[7] HdrHistogram, High dynamic range histogram, 2021, URL http://hdrhistogram.org/, Accessed: 2021-08-05.

[8] NoSQLBench, The open source nosql benchmarking suite, 2021, URL https://github.com/nosqlbench/nosqlbench, Accessed: 2021-08-05.

[9] Chaos Mesh, A powerful chaos engineering platform for kubernetes, 2021, URL https://chaos-mesh.org/ Accessed: 2021-10-07.

[10] pytest, Pytest: helps you write better programs, 2021, URL https://docs.pytest.org/en/6.2.x/, Accessed: 2021-10-07.

[11] Mustache, Logic-less templates, 2021, URL https://mustache.github.io/, Accessed: 2021-07-28.

[12] Jinja, Template engine for python, 2021, URL https://palletsprojects.com/p/jinja/, Accessed: 2021-08-04.

[13] snakeyaml, Yaml 1.1 parser and emitter for java, 2021, URL https://bitbucket.org/asomov/snakeyaml/src/master/, Accessed: 2021-08-05.

[14] Y. Brikman, Terraform: Up & Running: Writing Infrastructure As Code, O'Reilly Media, 2019.

[15] J. Waller, N.C. Ehmke, W. Hasselbring, Including Performance Benchmarks into Continuous Integration to Enable DevOps, Vol. 40 (2) (2015) 1–4, http://dx.doi.org/10.1145/2735399.2735416.

[16] M. Grambow, F. Lehmann, D. Bermbach, Continuous benchmarking: Using system benchmarking in build pipelines, in: 2019 IEEE International Conference on Cloud Engineering (IC2E), 2019, pp. 241–246, http://dx.doi.org/10.1109/IC2E.2019.00039.

[17] J. Hasenburg, M. Grambow, D. Bermbach, Mockfog 2.0: Automated execution of fog application experiments in the cloud, IEEE Trans. Cloud Comput. (2021) 1, http://dx.doi.org/10.1109/tcc.2021.3074988.

[18] Adelphi, Automation tool for testing cassandra OSS, 2021, URL https://github.com/datastax/adelphi, Accessed: 2021-08-05.

[19] H. Ingo, D. Daly, Automated system performance testing at mongodb, in: Workshop on Testing Database Systems (DBTest'20), 2020, http://dx.doi.org/10.1145/3395032.3395323.

[20] D. Gupta, K.V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, G.M. Voelker, Diecast: Testing distributed systems with an accurate scale model, ACM Trans. Comput. Syst. 29 (2) (2011) 1–48.

[21] Z. Cao, S. Dong, S. Vemuri, D.H. Du, Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook, in: 18th USENIX Conference on File and Storage Technologies (FAST 20), 2020, pp. 209–223.

[22] K.-T. Rehmann, C. Seo, D. Hwang, B.T. Truong, A. Böhm, D.H. Lee, Performance monitoring in SAP HANA's continuous integration process, in: ACM SIGMETRICS Performance Evaluation Review, Vol. 43, 2016, pp. 43–52, http://dx.doi.org/10.1145/2897356.2897362.

BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100011

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Revisiting the effects of the Spectre and Meltdown patches using the top-down microarchitectural method and purchasing power parity theory

Yectli A. Huerta [a,c,d,*], David J. Lilja [a,b,d]

[a] Scientific Computation
[b] Electrical and Computer Engineering Department
[c] Minnesota Supercomputing Institute
[d] University of Minnesota, USA

ARTICLE INFO

Keywords:
Bottleneck analysis
System characterization
Performance measurement

ABSTRACT

Software patches are made available to fix security vulnerabilities, enhance performance, and usability. Previous works focused on measuring the performance effect of patches on benchmark runtimes. In this study, we used the Top-Down microarchitecture analysis method to understand how pipeline bottlenecks were affected by the application of the Spectre and Meltdown security patches. Bottleneck analysis makes it possible to better understand how different hardware resources are being utilized, highlighting portions of the pipeline where possible improvements could be achieved. We complement the Top-Down analysis technique with the use a normalization technique from the field of economics, purchasing power parity (PPP), to better understand the relative difference between patched and unpatched runs. In this study, we showed that security patches had an effect that was reflected on the corresponding Top-Down metrics. We showed that recent compilers are not as negatively affected as previously reported. Out of the 14 benchmarks that make up the SPEC OMP2012 suite, three had noticeable slowdowns when the patches were applied. We also found that Top-Down metrics had large relative differences when the security patches were applied, differences that standard techniques based in absolute, non-normalized, metrics failed to highlight.

1. Introduction

Operating systems are complex computer programs that are continuously evolving to accommodate changes and updates to the underlying hardware it runs on. Like any other piece of software, frequent updates are released to address security issues, improve usability, enhance performance, and fix software bugs. These fixes have the potential of affecting performance, and it is essential to gain an understanding on the effect software patches have on a system. It is through the use of well known performance metrics that a proper assessment of security patches can be made by quantifying their effect, not only on overall performance, but on the different subsystems that make up a CPU.

In January 2008, two major vulnerabilities were reported, Spectre and Meltdown [1,2]. These vulnerabilities made it possible for attackers to gain access to data, stored in memory or caches, by bypassing security mechanisms. The exploits took advantage of CPU features that make it possible to use speculative execution to increase CPU performance. It was fear that the security fixes would have a major detrimental effect on performance by possible curtailing the speculation capabilities of CPUs.

A number of studies on the effects of the Spectre and Meltdown security patches had on performance were published. In one study, a number of Cray supercomputers were used to analyse the effects patches had on runtime performance. A number of benchmarks were tested, and it was found that the overall impact of the security patches was minimal [3]. Another study, showed the effects different patches had on two computational intensive workflows, pMatlab and Keras with TensorFlow, on a Intel based cluster [4]. It reported that significant negative effects, up to 21% for pMatlab and 16% for TensorFlow, once the CPU microcode update was applied.

To quantify the effect a change on the configuration or code had on performance, performance metrics such as the ones derived from the Top-Down bottleneck analysis are used [5]. This approach, the comparison of metrics after a change, is called differential analysis, and it makes it possible to associate specific changes on the system or code with changes on performance metrics [6]. A problem can arise when absolute rates are compared. The issue is that the comparison might provide an incomplete picture of changes between rates. Relative changes, normalized with the purchasing power parity technique [7], can provide additional information on the metric drift. This technique has been used to account for differences across GCC compiler suite releases [8] using the Top-Down bottleneck classification method. PPP

* Corresponding author at: Scientific Computation .
E-mail addresses: yhuerta@umn.edu (Y.A. Huerta), lilja@umn.edu (D.J. Lilja).

https://doi.org/10.1016/j.tbench.2021.100011
Received 6 August 2021; Received in revised form 11 October 2021; Accepted 20 October 2021
Available online 4 November 2021

made it possible to identify significant relative variations in different Top-Down categories. The Top-Down classification, in conjunction with PPP normalization, has been similarly applied to the AArch64 architecture [9], where it was used to analyse strong scaling and its resulting bottlenecks.

In this study, a comprehensive Top-Down and PPP analyses were made to quantify absolute and relative bottleneck metric changes, bottleneck drift, of a system when the Meltdown and Spectre security patches were enabled. Our study makes the following contributions:

- We found little difference in all but one of the benchmarks between patch settings.
- We showed that bottleneck profiles can differ even when security patches had little effect on performance.
- We showed that relative rates can vary significantly, while absolute bottleneck can remain relatively similar.
- We highlighted trends and differences in metrics that might have otherwise gone unnoticed by standard evaluation practices.

Performance analysis and system characterizations is a time consuming and complex process. Checking the impact of security patches requires multiple testing of different programs. Our approach makes it possible to compare bottleneck metrics by quantifying their absolute and relative changes when patches are applied. This makes it possible to obtain a more complete picture of the effect security patches had on systems.

2. Background

2.1. Spectre and Meltdown vulnerabilities

The Meltdown vulnerability allows an attacker to gain read access to all memory, even when lacking the appropriate privileges to do so [2]. The Spectre exploit allows attackers to gain access to private information through branch mispredictions [1]. For this study, we focused on the effect the patches had on pipeline bottlenecks. The following variant security patches were provided the OS vendor and applied to the system: [11,12]:

- *Spectre, variant 1:* This is a kernel patch fix that is always enabled. It provides bounds checking during branching to prevent arbitrary bypassing.
- *Spectre, variant 2:* This fix includes microcode and kernel patches. It can be disabled to prevent performance impacts. It prevents data leakage through indirect branch poisoning.
- *Meltdown, variant 3:* This is a kernel fix. It can be disabled to prevent performance impacts. It prevents an attacker from reading memory through speculative cache loading.

In the following subsections, we discuss the methods used to analyse the performance impact of the security patches for Spectre, variant 2, and Meltdown, variant 3, had on the system.

2.2. Top-Down classification method

The Top-Down analysis method is a bottleneck classification technique that identifies dominant bottlenecks of an application. This method tracks CPU pipeline slots — resources needed to process a micro-operation (uop). Uops are low level hardware operations of microarchitectural instructions which were generated to represent the application being executed by the CPU. Pipeline slots are assigned into four main categories: Frontend Bound, Backend Bound, Retiring and Bad Speculation [5]. A simple classification is applied to pipeline slots to assign the bottleneck to the right category. If a slot was allocated, it will be classified as Retiring if the slot is eventually retired. It will be assigned to the Bad Speculation category if it is not retired. If the slot cannot be allocated, it will be assigned to the Backend Bound category if it is a back end stall. Otherwise, it will be assigned to the Frontend Bound category. Back end stalls occur when there are not enough resources in the back end portion of the pipeline to handle new slots. Front end stalls take place when the front end cannot supply slots to the back end portion of the pipeline. Non stalled slots are classified as Bad Speculation, when a slot will never retire due to an incorrect speculation, or slots were blocked by the pipeline due to recovery operations due to an earlier bad speculation. Retired slots are the slots that successfully completed their operations.

To apply the Top-Down analysis technique, a user would first compute the main category metrics to identify which classification has the highest bottleneck rate. Once a category is identified, the user can narrow down the metrics needed to analyse by just focusing in the subcategories of the selected main category. The user can continue generating metrics until the source of the problem is identified. Since our goal is to provide a comprehensive view of the different components that make up the processor, our experimental runs included multiple categories. This made it possible to get a more complete picture of bottlenecks across the processor, and a better understanding of how the different processor components were affected by the use of security patches. Table 1 lists the main Top-Down categories and subcategories that were used in this paper, along the corresponding formulas needed to compute the metrics. More in-depth descriptions, definitions and techniques of the Top-Down metrics, and how to use the Top-Down analysis method, were made available by the CPU vendor [13].

2.3. Purchasing power parity

Purchasing power parity theory underlies different methods to compare the cost of identical products such as lattes, and iPods between different countries, each of them with different currencies [7,14,15]. The most famous PPP index is the Big Mac Index (BMI), which was developed by The Economist magazine [16]. The goal of the BMI is to compare the strength of the currency by testing how much of the same product a currency can buy when compared to another currency. A currency is overvalued – when the product bought using that currency is more expensive – or undervalued – when the product is cheaper – when compared to a base currency.

The following is an illustrative example of the purchasing power of the Chinese yuan versus the US dollar as described in The Economist magazine. For this example, the dollar to yuan exchange rate is $\$1 = 6.4$ yuan. The quoted Big Mac price was \$5 and 20 yuan. Eq. (1) computes the Big Mac exchange rate which is based on its local price.

$$20/5 = 4 \tag{1}$$

Eq. (1) shows that on the basis of Big Mac burger prices, the exchange rate should be set at 4 yuans per dollar. Since the actual exchange rate is 6.4 yuans per dollar, Eq. (2) shows that the yuan is 37.5% undervalued as compared to the US dollar.

$$(4 - 6.4) * 100/6.4 = -37.5 \tag{2}$$

PPP theory can be used to determine the relative difference between bottlenecks generated by the same benchmark but generated under different system configurations. The *currency* used to compare the cost is the number of cycles it took to run the program to completion. The *product* being compared are the Top-Down metrics for each benchmark. The goal is to show that a metric value can differ, or be similar to another, as defined by the Top-Down formulas, while its true cost might be relatively higher or lower, when compared to a baseline run. PPP normalized rates close to 0% imply parity between the patches disabled and enabled metrics. It takes about the same number of CPU_clk cycles for a similar number of pipeline slots to achieve similar Top-Down metric rates. For positive PPP rates, it implies that the patches enabled CPU_clk cycles are overvalued. It requires less cycles to achieve same metric magnitude when compared to a configuration with the security patches disabled. Negative PPP rates imply that the CPU_clk cycles

Table 1
Top-Down Metric formulas for Intel Skylake processor [5,10].

Metric	Formula
CORE_CLKS	CPU_CLK_UNHALTED.THREAD_ANY/2
CLKS	CPU_CLK_UNHALTED.THREAD
SLOTS	4 * CORE_CLKS
Frontend Bound	
Frontend Bound	IDQ_UOPS_NOT_DELIVERED.CORE/SLOTS
DSB	(IDQ.ALL_DSB_CYCLES_ANY_UOPS - IDQ.ALL_DSB_CYCLES_4_UOPS) /CORE_CLKS
Branch Resteers	(INT_MISC.CLEAR_RESTEER_CYCLES+BACLEARS.ANY)/CLKS
Bad Speculation	
Bad Speculation	(UOPS_ISSUED.ANY - UOPS_RETIRED.RETIRE_SLOTS + (4*Recovery_Cycles))/SLOTS
Recovery_Cycles	INT_MISC.RECOVERY_CYCLES_ANY/2
Branch Mispredicts	(BR_MISP_RETIRED.ALL_BRANCHES/(BR_MISP_RETIRED.ALL_BRANCHES + MACHINE_CLEARS.COUNT)) * Bad Speculation
Machine Clears	Bad Speculation - Branch Mispredicts
Retiring	
Retiring	UOPS_RETIRED.RETIRE_SLOTS/SLOTS
Microcode Sequencer	((UOPS_RETIRED.RETIRE_SLOTS /UOPS_ISSUED.ANY)* IDQ.MS_UOPS) /SLOTS
Base	Retiring - Microcode Sequencer
Backend Bound	
Backend Bound	1 - (Frontend Bound + Bad Speculation + Retiring)
Backend Bound, Memory Bound	
Store Bound	EXE_ACTIVITY.BOUND_ON_STORES/CLKS
L2_Bound_Ratio	(CYCLE_ACTIVITY.STALLS_L1D_MISS-CYCLE_ACTIVITY.STALLS_L2_MISS) / CLKS
LOAD_L2_HIT	MEM_LOAD_RETIRED.L2_HIT* (1+MEM_LOAD_RETIRED.FB_HIT/MEM_LOAD_RETIRED.L1_MISS)
L1 Bound	(CYCLE_ACTIVITY.STALLS_MEM_ANY-CYCLE_ACTIVITY.STALLS_L1D_MISS) /CLKS
L2 Bound	(LOAD_L2_HIT/(LOAD_L2_HIT + L1D_PEND_MISS.FB_FULL)) * L2_Bound_Ratio
L3 Bound	(CYCLE_ACTIVITY.STALLS_L2_MISS-CYCLE_ACTIVITY.STALLS_L3_MISS) / CLKS
DRAM Bound	(CYCLE_ACTIVITY.STALLS_L3_MISS/CLKS) + L2_Bound_Ratio - L2_Bound
Backend Bound, Core Bound	
Divider	ARITH.DIVIDER_ACTIVE /CLKS
UPC	UOPS_RETIRED.RETIRE_SLOTS/CLKS
Few_Uops_Executed _Threshold	EXE_ACTIVITY.2_PORTS_UTIL * UPC/5
Core_Bound_Cycles	EXE_ACTIVITY.EXE_BOUND_0_PORTS + EXE_ACTIVITY.1_PORTS_UTIL + Few_Uops_Executed_Threshold
Ports Utilization	**if** ARITH.DIVIDER_ACTIVE < EXE_ACTIVITY.EXE_BOUND_0_PORTS **then** Ports Utilization = Core_Bound_Cycles/CLKS **else** Ports Utilization = (Core_Bound_Cycles - EXE_ACTIVITY.EXE_BOUND_0_PORTS)/CLKS

for a configuration with patches enabled are undervalued. It requires more cycles to achieve the same metric value when compared to a configuration with security patches disabled.

Top-Down metrics were computed using the formulas described in Table 1. The use of *CPU_CLK_UNHALTED. THREAD*, or *CPU_CLK_UNHALTED.THREAD_ANY* to compute the PPP Exchange Rate, Eq. (3), was based on which PMU event the Top-Down metric formula used in its computation. Some metrics use *CLKS* while others use the *CORE_CLKS* performance metric. The baseline CPU_clk values used for comparison were obtained from runs with the security patches disabled.

$$PPP\_Exchange\_Rate = CPU\_clk/CPU\_clk_{baseline} \tag{3}$$

Eq. (4) computes the PPP index. The baseline, represented by the variable $Metric_{baseline}$, was the resulting Top-Down metric value with the security patches disabled.

$$PPP = 100 * ((Metric/Metric_{baseline}) - PPP\_Exchange\_Rate)/PPP\_Exchange\_Rate \tag{4}$$

The following is an example of how to compute the drift in terms of relative difference of the Retiring metric for the *370.mgrid331* benchmark. The Retiring metric was computed using the formulas provided by the Top-Down method as shown in Table 1, and the results were found to be 0.06789046 when patches were enabled, and 0.09773369 when patches were disabled. The PPP exchange rate was computed using the PMU event, *CPU_CLK_UNHALTED. THREAD_ANY* with a resulting value of 1.46, Eq. (5). The drift between security patch settings was found to be −52.42%, Eq. (6). When patches were enabled, the *CPU_CLK_UNHALTED.THREAD_ANY* were overvalued. The system used more *CPU_CLK_UNHALTED.THREAD_ANY* cycles to retire a similar number of uops when patches were disabled.

$$81671652350125/55924600475776 = 1.46 \tag{5}$$

$$100 * ((0.06789046/0.09773369) - 1.46)/1.46 = -52.42 \tag{6}$$

In this paper, we use the relative change between metrics, the difference in Top-Down metrics between patch settings divided by the metric value when patches were disabled, as an additional indicator of the changes between patch settings. For the example just described, *370.mgrid331* had a relative change in its Retiring metric of −30.54%. The relative change and PPP normalized rates give us a sense of the relative change of the Top-Down metric between patch settings, not the change of its effect on the system. Additionally, PPP normalized rates give us information on the relative change when taking into account the number of core cycles that were used to compute the metrics, which is useful when putting large percentage values in relative changes in perspective.

Table 2
SPEC OMP2012 Benchmark description [17].

Benchmark name	Programming language	Description
350.md	Fortran	Physics: Molecular dynamics
351.bwaves	Fortran	Physics: Computational Fluid Dynamics (CFD)
352.nab	C	Molecular Modelling
357.bt331	Fortran	Physics: Computational Fluid Dynamics (CFD)
358.botsalgn	C	Protein Alignment
359.botsspar	C	Sparse LU
360.ilbdc	Fortran	Lattice Boltzmann
362.fma3d	Fortran	Mechanical Response Simulation
363.swim	Fortran	Weather Prediction
367.imagick	C	Image Processing
370.mgrid331	Fortran	Physics: Computational Fluid Dynamics (CFD)
371.applu331	Fortran	Physics: Computational Fluid Dynamics (CFD)
372.smithwa	C	Optimal Pattern Matching
376.kdtree	C++	Sorting and Searching

3. Experimental setup

We used the Top-Down method in combination with the SPEC OMP2012 benchmarks [17,18] to measure the effects of Spectre and Meltdown patches have on the Intel 2021.1 compiler suite. The following compiler options were used as the default to compile most benchmarks: *-fopenmp -O3 -march=skylake-avx512 -g -pg*. Some of the benchmarks required additional or different options. The *371.applu331* benchmark used *-fopenmp -O2 -march=skylake-avx512 -g -pg*. *367.imagick* required the compiler option *-std=c99* to be added to the default options. Additionally, the option *-FR* was added to *350.md*, while *-mcmodel=medium* needed to be added to the *363.swim* and *357.bt331* benchmarks. The SPEC OMP2012 benchmarks are described in Table 2. OMP2012 results that followed the SPEC reporting guidelines can be submitted for publication [19]. A two socket Intel(R) Xeon(R) Silver 4110 CPU @ 2.10 GHz with 8 cores per socket, 2 threads per core was used running the CentOS 7.6.1810 Linux version installed. *perf record* collected the data from eight performance counters per experimental run. There were at least five data points per performance counter for both patch settings. To compute Top-Down metric rates, the average of the performance counter values was used.

It is possible to disable the Spectre variant 2 and Meltdown variant 3 through an interface made available by the Red Hat Linux vendor, which is also available to the CentOS distribution. The vendor also made available a script to check the state of the security patches, to see whether or not the system currently has its patches enabled or disabled [12]. In this study, version 3.1 of the verification script was used. To disable the security patches, a 0 was stored in the following files located in */sys/kernel/debug/x86/*: *ibrs_enabled*, *retp_enabled* and *pti_enabled*. Patches were enabled by replacing the 0 with a 1 in the same files.

4. Analysis of results

Fig. 1 shows the speedup gains or losses when the security patches were enabled. There were at least 55 runs for each benchmark for both patch settings, and the averages were taken to compute the speedups. The plot shows that most benchmarks suffer about a $0.01x$ speedup loss. The exception is *360.ilbdc*, which experienced a small gain of $0.02x$, and *370.mgrid331*, which had a negative effect close to $0.04x$. While these are not significant effects on runtime, we further analysed the effects of the security patches through the use of the Top-Down classification method to see how bottlenecks were affected on a subset of benchmarks. We showed that while benchmark runtimes were similar, their bottleneck profiles were different. With the use of PPP techniques, we were able to highlight and quantify these relative differences when compared to the baseline, a system with its security patches disabled.

Table 3 shows the performance counters of significance that were used to compute the Top-Down metrics. When the results followed a normal distribution, the unpaired two-sample t-test was used. When the results did not follow a normal distribution, the non parametric two-samples Wilcoxon rank test was used. We had a minimum of five runs per counter for both settings, patches enabled and disabled. P-values for the performance counters were identified as significant at values less than 0.05. For the Retiring category, there were more uops delivered by the Microcode Sequencer for *358.botsalgn*, *359.botsspar*, and *370.mgrid331* when patches were enabled. Additionally, *359.botsspar* had increases in number of uops issued by the resource allocation table while at the same time the number of retiring slots increased when patches were enabled. *358.botsalgn* had the opposite effect.

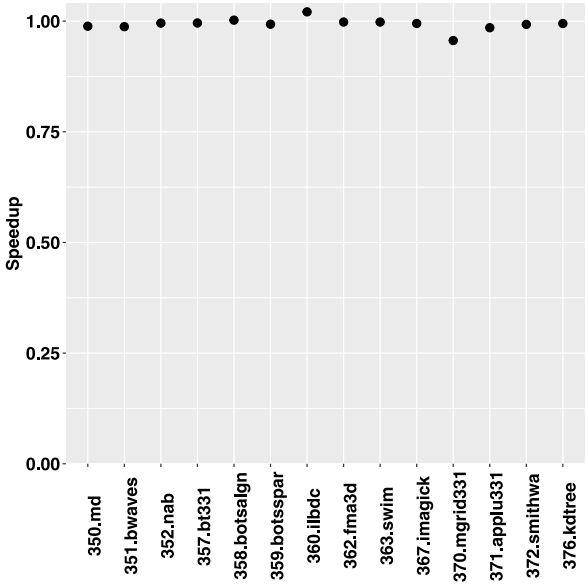


Fig. 1. Speedup comparison for the Intel 2021.1 compiler suite when patches are enabled using SPEC OMP2012 benchmarks with 32 threads and SMT enabled. Higher is better.

For the Frontend Bound metric, *358.botsalgn*, *359.botsspar*, *360.ilbdc* and *370.mgrid331* had increases in the number of uops not delivered to the resource allocation table per thread when the patches were enabled. A higher number in none delivered uops could potentially translate in the frontend under-supplying the CPU's backend portion of the pipeline. Similarly, the *358.botsalgn*, *359.botsspar*, *360.ilbdc*, *370.mgrid331* and *371.applul331* reported an increase in the number of times frontend resources are resteered when encountering branch instructions in a fetch line with patches enabled. *358.botsalgn* and *359.botsspar* had decreases in the number of uops and 4 uops cycles that were delivered to the instruction decode queue unit. These decreases occurred when patches were enabled.

Table 3
Performance counters of significance.

Performance counter	Benchmark	Category
BACLEARS.ANY	358.botsalgn, 359.botsspar, 360.ilbdc, 370.mgrid331, 371.applu331	Branch Resteers
BR_MISP_RETIRED.ALL_BRANCHES	358.botsalgn, 359.botsspar, 360.ilbdc, 370.mgrid331, 371.applu331	Branch Mispredicts
CPU_CLK_UNHALTED.THREAD	360.ilbdc	CLKS
CPU_CLK_UNHALTED.THREAD_ANY	359.botsspar, 360.ilbdc, 359.botsspar	CORE_CLKS
CYCLE_ACTIVITY.STALLS_L1D_MISS	359.botsspar	L1, L2 Bound
CYCLE_ACTIVITY.STALLS_L2_MISS	359.botsspar	L2, L3 Bound
CYCLE_ACTIVITY.STALLS_L3_MISS	359.botsspar, 370.mgrid331	L3, DRAM Bound
CYCLE_ACTIVITY.STALLS_MEM_ANY	358.botsalgn	L1 Bound
EXE_ACTIVITY.2_PORTS_UTIL	359.botsspar	Few_Uops_Executed_Threshold
EXE_ACTIVITY.EXE_BOUND_0_PORTS	358.botsalgn, 359.botsspar, 360.ilbdc	Ports Utilization
IDQ.ALL_DSB_CYCLES_4_UOPS	358.botsalgn, 359.botsspar	DSB
IDQ.ALL_DSB_CYCLES_ANY_UOPS	358.botsalgn, 359.botsspar	DSB
IDQ.MS_UOPS	358.botsalgn, 359.botsspar, 370.mgrid331	Microcode Sequencer
IDQ_UOPS_NOT_DELIVERED.CORE	358.botsalgn, 359.botsspar, 360.ilbdc, 370.mgrid331	Frontend
INT_MISC.CLEAR_RESTEER_CYCLES	358.botsalgn, 359.botsspar, 360.ilbdc, 370.mgrid331	Branch Resteers
INT_MISC.RECOVERY_CYCLES_ANY	358.botsalgn, 359.botsspar, 360.ilbdc, 370.mgrid331, 371.applu331	Recovery_Cycles
MACHINE_CLEARS.COUNT	358.botsalgn, 359.botsspar, 360.ilbdc, 370.mgrid331, 371.applu331	Branch Mispredicts
MEM_LOAD_RETIRED.FB_HIT	358.botsalgn, 359.botsspar	L2 Bound
MEM_LOAD_RETIRED.L1_MISS	358.botsalgn	L2 Bound
UOPS_ISSUED.ANY	358.botsalgn, 359.botsspar	Bad Speculation, Microcode Sequencer
UOPS_RETIRED.RETIRE_SLOTS	358.botsalgn, 359.botsspar	Bad Speculation, Retiring, Microcode Sequencer

In the Bad Speculation category, the following benchmarks had increases of statistical significance when patches were enabled. *358.botsalgn*, *359.botsspar*, *360.ilbdc*, *370.mgrid331* and *371.applu331* had increases in the number of events that require the clearing of the pipeline, the number of mispredicted retired instructions, and the number of stalls due to recoveries from earlier clear events increased for these benchmarks. In the core bound category, *358.botsalgn*, *359.botsspar* and *360.ilbdc* had statistically significant increases of cycles with where no uops were executed on all ports. *359.botsspar* had an increase in the number cycles in which 2 uops were executed on all ports. In the memory bound classification, the number of execution stalls due data misses increased for *359.botsspar* for L1D, *359.botsspar* for L2, *359.botsspar* and *370.mgrid331* for L3. Additionally, *359.botsspar* reported statistically significant increases for execution stalls due to memory subsystem outstanding loads.

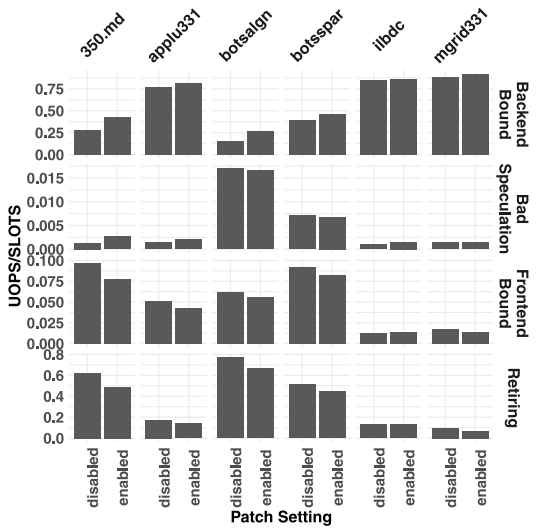
4.1. Top-Down metrics

Fig. 2(a) shows the effects the security patches had on the main Top-Down categories. With the exception of *360.ilbdc*, all Frontend Bound values dropped. This was driven by two factors when patches were enabled: the number of *CPU_CLK_UNHALTED.THREAD_ANY* increased for all benchmarks, and the number of uops not delivered to the resource allocation table when the backend portion of the pipeline was not stalled, the *IDQ_UOPS_NOT_DELIVERED.CORE* performance counter, stayed relatively the same. In the case of *360.ilbdc*, the opposite was true, the number of CPU clock cycles stayed relatively the same while the number of uops not delivered increased by more than 11%. This resulted in an increase of 7.5% in the Frontend Bound metric when the security patches were enabled. Fig. 2(b) shows the corresponding PPP normalized rates. Except for *360.ilbdc*, which had a PPP rate of 3.67%, all benchmarks had negative PPP rates that ranged from −22% for *358.botsalgn* and *359.botsspar*, to −49% for *370.mgrid*. As the number of cycles, CPU_CLK_UNHALTED.THREAD_ANY, increased, the number of uops not delivered increased modestly. Resulting in less not delivered uops per cycles, making the cycles undervalued. The runs with patches enabled handled relatively the same number of stalls with more core cycles.

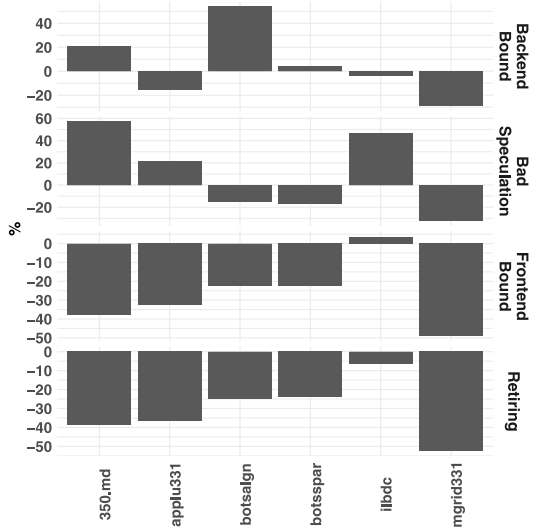
The Retiring metric values decreased for all benchmarks when the patches were applied. It had a drop of −30.53% for *370.mgrid331*, while *350.md* and *371.applu331* had drops of about −20%. *360.ilbdc* had a drop of −2.73%. This is attributed to the increase in core cycles, CPU_CLK_UNHALTED.THREAD_ANY, while the number of retired slots remained relatively the same when patches were enabled. PPP rates decreased for all benchmarks. Expect for *360.ilbdc*, all had at least a −24% drop, with *370.mgrid* recording a drop of −52%. As the number of core cycles increased with the security patches enabled, the number of retired slots remained relatively the same. *370.mgrid* had the largest increase in core cycles, 46%, while *360.ilbdc* had the smallest increase, 3.73%. This explains the difference in magnitude in PPP rates.

The Bad Speculation metric had an increase of 101% for *350.md*, 50% for *371.applu331*, and 51% for *360.ilbdc* when patches were enabled. This is due to an increase of *UOPS_ISSUE_ANY* and *Recovery_Cycles* while the number of *UOPS_RETIRED.RETIRED_SLOTS* stayed relatively the same. The other benchmarks, *359.botsspar*, *358.botsalgn* and *370. mgrid331*, had less than a −4% decrease in Bad Speculation rates. Since the number of clock cycles, the denominator in the formula, also increased but at a larger rate, the Bad Speculation rate decreased when patches were enabled. While regular rates showed a decrease of −4%, PPP normalized rates were larger in magnitude, 32.23% for *370.mgrid*, 16.63% for *359.botsspar* and 14.84% for *358.botsalgn*. The effects of large increases in core cycles, 46% for *370.mgrid331*, and 14% for *359.botsspar* and *358.botsalgn*, resulted in decreases of PPP rates. There were more core cycles to do the same amount of work once the patches were enabled, which resulted in a depreciation in the value of core cycles. The other benchmarks experienced gains in PPP normalized rates. *350.md* had a gain of 57.15%, *371.applu331* had a gain of 21.06%, and *360.ilbdc* had a gain of 46.42%. The *Recovery_Cycles* metric increased at a much larger rate for this subset of benchmarks than the benchmarks with negative PPP rates. *350.md* had an increase in *Recovery_Cycles* of 1334%, while *371.applu331* had an increase of 1446% and *360.ilbdc* had an increase of 983%. Positive PPP rates show that core cycles were overvalued, the same amount of work is being done with less core cycles relatively to baseline runs.

The *360.ilbdc* benchmark saw less than a 1% difference in the Backend Bound metric between patch settings. For *360.ilbdc*, the Bad Speculation, Front End, and Retiring metrics stayed relatively the same, resulting in a very similar Backend Bound rate. All other benchmarks had an increase in the Backend Bound rates because of lower Retiring rates when patches were enabled. *370.mgrid331* had a 4% increase, while *371.applu331* recorded an increase of 6%. Other benchmarks had large increases. *359.botsspar* had an increase of 19%, *350.md* had a 55% increase, and *botsalgn* had a 76% increase. PPP rates increased for benchmarks that had large increases in Backend Bound rates. For instance, *350.md* had a core cycles increase of 28% but its Backend Bound rate had a larger effect when it increased by 54%, resulting in PPP rate of 20%. *370.mgrid331* had a core rate increase of 46% and a Backend Bound rate increase of 3.90%, resulting in a PPP rate of

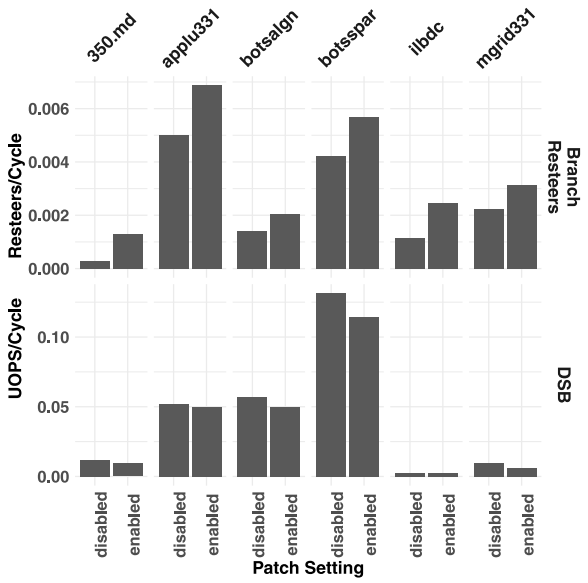


(a) Regular Rates

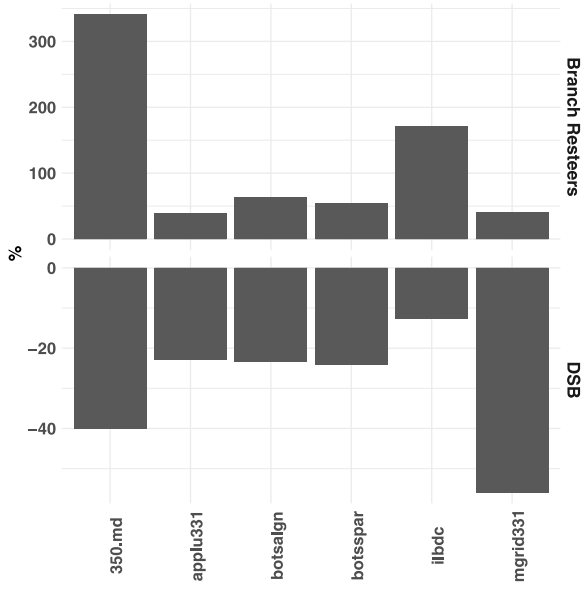


(b) PPP Normalized Rates

Fig. 2. Results of the Top-Down architectural bottleneck classification main categories.



(a) Regular Rates



(b) PPP Normalized Rates

Fig. 3. Frontend Bound subcategories.

−28%. When patches were enabled, the number of cycles increased for some benchmarks at higher rates than there were uops to be processed resulting in negative PPP rate, while others had proportionally fewer cycles for an increasing number of uops resulting in overvalued cycles and positive PPP rates.

The following subsections describe the effects of the security patches had on different Top-Down subcategories.

4.1.1. Frontend bound

Frontend Stalls track the fraction of slots that were affected when the frontend of the pipeline undersupplies the pipeline's backend. Two subcategories were examined in this paper: DSB and Branch Resteers. The DSB metric tracts the fraction of CPU cycles that were affected by the decoded uop cache, DSB, fetch pipeline. Branch Resteers account for the CPU stalls due to branch resteers, delays from a corrected path, after a mispredicted branch. Fig. 3(a) shows that the Branch Resteers metric increased when the security patches were enabled. This is due to increases in the number of cycles the issue stage had to wait to recover from bad speculation events, while at the same time, the number of CPU_CLK_UNHALTED.THREAD decreased or stayed the same. For example, *350.md* had a Branch Resteers rate increase of 329% due mostly in part to an increase of 234% in INT_MISC.CLEAR_RESTEER_CYCLES, and a slight decrease of −2.58% for CPU_CLK_UNHALTED.THREAD. *360.ilbdc* had a increase of 117.93% for Branch Resteers resulting from an increase of 61.81% in INT_MISC.CLEAR_RESTEER_CYCLES, and a decrease of −19.65% in CPU_CLK_UNHALTED.THREAD. Normalized PPP rates for the Branch Resteers metric were all positive, Fig. 3(b). While CPU core cycles increased, Branch Resteers increased at higher rates. The largest percentage increases in PPP rates reflect large increases in Branch Resteers while lower increasing rates for core cycles. That is the case for *350.md*, which had a PPP rate of 340.42%. Its core cycles rate increased by 28.16% while its Branch Resteers rate increased by 329.04%. Similarly, *360.ilbdc* had a PPP rate of 171.25%, because of an increase of 3.73% in core cycles and a 117.94% increase in Branch Resteers when patches were enabled. Core cycles were overvalued, when compared to their baseline, because fewer core cycles had to do relatively less work.

The DSB metric decreased when patches were enabled. While the number of uops that were delivered to the instruction decode queue remained the same or had a slight decrease, there were increases in CPU_CLK_UNHALTED.THREAD_ANY, the divisor. This resulted in more CPU cycles for the same number of delivered uops for all the benchmarks. For instance, *370.mgrid331* and *350.md* had increases of 46% and 28.16% for CPU_CLK_UNHALTED.THREAD_ANY, resulting in DSB decreases of −35.54% and −23.05% respectively while the uops delivered stayed relatively the same. DSB PPP rates were negative for all benchmarks. The number of core cycles increased while DSB rates decreased when patches were enabled. The benchmarks with the highest rates reflect the large increases in core clocks and decreases in the DSB rate. That is the case for *370.mgrid*. It had a PPP rate of −55.87%, an increase of 46.04% for its CPU_CLK_UNHALTED.THREAD_ANY value and a decrease of −35.55% for its DSB rate. The lowest PPP rate was reported by *360.ilbdc*. It had a small increase in core cycles, 3.73%, and a decrease in DSB rate of −9.29%.

4.1.2. Retiring

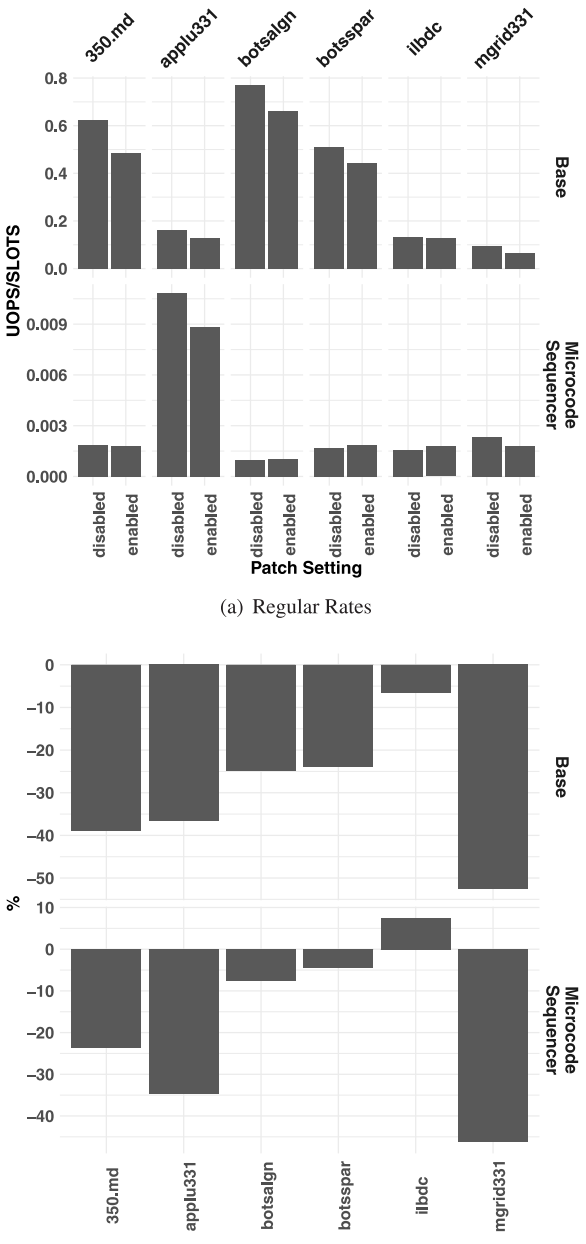
The Retiring category represents the fraction of pipeline slots of useful work, the uops that were eventually retired. The Microcode Sequencer metrics is a retiring subcategory that accounts for pipeline slots of uops that were retired and were fetched by the microcode sequencer ROM. The Base metric tracks retired uops that did not originate from the microcode sequencer. Fig. 4(a) shows that Base rates across all benchmarks decreased when patches were enabled. This was due to lower Retiring rates. PPP rates for the Base metric rates were negative, reflecting the increasing number of core cycles as the Base rates decreased, Fig. 4(b).

When the security patches were enabled, *370.mgrid331* and *371.applu331* had lower Microcode Sequencer rates, −21.54% and −18.56% respectively. This resulted from an increase in core cycles, CPU_CLK_UNHALTED.THREAD _ANY, the divisor in the metric formula, while the other performance counters remained the same. The other benchmarks had similar or slightly higher Microcode Sequencer rates because the number of uops delivered by the microcode sequencer, IDQ.MS_UOPS, increased at a similar or higher rate than CPU_CLK_UNHALTED.THREAD_ANY. *360.ilbdc* was the only benchmark with a positive PPP rate, 7.51%. This was due to an increase of 11.52% in the Microcode Sequence rate when patches were enabled while the number of core cycles increased modestly, 3.73%. All other benchmarks had negative PPP rates, up to −46.28% for *370.mgrid331* because of the large increase of core cycles and a decrease in the Microcode Sequencer rate when the security patches were enabled.

4.1.3. Backend bound

The Backend Bound metric measures the fraction of slots where no uops were delivered to the backend portion of the pipeline due to bottlenecks in the computational or memory subsystems. This metric is further divided into Memory and Core Bound subcategories. In this study, the following memory subsystem stalls due to load accesses were tracked through their corresponding Top-Down metrics: L1, L2, L3 and DRAM. Additionally, the Store Bound metric tracks stalls due to store memory accesses.

The numerator of the L1 Bound metric is the difference between the number of execution stalls due to outstanding loads in the memory subsystem, CYCLE_ACTIVITY.STALLS_MEM_ANY, minus the number of stalls due to outstanding L1 cache miss demand load, CYCLE_ACTIVITY.STALLS_L1D_MISS. When the security patches were applied, both type of stalls increased, Fig. 5(a). Some of the benchmarks had negative values because the CYCLE_ACTIVITY.STALLS_L1D_MISS values were larger in magnitude. This was the case for *359.botsspar*, *360.ilbdc* and *370.mgrid331*, which had a negative L1 Bound rate only when patches were enabled. *350.md*, *358.botsalgn* and *371.applu331* had all positive L1 Bound rates. *350.md* had an increase of 31.39%



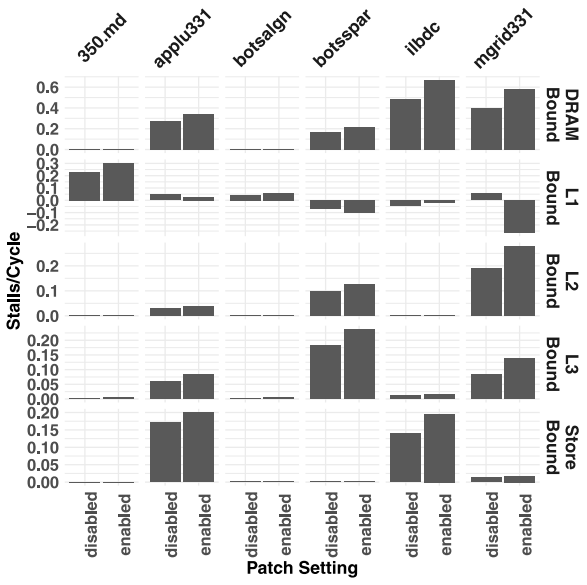
(a) Regular Rates

(b) PPP Normalized Rates

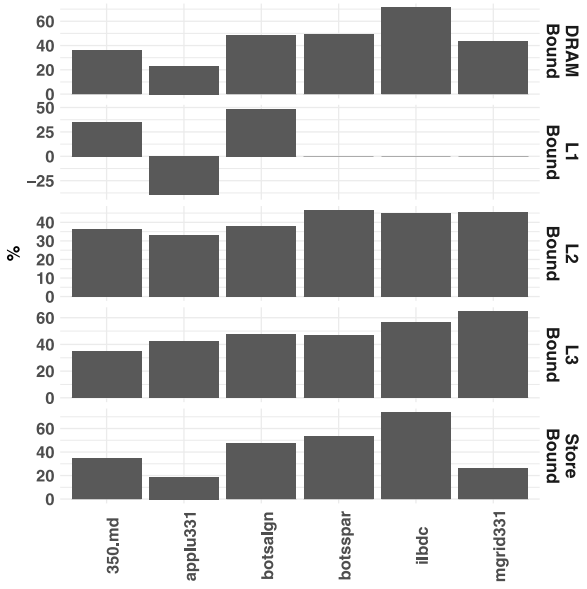
Fig. 4. Retiring subcategory.

and *358.botsalgn* an increase of 30.67% with patches enabled due to increases in stalls and a decrease in core cycles, CPU_CLK_UNHALTED.THREAD. *371.applu331* had a decrease in the L1 Bound rate of −40.83%. It had a small decrease in the core cycles, and a higher increase in stalls due to L1 cache miss activity, CYCLE_ACTIVITY.STALLS_L1D_MISS, than stalls due to the memory subsystem, CYCLE_ACTIVITY.STALLS_MEM_ANY.

L2 Bound rates were higher when the security patches were enabled. This was attributed to large increases in the L2_Bound_Ratio rates, higher execution stalls for L1 cache misses, CYCLE_ACTIVITY.STALLS_L1D_MISS, and a decrease in CPU_CLK_UNHALTED.THREAD. *370.mgrid 331* had an increase of 32.11% while *359.botsspar* and *371.applu331* had increases in the low 20s. L3 Bound rates increased with patches enabled. This was due mostly by increases in execution stalls for L2 cache misses, CYCLE_ACTIVITY.STALLS_L2_MISS, and a decrease in core cycles, CPU_CLK_UNHALTED.THREAD. *370.mgrid331* had an increase of



(a) Regular Rates



(b) PPP Normalized Rates

Fig. 5. Memory Bound subcategories which are part of the Backend Bound classification.

66.46%, while *371.applu331* had an increase of 40% and *359.botsspar* an increase of 29.27%.

DRAM Bound rates increased when patches were enabled due mainly to increases in stalls while L3 cache miss load demands were waiting, CYCLE_ACTIVITY.STALLS_L3_MISS, and decreases in core cycles, CPU_CLK_UNHALTED.THREAD. The L2_Bound_Ratio also increased but had a smaller effect on the DRAM Bound results. *370.mgrid331* had a DRAM Bound rate increase of 45.08%, while others had increases of 37.41% *360.ilbc*, 31.32% for *359.botsspar*, and 21.42% for *371.applu331*. Store Bound rates also increased when patches were applied. This was the result of increases in the number of cycles when the store buffer was full, EXE_ACTIVITY.BOUND_ON_STORES, and a decrease in the number of core cycles, CPU_CLK_UNHALTED.THREAD. Two benchmarks, *360.ilbdc* and *371.applu331*, recorded gains of 39.25% and 17.06% respectively.

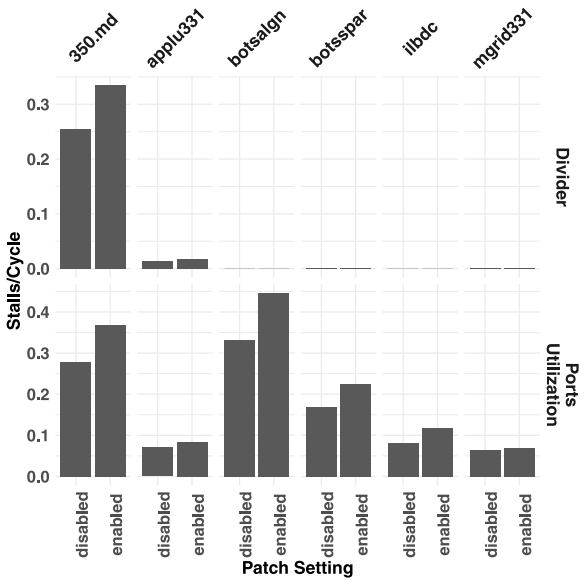
PPP rates for DRAM, L2, L3 and Store Bound metrics were consistently positive across all benchmarks, Fig. 5(b). This was the result of increasing stall rates across these metrics while the number of cycles decreased. There were fewer cycles to handle the increasing number of stalls, so the cycles became overvalued when the security patches were enabled. PPP rates also showed that while not all the benchmarks had significant stall rates in some of the categories, the impact of the patches was significant across all of them. That is the case of the Store Bound metric. For this category, *371.applu331*, and *360.ilbdc* had the largest Store Bound rates of at least 0.14, but the effects the patches had on all benchmarks were found to be of at least 19%, which was the case for *371.applu331* and as much as 53.46% for *359.botsspar*. Large relative changes, as reported by PPP rates, of a metric that is small in magnitude will not have a big effect on the overall Top-Down classification results, it does gives us information on the relative effect the security patches are having on the metric.

The L1 Bound PPP rates for *359.botsspar*, *360.ilbdc* and *370.mgrid331* were not computed because they provided no useful information since the regular rates were negative. The regular rates were negative because of the large increases in the execution stalls due to L1 cache misses, CYCLE_ACTIVITY.STALLS_L1D_MISS, when the security patches were enabled. Not all benchmarks reported negative L1 Bound rates. *350.md* and *358.botsalgn* followed the premise previously stated that an increasing number of stalls in combination with a decreasing number of core cycles resulted in positive PPP rates. The PPP rates were 34.87% for *350.md* and 48.35% for *358.botsalgn*. *371.applu331* had a negative PPP rate of −39.96% because its drop in the L1 Bound rate when the patches were applied.

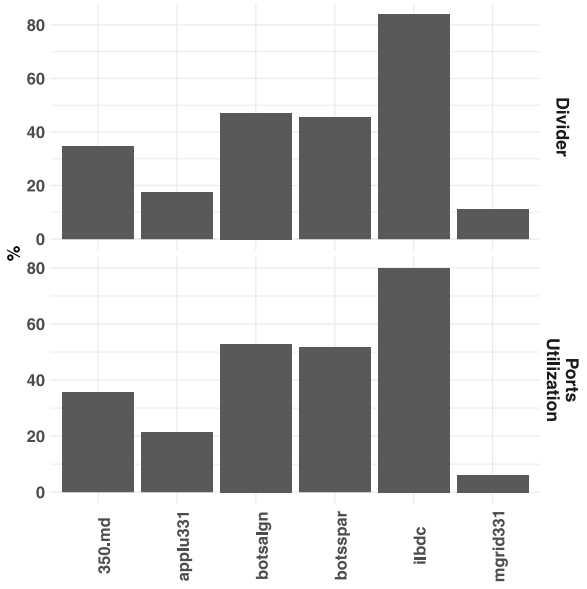
Core Bound is the second set of subcategories of the Backend Bound classification. They represent all non-memory related bottlenecks. The Divider metric tracks the fraction of cycles in which divide and square root operations used the DIV unit. When patches were enabled, the number of cycles when the divide unit was busy, ARITH.DIVIDER_ACTIVE, increased, while the number of CPU_CLK_UNHALTED.THREAD remained the same or decreased. Fig. 6(a) shows that the *360.ilbdc* benchmark had an increase of cycles that required root or division operations of 47.67% while the number of core cycles decreased by −19.65%. For benchmarks that had smaller increases in the Divider metric, the increase of cycles used in division and root operations was smaller, while the number of core cycles remained relatively the same. That is the case for *370.mgrid*, where CPU_CLK_UNHALTED.THREAD had an increase of 1.15% and an increase in ARITH.DIVIDER_ACTIVE of 13.63%.

The Ports Utilization metric tracks the fraction of CPU cycles affected by limitations in computational resources that do not involve the DIV unit. For benchmarks *350.md* and *371.applu331*, the performance counter ARITH.DIVIDER_ACTIVE was smaller in magnitude than EXE_ACTIVITY.EXE_BOUND_0_PORTS, as a result, the Ports Utilization metric depended only on the Core Bound metric and CPU_CLK_UNHALTED.THREAD. Both of these benchmarks reported increases in the Ports Utilization metric when patches were enabled. This was attributed to increases in the Core Bound rates while the number of core cycles decreased slightly. The other four benchmarks had higher EXE_ACTIVITY.EXE_BOUND_0_PORTS rates, so the formula used to compute Ports Utilization had to include the EXE_ACTIVITY.EXE_BOUND_0_PORTS performance counter in the computation of Ports Utilization. *358.botsalgn*, *359.botsspar*, and *360.ilbdc* had increases of 34.64%, 33.75% and 44.63% respectively when patches were enabled. This was the result of increases in Core Bound rates, and a decrease in core cycles. *370.mgrid* experienced only a 7.31% increase rate because there was a small increase in core cycles and a smaller increase in its Core Bound rate.

Ports Utilization and Divider PPP normalized rates followed the same patterns, Fig. 6(b). The rates were all positive, due to a decreasing number of CPU_CLK_UNHALTED.THREAD, while the Ports Utilization and Divider regular rates increased as the patches were enabled. The

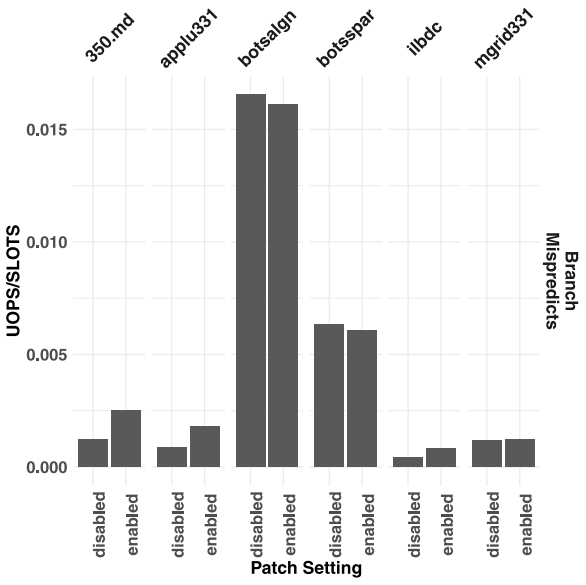


(a) Regular Rates

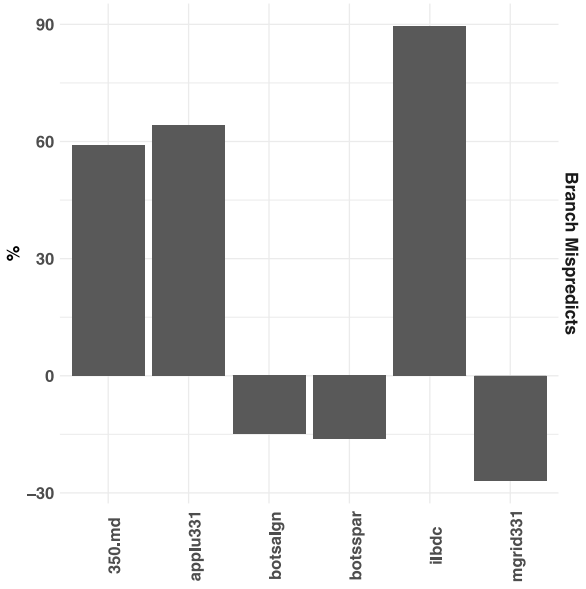


(b) PPP Normalized Rates

Fig. 6. Core Bound subcategories, which are part of the Backend Bound classification.



(a) Regular Rates



(b) PPP Normalized Rates

Fig. 7. Bad Speculation subcategories.

highest PPP rates occurred when the Ports utilization had the largest increase while the core cycles decreased the most. This is the case for *360.ilbdc*. It had a PPP rate of 80.01%, because of an increase in the Ports Utilization rate of 44.63% and a drop in the core cycle count of −19.65%. The same benchmark had the highest Divider rate, 83.79% which resulted from a Divider rate increase of 47.67%.

4.1.4. Bad speculation

The Bad Speculation metric is used to account for the slots that were wasted due to incorrect speculation. These uops will never get retired. In this study, we analysed one additional subcategory, Branch Mispredicts, which had relevance due to its rates. The Branch Mispredicts metric tracks slots that were affected by wasted uops that were fetched from an incorrectly speculated path, or stalls that occur when the out-of-order portion of the machine needs to recover its state from a speculative path. With patches enabled, *350.md* had an increase in the Branch Misprediction rate of 103.74% due to an increase in the Bad Speculation metric, Fig. 7(a). In this case, the branch misprediction machine clear fraction, BR_MISP_RETIRED.ALL_BRANCHES divided by difference between BR_MISP_RETIRED.ALL_BRANCHES and Machine_CLEARS.COUNT, rate stayed relatively the same while the number of bad speculation events increased. *360.ilbdc* and *371.applu331* had increases of 96.57% and 104.25% in their Branch Misprediction rates respectively, due to increases in the Bad Speculation rate and the misprediction machine clears fraction. These increases resulted from both, an increase of bad speculation events and the branch mispredictions machine clears fraction. *358.botsalgn* and *359.botsspar* had small Branch Misprediction decreases, less than −4% due to a decrease in the Bad Speculation rate, while the misprediction machine clears fraction remained the same. For these two benchmarks, the effect of the patches was a decrease in the number of bad speculation events resulting in a lower Branch Mispredicts rate. *370.mgrid* had a 6.82% increase due to an increase in the misprediction machine clears ratio.

PPP rates were affected by variations in the Branch Misprediction rates, since all benchmarks had increased in CPU core cycles Fig. 7(b). When patches were enabled, *350.md* had a PPP rate of 58.98%, *371.applu331* had a rate of 64.15%, and *360.ilbdc* had a rate of 89.50%. More work for relatively less number of core cycles resulted in the core cycles being overvalued when the security patches were enabled. The opposite is true for *358.boltsalgn* that a PPP rate of −14.92%, *359.botsspar* which had a rate of −16.15%, and *370.mgrid331* that had a PPP rate of −26.86%. For these benchmarks, the Branch Misprediction rates either dropped or they stayed relatively at the same levels, while the number of CPU core cycles increased. This resulted in less work for an increasing number of cycles making the core cycles undervalued.

5. Conclusion

In this study, we analysed the effects that the Spectre and Meltdown security patches had on CPU pipeline bottlenecks. Previous studies reported the effects patches had on performance, by focusing on two computationally intensive workflows [4] on an Intel based cluster, and on a diverse set of multiple benchmarks on different Cray based clusters [3]. The first study ran different tests under different conditions: before patches were applied, and with patches applied one at a time. This strategy was very comprehensive because some of the security patches, the BIOS and microcode fixes, could not be disabled once they were applied. The authors found that there was a negative effect when patches were applied and even when they disabled some of the patches via the vendor provided tunable feature, the performance degradation on their workflows was significant. The microcode and BIOS fixes had a major impact on performance. The second study reported minimal effect on their results. The systems used in their experiments were compared before and after all of the recommended patches were applied.

Our work compares the effects patches had on the CPU's pipeline by comparing Top-Down bottleneck metrics. We did not run experiments before all patches, including the microcode and BIOS fixes, were applied so the performance baseline included the Spectre, variant 1 fix. We compared the effects of the Spectre variant 2 and Meltdown variant 3 had on the test system. This comparison was possible because the OS vendor added tunable features that can enable or disable the two security patches, variant 2 and variant 3, to prevent a decrease in performance. To quantify relative changes of the metrics between the patch settings, we modified the Big Mac Index, a PPP theory based technique. This made it possible to compare Top-Down metric rates against a baseline performance counter, either *CPU_CLK_UNHALTED.THREAD*, or *CPU_CLK_UNHALTED.THREAD_ANY*. The goal was to determine if the number of cycles used for a given operation, stalls for instance, was relatively higher, lower or similar when compared to the same metric when the patches were disabled. This relative difference can be used to identify situations like the ones observed in the Backend Bound rates, Figs. 2(a) and 2(b), where the rates dropped or stayed the same for the regular rates, but the PPP normalized rates fell. This is the case for *371.applu331*, which had an increase in the Backend Bound rate of 5.70% but a decrease in the PPP rate of −15.05%. This drop was due to an increase of 24.43% in core cycles. Similarly for *370.mgrid331*, its regular Backend Bound rates stayed relative little change between patch settings, 3.90%. Its PPP normalized rate was found to be 28.86%, because its cycle count increased by 46.04% between patch settings. For both benchmarks, there were more cycles for the amount of stalls as compared to the baseline, so the cycles became overvalued.

Other techniques, such as the Roofline model [20], can give users an idea of how their code is performing relative to memory and floating-point peak performance. Another approach is to use statistical methods to model performance based on metrics such as cache hit rates and memory latencies [21]. These tools can provide information on how performance is affected when changes to the system settings or the code base are made. But they have some limitations. Statistical models provide information specific to the parameters that were used to create the model. These parameters were selected after being found to be of significance to the model. The Roofline model provides information of the changes made to the system configuration, or code changes in terms of memory and floating-point performance. Our study uses a more general technique that was applied to different metrics, including different categories of the Top-Down classification method.

We showed that Top-Down classification metrics varied when the security patches were enabled. We were able to quantify the relative changes when compared to a baseline run. Additionally, the use of PPP normalized rates made it possible to put into context the large percentage changes reported by the relative difference between metrics. The next step is to understand the effects these relative changes, which are not reflected in regular metrics, have on power efficiency. Our goal is to identify relationships between CPU pipeline bottlenecks and power efficiency before and after patches are applied. Other works have focused on the effect Spectre and Meltdown patches had on power efficiency by focusing in models based on performance metrics, for instance instructions-per-cycle, and branches-per-cycle, to develop models [22]. Our future work will focus in understanding the relation between PPP rates and power efficiency.

CRediT authorship contribution statement

Yectli A. Huerta: Conceptualization of this study, Methodology, Data curation, Writing – Original draft preparation. **David J. Lilja:** Editing of draft, final draft revision.

References

[1] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, Y. Yarom, Spectre attacks: Exploiting speculative execution, in: 40th IEEE Symposium on Security and Privacy, S&P'19, 2019.

[2] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, M. Hamburg, Meltdown: Reading kernel memory from user space, in: 27th USENIX Security Symposium, USENIX Security 18, 2018.

[3] V.G. Vergara Larrea, M.J. Brim, W. Joubert, S. Boehm, M. Baker, O. Hernandez, S. Oral, J. Simmons, D. Maxwell, Are we witnessing the spectre of an HPC meltdown? Concurr. Comput.: Pract. Exper. 31 (16) (2019) e5020, http://dx.doi.org/10.1002/cpe.5020, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5020, arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5020, e5020 cpe.5020.

[4] A. Prout, W. Arcand, D. Bestor, B. Bergeron, C. Byun, V. Gadepally, M. Houle, M. Hubbell, M. Jones, A. Klein, P. Michaleas, L. Milechin, J. Mullen, A. Rosa, S. Samsi, C. Yee, A. Reuther, J. Kepner, Measuring the impact of spectre and meltdown, in: 2018 IEEE High Performance Extreme Computing Conference, HPEC, 2018, pp. 1–5, http://dx.doi.org/10.1109/HPEC.2018.8547554.

[5] A. Yasin, A Top-Down method for performance analysis and counters architecture, in: 2014 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS, 2014, pp. 35–44.

[6] P.E. McKenney, Differential profiling, in: MASCOTS '95. Proceedings of the Third International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, 1995, pp. 237–241.

[7] K.W. Clements, Currencies, Commodities and Consumption, Cambridge University Press, 2013, http://dx.doi.org/10.1017/CBO9781139045612.

[8] Y.A. Huerta, B. Swartz, D.J. Lilja, Enhancing the top-down microarchitectural analysis method using purchasing power parity theory, in: International Workshop on Languages and Compilers for Parallel Computing, LCPC 2020, Springer, 2021.

[9] Y. Huerta, D. Lilja, Analysis of a ThunderX2 system using top-down and purchasing power parity methods, in: Practice and Experience in Advanced Research Computing, PEARC '21, Association for Computing Machinery, New York, NY, USA, 2021, http://dx.doi.org/10.1145/3437359.3467027.

[10] A. Kleen, pmu-tools: Intel PMU profiling tools, 2021, URL: https://github.com/andikleen/pmu-tools, (accessed on 18 June 2021).

[11] Red Hat, Inc., Meltdown & spectre - kernel side-channel attacks, 2018, URL: https://access.redhat.com/security/vulnerabilities/speculativeexecution, (accessed on 18 June 2021).

[12] Red Hat, Inc., Controlling the performance impact of microcode and security patches for CVE-2017-5754 CVE-2017-5715 and CVE-2017-5753 using red hat enterprise linux tunables, 2020, URL: https://access.redhat.com/articles/3311301, (accessed on 18 June 2021).

[13] Intel Corporation, Intel vtune profiler user guide, 2021, URL: https://software.intel.com/content/www/us/en/develop/documentation/vtune-help/top.html.
[14] Visual Capitalist, The latte index: Using the impartial bean to value currencies, 2017, URL: https://www.visualcapitalist.com/latte-index-currencies/, (accessed on 10 May 2021).
[15] The Economist Intelligence Unit, The iod index, 2017, URL: https://www.intelligenteconomist.com/the-ipod-index/, (accessed on 8 January 2021).
[16] The Economist, The big mac index, 2020, URL: https://www.economist.com/news/2020/01/15/the-big-mac-index, (accessed on 18 June 2021).
[17] Standard Performance Evaluation Corporation, SPEC OMP2012 documentation, URL: https://spec.org/omp2012/Docs/index.html.
[18] M.S. Müller, J. Baron, W.C. Brantley, H. Feng, D. Hackenberg, R. Henschel, G. Jost, D. Molka, C. Parrott, J. Robichaux, P. Shelepugin, M. van Waveren, B. Whitney, K. Kumaran, SPEC OMP2012 — An application benchmark suite for parallel systems using openmp, in: B.M. Chapman, F. Massaioli, M.S. Müller, M. Rorro (Eds.), OpenMP in a Heterogeneous World, Springer Berlin Heidelberg, Berlin, Heidelberg, 2012, pp. 223–236.
[19] Standard Performance Evaluation Corporation, SPEC OMP2012 results, URL: https://www.spec.org/omp2012/results/.
[20] S. Williams, A. Waterman, D. Patterson, Roofline: An insightful visual performance model for multicore architectures, Commun. ACM 52 (4) (2009) 65–76, http://dx.doi.org/10.1145/1498765.1498785.
[21] G. Ayers, J.H. Ahn, C. Kozyrakis, P. Ranganathan, Memory hierarchy for web search, in: 2018 IEEE International Symposium on High Performance Computer Architecture, HPCA, IEEE, 2018, pp. 643–656, http://dx.doi.org/10.1109/HPCA.2018.00061.
[22] B. Herzog, S. Reif, J. Preis, W. Schröder-Preikschat, T. Hönig, The price of meltdown and spectre: Energy overhead of mitigations at operating system level, in: Proceedings of the 14th European Workshop on Systems Security, EuroSec '21, Association for Computing Machinery, New York, NY, USA, 2021, pp. 8–14, http://dx.doi.org/10.1145/3447852.3458721.

Yectli A. Huerta is a Ph.D. candidate in Scientific Computation at the University of Minnesota. He works as an HPC Systems Administrator at the Minnesota Supercomputing Institute.

David J. Lilja received a Ph.D. and an M.S., both in Electrical Engineering, from the University of Illinois at Urbana-Champaign, and a B.S. in Computer Engineering from Iowa State University in Ames. He is currently Professor of Electrical and Computer Engineering at the University of Minnesota in Minneapolis, where he also serves as a member of the graduate faculties in Computer Science, Scientific Computation, and Data Science. He was elected a Fellow of the Institute of Electrical and Electronics Engineers (IEEE) and a Fellow of the American Association for the Advancement of Science (AAAS) for contributions to the statistical analysis of computer performance.

BenchCouncil Transactions on Benchmarks, Standards and Evaluations 1 (2021) 100013

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

journal homepage: https://www.keaipublishing.com/en/journals/benchcouncil-transactions-on-benchmarks-standards-and-evaluations/

Stars shine: The report of 2021 BenchCouncil awards

Taotao Zhan [a,*,1], Simin Chen [b,2]

[a] Beijing No. 4 Middle School International Campus, China
[b] University of Chinese Academy of Sciences, China

ARTICLE INFO

Keywords:
BenchCouncil awards and committees
Award selection rules
Awardees and their contributions

ABSTRACT

This report introduces the awards presented by the International Open Benchmark Council (BenchCouncil) in 2021 and highlights the award selection rules, committee, awardees, and their contributions.

1. The introduction of BenchCouncil 2021 awards

According to the decision of the BenchCouncil Steering committee, four awards were set up to encourage and reward scientists who have made contributions in benchmarking, standardizing, measuring, evaluating and optimizing: the Benchcouncil Achievement Award, the Benchcouncil Rising Star Award, the BenchCouncil Distinguished Doctoral Dissertation Award, and the BenchCouncil Bench conference Best Paper Award. In addition, Prof. Tony Hey donated to set up the BenchCouncil Bench conference Tony Hey Best Student Paper Award.

The virtual award ceremony was held at the 2021 BenchCouncil International Symposium on Benchmarking, Measuring, and Optimizing (Bench'21). The Bench'21 General Chairs are Prof. Resit Sendag from the University of Rhode Island, USA, Dr. Arne J. Berre from SINTEF Digital, Norway. The Program Chairs are Dr. Lei Wang from ICT, Chinese Academy of Sciences, China; Prof. Axel Ngonga, from Paderborn University, Germany; and Prof. Chen Liu from Clarkson University, USA. The Special Session Chair is Prof. Xiaoyi Lu from The University of California, Merced, USA. Three associate coordinators from BenchCouncil: Ke Liu, Simin Chen, Fanda Fan from the University of Chinese Academy of Sciences, and one coordinator from BenchCouncil: Shaopeng Dai from ICT, Chinese Academy of Sciences, provided technical services for organizing this virtual event.

2. The BenchCouncil Achievement Award

This award recognizes a senior member who has made long-term contributions to benchmarking, standardizing, measuring, evaluating and optimizing. The winner will automatically become BenchCouncil Fellow and join the BenchCouncil Achievement and Rising Star Award Committees the following year. The award carries a $3,000 honorarium.

2.1. Award committee

The 2021 BenchCouncil Achievement Award committee consists of six members. They are Prof. Lizy John from the University of Texas at Austin (architecture), Prof. Geoffrey Fox from Indiana University (systems and applications), Prof. D.K. Panda from the Ohio State University (high-performance computing), Prof. Jianfeng Zhan from the Chinese Academy of Sciences (systems, architecture, and applications), Prof. Tony Hey from Rutherford Appleton Laboratory STFC (systems and applications), and Prof. David Lilja from the University of Minnesota (high-performance computing and architecture).

2.2. Award selection rule [1]

Only the award committee members can nominate the candidates. Each committee member can nominate one person. A coordinator in the award committee is responsible for collecting the nomination and votes. After receiving the nominations, the coordinator will send the nomination information to all members. Before the voting is over, the nominator is anonymous. After that, all details of who nominated who and who voted will be disclosed within the committee. The candidate who gets the highest votes will become the winner.

2.3. The awardee and contribution

BenchCouncil Achievement Award is given to Dr. Jack J. Dongarra from the University of Tennessee. He specializes in numerical algorithms in linear algebra, parallel computing, the use of advanced computer architectures, programming methodology, and tools for parallel computers. His research includes developing, testing, and documentation of high-quality mathematical software. According to the decision

* Corresponding author.
E-mail addresses: dtaotaozhan@gmail.com (T. Zhan), chensimin000@gmail.com (S. Chen).
[1] Assistant coordinator.
[2] Associate coordinator.

https://doi.org/10.1016/j.tbench.2021.100013

Available online 24 December 2021
2772-4859/

Fig. 1. Prof. Jack. J. Dongarra is honored with 2021 BenchCouncil Achievement Award for the "novel and substantial contribution to development, testing and documentation of high-quality mathematical software" and "benchmarking HPC systems".

Fig. 2. Prof. Jack. J. Dongarra's award certification.

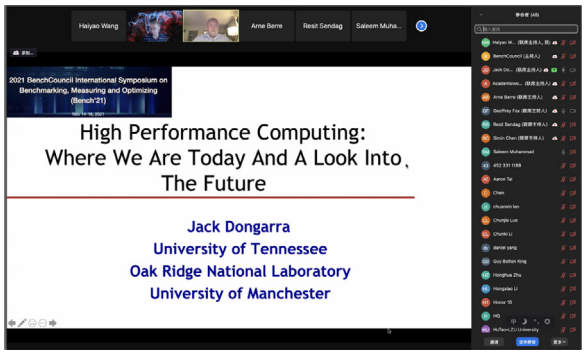


Fig. 3. Prof. Jack. J. Dongarra delivered a keynote speech at the Bench21 award ceremony. Prof. Geoffrey Fox from Indiana University chaired the speech.

of the award committee, Prof. Jack J. Dongarra has been selected for the "novel and substantial contribution to development, testing and documentation of high-quality mathematical software [2–4]" and "benchmarking HPC systems [5,6]".

Dr. Jack J. Dongarra is a Member of the US National Academy of Engineering, a Foreign Member of the Russian Academy of Sciences, and a Foreign Fellow of the British Royal Society (see Figs. 1–3).

3. BenchCouncil Rising Star Award

This award recognizes young researchers who demonstrate outstanding research and practice in benchmarking, standardizing, measuring, evaluating, and optimizing. The winner will automatically become BenchCouncil Senior Fellow and join the BenchCouncil Rising Star Award Committee the following year. The award carries a $1,000 honorarium.

Fig. 4. Dr. Petter Mattson from Google is honored with 2021 BenchCouncil Rising Star Award. Prof. Jianfeng Zhan from Chinese Academy of Sciences chaired the award ceremony.

3.1. Award committee

The 2021 BenchCouncil Rising Star Award committee consists of seven members. They are Prof. Lizy John from the University of Texas at Austin (architecture), Prof. Geoffrey Fox from Indiana University (systems and applications), Prof. D.K. Panda from the Ohio State University (high-performance computing), Prof. Jianfeng Zhan from the Chinese Academy of Sciences (systems, architecture, and applications), Prof. Tony Hey from Rutherford Appleton Laboratory STFC (systems and applications), and Prof. David Lilja from the University of Minnesota (high-performance computing and architecture), and Prof. Torsten Hoefler from ETH Zurich (high-performance computing).

3.2. Award selection rule [7]

Only the award committee members can nominate the candidates. Each committee member can nominate one researcher or a group of researchers up to three people who co-advance the state-of-the-art and state-of-the-practice in the same field. A coordinator in the award committee is responsible for collecting the nomination and votes. After receiving the nominations, the coordinator will send the nomination information to all members. Before the voting is over, the nominator is anonymous. After that, all details of who nominated who and who voted will be disclosed within the committee. The candidate (one researcher or a group of researchers) who gets the highest votes will win.

3.3. Awardees and their contributions

Three primary contributors to the MLPerf and AIBench projects were honored with the 2021 BenchCouncil Rising Star Award: Dr. Peter Mattson from Google, Prof. Dr. Vijay Janapa Reddi from Harvard University, and Prof. Dr. Wanling Gao from the Chinese Academy of Sciences.

Dr. Peter Mattson was chosen for the contributions "as a lead researcher, proposing AI training benchmarks and performing large-scale industry testing [8,9]" and "co-proposing memory access scheduling technique that reorders memory references to exploit locality within the 3-D memory structure [10]".

Dr. Peter Mattson leads the ML Performance Measurement at Google. He co-founds and is General Chair of the MLPerf consortium. Previously, he founded the Programming Systems and Applications Group at NVIDIA Research. He ever was V.P. of software infrastructure for Stream Processors Inc (SPI), and a managing engineer at Reservoir Labs. His research focuses on accelerating and understanding the behavior of machine learning systems by applying novel benchmarks and analysis tools (see Figs. 4–6).

Dr. Wanling Gao was selected for the contributions "as one of the primary researchers, proposing AI scenario [11], AI training [12], and

Fig. 5. Dr. Petter Mattson's Award Certificate.

Fig. 8. Dr. Wanling Gao's Award Certificate.

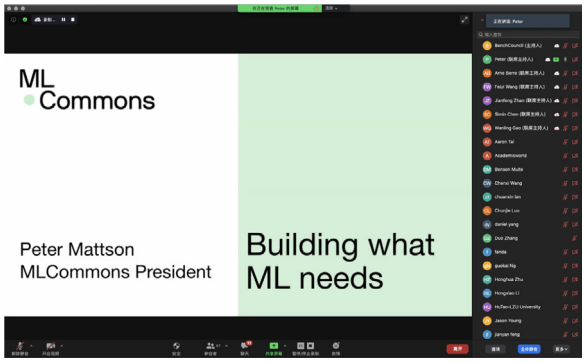


Fig. 6. Dr. Petter Mattson delivered a keynote speech at the Bench 21 award ceremony.

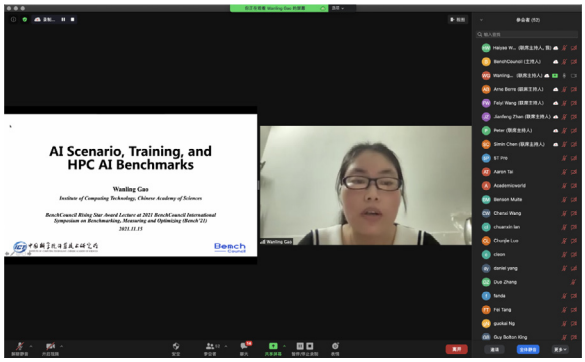


Fig. 9. Dr. Wanling Gao delivered a keynote speech at the Bench 21 award ceremony.

Fig. 7. Dr. Wanling Gao from Chinese Academy of Sciences is honored with 2021 BenchCouncil Rising Star Award. Prof. Jianfeng Zhan from Chinese Academy of Sciences chaired the award ceremony.

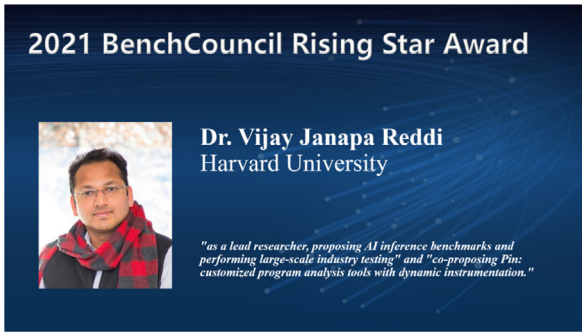


Fig. 10. Dr. Vijay Janapa Reddi from Harvard University is honored with the 2021 BenchCouncil Rising Star Award. Prof. Jianfeng Zhan from Chinese Academy of Sciences chaired the award ceremony.

HPC AI benchmarks [13]" and "proposing a data motif abstraction that tries to unify the big data and AI workloads [14]".

Dr. Wanling Gao is an Associate Professor at the Institute of Computing Technology, Chinese Academy of Sciences. Dr. Wanling Gao received her B.S. degree from Huazhong University of Science and Technology in 2012, and her Ph.D. degree from the Institute of Computing Technology, the Chinese Academy of Sciences, and the University of Chinese Academy of Sciences in 2019. Her works focus on big data and AI benchmarking, workload characterization, computer architecture, and proxy benchmarks for simulation (see Figs. 7–9).

Dr. Vijay Janapa Reddi was selected for the contributions "as a lead researcher, proposing AI inference benchmarks and performing large-scale industry testing [9,15]" and "co-proposing Pin: customized program analysis tools with dynamic instrumentation [16]".

Dr. Vijay Janapa Reddi is an Associate Professor in the John A. Paulson School of Engineering and Applied Sciences (SEAS) at Harvard University. His research is centered on mobile and edge-centric computing systems (see Figs. 10–12).

4. BenchCouncil Distinguished Doctoral Dissertation Award

This award recognizes and encourages superior research and writing by doctoral candidates in the broad field of benchmarking, standardizing, measuring, evaluating, and optimizing. Among the submissions, four candidates will be selected as finalists. They will be invited to give a 30-minute presentation at the BenchCouncil Bench Conferences and contribute research or survey articles to BenchCouncil Transactions on Benchmarks, Standards, and Evaluation. Finally, one among the four will receive the award, which carries a $1,000 honorarium.

Fig. 11. Dr. Vijay Janapa Reddi's Award Certificate.

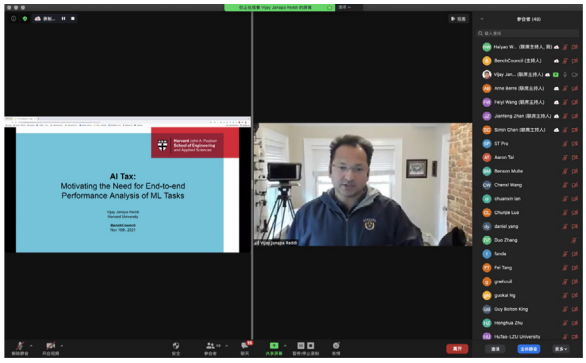


Fig. 12. Dr. Vijay Janapa Reddi delivered a keynote speech at the Bench 21 award ceremony.

4.1. Award committee

The 2021 BenchCouncil Distinguished Doctoral Dissertation Award consists of five members. They are Prof. Jack Dongarra from University of Tennessee (high-performance computing), Prof. Dr. Xiaoyi Lu from The University of California, Merced (high-performance computing), Dr. Jeyan Thiyagalingam from STFC-RAL (scientific Computing and AI), Dr. Lei Wang, ICT, Chinese Academy of Sciences (Systems and architecture), Prof. Dr. Spyros Blanas from The Ohio State University (data management). The committee members cannot nominate their students.

4.2. Award selection rule [17]

The committee welcomes the proposals from the following but not limited to the following communities: architecture, systems, database, high-performance computing, machine learning or AI, scientific computing, medicine, or other disciplines.

Only those awarded Ph.D. in the past two years are eligible for this award. Only the accepted final version of a nominated Ph.D. dissertation will be considered, and it must have been filed with the writer's institution during the nomination cycle. The writer or the writer's Ph.D. advisor can nominate a dissertation, nominated only once. The benchmarks, data, or tools that are the essential contributions of the dissertation should be open-sourced.

At least two supporting letters should be included from experts in the field who can provide additional insights or evidence of the dissertation's impact. The nominator/advisor may not write a letter of support. Each letter should include the name, address, and telephone number of the endorser. The nominator should collect the letters and bundle them for submission.

Fig. 13. Dr. Romain Jacob is selected as one of the finalists for the 2021 BenchCouncil Distinguished Doctoral Dissertation Award. Prof. Xiaoyi Lu from The University of California, Merced chaired the finalist speech.

In the first round, the four candidates will be singled out. Each one will give a 30-minute presentation in the distinguished Ph.D. dissertation session in the Bench conference https://www.benchcouncil.org/conferences.html#benchconf chaired by the committee.

Each finalist must submit an article to the BenchCouncil Transactions on Benchmarks, Standards, and Evaluations (TBench). In advance, each submitter is encouraged to submit a survey article or a research article to TBench. The article should include the following contents. (a) The fundamental issue your dissertation tackles. Why is it essential and challenging? (10%). (b) The summary of state-of-the-art and state-of-the-practice (30%). (c) How do you advance state-of-the-art and state-of-the-practice? What are your innovative approaches, systems, tools, and insights? (40%) (d) Open issues and future work (20%).

The article that is submitted to TBench will be reviewed for technical depth and significance of the research contribution, the potential impact on theory and practice. Finally, one among the four finalists will receive the award. The committee will present the award at the award ceremony of the Bench conference.

4.3. Finalists and citations

Dr. Romain Jacob from ETH Zurich, Dr. Pei Guo from University of Maryland, Baltimore County (UMBC), Dr. Kai Shu from Arizona State University, and Dr. Belen Bermejo from University of the Balearic Islands are selected as the finalists.

Dr. Romain Jacob completed his doctoral studies under the supervision of Prof. Lothar Thiele at ETH in 2019. His dissertation is entitled Leveraging Synchronous Transmissions for the Design of Real-time Wireless Cyber–Physical Systems. He was suggested for "his leading efforts in establishing benchmarks for low-power wireless networking and for the development of a concrete methodology to foster the replicability of networking experiments [18,19]".

Dr. Romain Jacob's advisor, Prof. Lothar Thiele, is a Full Professor of Computer Engineering at ETH Zurich. His research interests include models, methods, and software tools for designing real-time embedded systems, the internet of things, cyber–physical systems, sensor networks, embedded software, and bioinspired optimization techniques (see Figs. 13–15).

Currently, Dr. Romain Jacob is a postdoctoral researcher at ETH Zurich in the group of Prof. Laurent Vanbever. His current interests focus on computer networks, communication protocols, (real-time) scheduling theory, and statistics applied to experimental design.

Dr. Pei Guo received her Ph.D. in Information Systems from the University of Maryland, Baltimore County, in 2021. Her dissertation is entitled Scalable Multivariate Causality Discovery From Large-scale Global Spatio-temporal Climate Data. She was selected for "the interdisciplinary research on causality discovery approaches for climate

Fig. 14. Dr. Romain Jacob completed his doctoral studies under the supervision of Prof. Lothar Thiele.

Fig. 15. Dr. Romain Jacob's Finalist Certificate.

Fig. 16. Dr. Pei Guo is selected as one of the finalists for the 2021 BenchCouncil Distinguished Doctoral Dissertation Award. Prof. Xiaoyi Lu from The University of California, Merced, chaired the finalist speech.

data and extensive benchmarking of proposed work via both synthetic datasets and real-world applications [20,21]".

Dr. Pei Guo's advisor, Prof. Jianwu Wang, is an Associate Professor at the Department of Information Systems, University of Maryland, Baltimore County (UMBC). His current research interests include Big Data Analytics, Distributed Computing, and Scientific Workflows with an application focusing on climate and manufacturing (see Figs. 16–18).

Currently, Dr. Pei Guo is working as a Data Scientist at Wyze Labs. She involves in the research on spatiotemporal causal modeling on large-scale data, big data application parallelizing, and cloud computing.

Dr. Kai Shu obtained his Ph.D. in Computer Science at Arizona State University in 2020 and was the recipient of the 2020 ASU Engineering

Fig. 17. Dr. Pei Guo completed his doctoral studies under the supervision of Prof. Jianwu Wang.

Fig. 18. Dr. Pei Guo's Finalist Certificate.

Fig. 19. Dr. Kai Shu is selected as one of the finalists for the 2021 BenchCouncil Distinguished Doctoral Dissertation Award. Prof. Xiaoyi Lu from The University of California, Merced, chaired the finalist speech.

Dean's Dissertation Award. His dissertation is entitled Understanding Disinformation: Learning with Weak Social Supervision. He was suggested for "creating, curating and maintaining FakeNewsNet - a widely used, de facto benchmark data repository on Fake News Detection in his doctoral dissertation [22,23]".

Dr. Kai Shu's advisor, Prof. Huan Liu, is a professor of Computer Science and Engineering at Arizona State University. He is a Fellow of ACM, AAAI, AAAS, and IEEE. His research interests are in data mining, machine learning, social computing, and artificial intelligence, investigating interdisciplinary problems that arise in many real-world, data-intensive applications with high-dimensional data of disparate forms such as social media.

Currently, Dr. Kai Shu is a Gladwin Development Chair Assistant Professor in the Department of Computer Science at Illinois Institute of Technology since Fall 2020. His research and computational tool development address challenges on fake news detection, explainable

Fig. 20. Dr. Kai Shu completed his doctoral studies under the supervision of Prof. Huan Liu.

Fig. 21. Dr. Kai Shu's Finalist Certificate.

Fig. 22. Dr. Belen Bermejo is selected as one of the finalists for the 2021 BenchCouncil Distinguished Doctoral Dissertation Award. Prof. Xiaoyi Lu from The University of California, Merced, chaired the finalist speech.

machine learning, trust social computing, and social media mining (see Figs. 19–21).

Dr. Belen Bermejo obtained a Ph.D. degree in 2020 at the University of the Balearic Islands. Her dissertation is entitled Performance and Energy Consumption Tradeoff in Server Consolidation. She was suggested for "the creation of the CiS2 index which is based on monitoring and benchmarking to manage the trade-off between power consumption and performance in virtualized servers [24,25]".

Dr. Belen Bermejo's advisor, Prof. Carlos Juiz, is heading the ACSIC research group (http://acsic.uib.es) at the University of the Balearic Islands. He is a senior member of the IEEE and a senior member of the ACM. His research interest mainly focuses on performance engineering, Green IT, and IT governance (see Figs. 22–24).

Currently, Dr. Belen Bermejo is an assistant lecture at the University of the Balearic Islands and a member of the ACSIC research group (http://acsic.uib.es). Her researches focus on the performance and energy consumption of virtualized systems.

Fig. 23. Dr. Belen Bermejo completed his doctoral studies under the supervision of Prof. Carlos Juiz.

Fig. 24. Dr. Belen Bermejo's Finalist Certificate.

Fig. 25. The group photo of the awardees of the Best Paper Award. From left to right are Ross Miller, Dr. Aristeidis Tsaris, Dr. Junqi Yin (the first author), Dr. Sajal Dash, Dr. Mallikarjun (Arjun) Shankar, and Dr. Feiyi Wang from Oak Ridge National Laboratory.

5. BenchCouncil Bench 21 Best Paper Award and Tony Hey Best Student Paper Award

5.1. BenchCouncil Bench 21 Best Paper Award

A group of computer scientists at Oak Ridge National Laboratory (ORNL) received the best paper award for the paper titled "Comparative Evaluation of Deep Learning Workload for Leadership-class Systems [26]" (see Figs. 25–27).

Since deep learning (DL) applications rely heavily on DL frameworks and underlying compute (CPU/GPU) stacks, it is essential to

Fig. 26. Prof. Tony Hey chaired the award ceremony of the best paper award.

Fig. 27. Best Paper Award Certificate.

gain a holistic understanding from compute kernels, models, and frameworks of popular DL stacks, and to assess their impact on science-driven, mission-critical applications. This paper employs a set of micro and macro DL benchmarks established through the Collaboration of Oak Ridge, Argonne, and Livermore (CORAL) to evaluate the AI readiness of the next-generation supercomputers. This paper presents the early observations and performance benchmark comparisons between the Nvidia V100 based Summit system with its CUDA stack and an AMD MI100 based testbed system with its ROCm stack.

The following introduces each author. Dr. Junqi Yin is a computational scientist in Analytics & AI Methods at Scale (AAIMS) group of National Center for Computational Sciences at ORNL. His research interests range from scalable machine learningdeep learning. Dr. Aristeidis Tsaris is a research scientist in the Analytics & AI Methods at Scale (AAIMS) group of National Center for Computational Sciences at ORNL. His research focus is on scalable machine learning applications on HPC systems, benchmarking, and imaging. Ross Miller has B.S. and M.S. degrees in computer science. He has been working at ORNL for 12 years on a variety of supercomputer-related topics including filesystems for SSD's, data archiving systems and exploring ARM architecture for HPC use. Dr. Sajal Dash is currently a postdoctoral research associate at Analytics & AI Methods at Scale (AAIMS) group of National Center for Computational Sciences at ORNL. His research interests include exploring scaling approaches for large-scale deep learning applications.

Dr. Feiyi Wang is a Senior Research Scientist and Group Leader of Analytics and AI methods at Scale Group (AAIMS) at National Center for Computational Sciences of ORNL. He is also a Senior Member of IEEE. His research interests include large-scale data analytics, distributed machine learning and benchmarking, high-performance storage systems, parallel I/O, and file systems. Dr. Mallikarjun (Arjun) Shankar is the Section Head for the Advanced Technologies Section (ATS) in the National Center for Computational Science at ORNL. He is the director of the Compute and Data Environment for Science (CADES) at ORNL. He is a member of the AAAS, a Senior Member of the ACM, and a Senior Member of the IEEE. His research involves designing large-scale data analysis, modeling systems, sensor networking systems, energy grid monitoring, and control frameworks, and deploying middleware to overlay data, computation, and control across systems and infrastructure.

Fig. 28. The group photo of the awardees of the Tony Hey Best Student Paper Award. From left to right are Prof. Guangzhong Sun, Dr. Jingwei Sun, Zhongtian Xu, and Jiaqiang Liu (the first author) from the University of Science and Technology of China.

5.2. BenchCouncil Bench 21 Tony Hey Best Student Paper award

A graduate student, Jiaqiang Liu, from the University of Science and Technology of China, and the other members, supervised by Prof. Guangzhong Sun, received the Tony Hey Best Student Paper Award for the paper titled 11Latency-Aware Automatic CNN Channel Pruning with GPU Runtime Analysis [27]" (see Figs. 28–30).

The huge storage and computation cost of convolutional neural networks (CNN) make them challenging to meet the real-time inference requirement in many applications. This paper proposes a latency-aware automatic CNN channel pruning method (LACP) to search for low latency and accurate pruned network structure automatically. The inference latency of convolutional layers on GPU is analyzed to bridge model pruning and inference acceleration. Results show that the inference latency of convolutional layers exhibits a staircase pattern along with channel number due to the GPU tail effect. Based on that observation, the search space of network structures is greatly shrunk. Then an evolutionary procedure is applied to search a computationally efficient pruned network structure, which reduces the inference latency and maintains the model accuracy. Experiments and comparisons with state-of-the-art methods on three image classification datasets show that the method in this paper can achieve better inference acceleration with less accuracy loss.

The short introduction of each author is as follows. Jiaqiang Liu is a graduate student at the University of Science and Technology of China. His research interests include high-performance computing and performance modeling. Dr. Jingwei Sun is currently a postdoctoral researcher with the School of Computer Science and Technology, University of Science and Technology of China. His research interests include high-performance computing, performance modeling, and algorithm optimization. Zhongtian Xu is a graduate student at the University of Science and Technology of China. His research interests include high-performance computing and algorithm optimization. Dr. Guangzhong Sun is a professor at the School of Computer Science and Technology, University of Science and Technology of China. He is also

Fig. 29. Prof. Tony Hey chaired the award ceremony of the Tony Hey Best Paper Award.

Fig. 30. The Tony Hey Best Paper Award Certificate.

a member of the National High-Performance Computing Center (Hefei) and the principal investigator of the Algorithm and Data Application (Ada) Research Group. His research interests include high-performance computing, algorithm optimizations, and data processing.

Acknowledgments

Taotao Zhan contributes Sections One, Two, and Three; Simin Chen contributes Sections Four and Five. We are very grateful to BenchCouncil Steering Committee, Award Committee for presenting these awards, and Bench 21 general chairs, program chairs, and other teams for organizing the excellent Bench 21 event. For simplicity, we do not list their names one by one.

References

[1] L. John, G. Fox, D. Panda, J. Zhan, T. Hey, D. Lilja, The 2021 BenchCouncil achievement award selection rule, 2021, https://www.benchcouncil.org/html/awards.html#achievement, accessed at Dec 2, 2021.

[2] J.J. Dongarra, C.B. Moler, J.R. Bunch, G.W. Stewart, LINPACK users' guide, SIAM, 1979.

[3] J.J. Dongarra, J. Du Croz, S. Hammarling, I.S. Duff, A set of level 3 basic linear algebra subprograms, ACM Trans. Math. Softw. 16 (1) (1990) 1–17.

[4] R.C. Whaley, J.J. Dongarra, Automatically tuned linear algebra software, in: SC'98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing, IEEE, 1998, p. 38.

[5] P.R. Luszczek, D.H. Bailey, J.J. Dongarra, J. Kepner, R.F. Lucas, R. Rabenseifner, D. Takahashi, The HPC Challenge (HPCC) benchmark suite, in: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, 2006, pp. 1188455–1188677.

[6] J. Dongarra, M.A. Heroux, P. Luszczek, Hpcg benchmark: a new metric for ranking high performance computing systems, Knoxville, Tennessee (2015) 42.

[7] L. John, G. Fox, D.K. Panda, J. Zhan, T. Hey, D. Lilja, T. Hoefler, The 2021 BenchCouncil rising star award selection rule, 2021, https://www.benchcouncil.org/html/awards.html#risingstar, accessed at Dec 2, 2021.

[8] P. Mattson, C. Cheng, C. Coleman, G. Diamos, P. Micikevicius, D. Patterson, H. Tang, G.-Y. Wei, P. Bailis, V. Bittorf, et al., Mlperf training benchmark, 2019, arXiv preprint arXiv:1910.01500.

[9] P. Mattson, V.J. Reddi, C. Cheng, C. Coleman, G. Diamos, D. Kanter, P. Micikevicius, D. Patterson, G. Schmuelling, H. Tang, et al., Mlperf: An industry standard benchmark suite for machine learning performance, IEEE Micro 40 (2) (2020) 8–16.

[10] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, J.D. Owens, Memory access scheduling, ACM SIGARCH Comput. Archit. News 28 (2) (2000) 128–138.

[11] W. Gao, F. Tang, J. Zhan, X. Wen, L. Wang, Z. Cao, C. Lan, C. Luo, X. Liu, Z. Jiang, Aibench scenario: Scenario-distilling ai benchmarking, in: 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT), IEEE, 2021, pp. 142–158.

[12] F. Tang, W. Gao, J. Zhan, C. Lan, X. Wen, L. Wang, C. Luo, Z. Cao, X. Xiong, Z. Jiang, et al., Aibench training: balanced industry-standard AI training benchmarking, in: 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), IEEE, 2021, pp. 24–35.

[13] Z. Jiang, W. Gao, F. Tang, L. Wang, X. Xiong, C. Luo, C. Lan, H. Li, J. Zhan, Hpc ai500 v2. 0: The methodology, tools, and metrics for benchmarking hpc ai systems, in: 2021 IEEE International Conference on Cluster Computing (CLUSTER), IEEE, 2021, pp. 47–58.

[14] W. Gao, J. Zhan, L. Wang, C. Luo, D. Zheng, F. Tang, B. Xie, C. Zheng, X. Wen, X. He, et al., Data motifs: A lens towards fully understanding big data and ai workloads, in: Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, 2018, pp. 1–14.

[15] V.J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, et al., Mlperf inference benchmark, in: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), IEEE, 2020, pp. 446–459.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, K. Hazelwood, Pin: building customized program analysis tools with dynamic instrumentation, Acm Sigplan. Notic. 40 (6) (2005) 190–200.

[17] J. Dongarra, X. Lu, J. Thiyagalingam, L. Wang, S. Blanas, The 2021 BenchCouncil distinguished doctoral dissertation selection rule, 2021, https://www.benchcouncil.org/html/awards.html#doctor, accessed at Dec 2, 2021.

[18] C.A. Boano, S. Duquennoy, A. Förster, O. Gnawali, R. Jacob, H.-S. Kim, O. Landsiedel, R. Marfievici, L. Mottola, G.P. Picco, et al., IoTBench: Towards a benchmark for low-power wireless networking, in: 2018 IEEE Workshop on Benchmarking Cyber-Physical Networks and Systems (CPSBench), IEEE, 2018, pp. 36–41.

[19] R. Jacob, A.-B. Schaper, A. Biri, R. Da Forno, L. Thiele, Synchronous transmissions on bluetooth 5 and IEEE 802.15. 4–a replication study, in: 3rd Workshop on Benchmarking Cyber-Physical Systems and Internet of Things (CPS-IoTBench 2020), ETH Zurich, Computer Engineering and Networks Laboratory (TIK), 2020.

[20] S. Hussung, S. Mahmud, A. Sampath, M. Wu, P. Guo, J. Wang, Evaluation of data-driven causality discovery approaches among dominant climate modes, UMBC Fac. Collect. (2019).

[21] Y. Huang, M. Kleindessner, A. Munishkin, D. Varshney, P. Guo, J. Wang, Benchmarking of data-driven causality discovery approaches in the interactions of arctic sea ice and atmosphere, Front. Big Data 4 (2021).

[22] K. Shu, D. Mahudeswaran, S. Wang, D. Lee, H. Liu, Fakenewsnet: A data repository with news content, social context, and spatiotemporal information for studying fake news on social media, Big Data 8 (3) (2020) 171–188.

[23] K. Shu, A. Sliva, S. Wang, J. Tang, H. Liu, Fake news detection on social media: A data mining perspective, ACM SIGKDD Explor. Newsl. 19 (1) (2017) 22–36.

[24] B. Bermejo, C. Juiz, C. Guerrero, On the linearity of performance and energy at virtual machine consolidation: the cis2 index for cpu workload in server saturation, in: 2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS), IEEE, 2018, pp. 928–933.

[25] B. Bermejo, C. Juiz, C. Guerrero, Virtualization and consolidation: a systematic review of the past 10 years of research on energy and performance, J. Supercomput. 75 (2) (2019) 808–836.

[26] J. Yin, A. Tsaris, S. Dash, R. Miller, F. Wang, M.A. Shankar, Comparative evaluation of deep learning workloads for leadership-class systems, BenchCouncil Trans. Benchmarks Stand. Eval. (2021) 100005.

[27] J. Liu, J. Sun, Z. Xu, G. Sun, Latency-aware automatic CNN channel pruning with GPU runtime analysis, BenchCouncil Trans. Benchmarks Stand. Eval. (2021) 100009.

Taotao Zhan is a first-year high school student in Beijing No. 4 Middle School International Campus. She is a volunteer for BenchCouncil as an assistant coordinator in media communication and conference organization. She feels interested in science and technology media.

Simin Chen is a senior student at University of Chinese Academy of Sciences. She is a volunteer for BenchCouncil as an associate coordinator. She majors in Computer Science and Technology and is interested in performance evaluation and optimization.

TBench Editorial Board

More information about this series at https://www.benchcouncil.org/tbench/

TBench Call For Papers

BenchCouncil Transactions on Benchmarks, Standards and Evaluations (TBench)
ISSN:2772-4859

Aims and Scopes

BenchCouncil Transactions on Benchmarks, Standards, and Evaluations (TBench) publishes position articles that open new research areas, research articles that address new problems, methodologies, tools, survey articles that build up comprehensive knowledge, and comments articles that argue the published articles. The submissions should deal with the benchmarks, standards, and evaluation research areas. Particular areas of interest include, but are not limited to:

- 1. Generalized benchmark science and engineering (see https://www.sciencedirect.com/science/article/pii/S2772485921000120), including but not limited to
  - measurement standards
  - standardized data sets with defined properties
  - representative workloads
  - representative data sets
  - best practices
- 2. Benchmark and standard specifications, implementations, and validations of:
  - Big Data
  - AI
  - HPC
  - Machine learning
  - Big scientific data
  - Datacenter
  - Cloud
  - Warehouse-scale computing
  - Mobile robotics
  - Edge and fog computing
  - IoT
  - Chain block
  - Data management and storage
  - Financial domains
  - Education domains
  - Medical domains
  - Other application domains
- 3. Data sets
  - Detailed descriptions of research or industry datasets, including the methods used to collect the data and technical analyses supporting the quality of the measurements.
  - Analyses or meta-analyses of existing data and original articles on systems, technologies, and techniques that advance data sharing and reuse to support reproducible research.
  - Evaluating the rigor and quality of the experiments used to generate the data and the completeness of the data description.
  - Tools generating large-scale data while preserving their original characteristics.
- 4. Workload characterization, quantitative measurement, design, and evaluation studies of:
  - Computer and communication networks, protocols, and algorithms
  - Wireless, mobile, ad-hoc and sensor networks, IoT applications
  - Computer architectures, hardware accelerators, multi-core processors, memory systems, and storage networks
  - High-Performance Computing
  - Operating systems, file systems, and databases

- Virtualization, data centers, distributed and cloud computing, fog, and edge computing
- Mobile and personal computing systems
- Energy-efficient computing systems
- Real-time and fault-tolerant systems
- Security and privacy of computing and networked systems
- Software systems and services, and enterprise applications
- Social networks, multimedia systems, Web services
- Cyber-physical systems, including the smart grid

- 5. Methodologies, metrics, abstractions, algorithms, and tools for:
  - Analytical modeling techniques and model validation
  - Workload characterization and benchmarking
  - Performance, scalability, power, and reliability analysis
  - Sustainability analysis and power management
  - System measurement, performance monitoring, and forecasting
  - Anomaly detection, problem diagnosis, and troubleshooting
  - Capacity planning, resource allocation, run time management, and scheduling
  - Experimental design, statistical analysis, simulation
- 6. Measurement and evaluation
  - Evaluation methodology and metric
  - Testbed methodologies and systems
  - Instrumentation, sampling, tracing, and profiling of Large-scale real-world applications and systems
  - Collection and analysis of measurement data that yield new insights
  - Measurement-based modeling (e.g., workloads, scaling behavior, assessment of performance bottlenecks)
  - Methods and tools to monitor and visualize measurement and evaluation data
  - Systems and algorithms that build on measurement-based findings
  - Advances in data collection, analysis, and storage (e.g., anonymization, querying, sharing)
  - Reappraisal of previous empirical measurements and measurement-based conclusions
  - Descriptions of challenges and future directions the measurement and evaluation community should pursue