BenchCouncil Transactions on Benchmarks, Standards and Evaluations 3 (2023) 100092

Contents lists available at ScienceDirect

BenchCouncil Transactions on Benchmarks, Standards and Evaluations

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

Review article

Enabling hyperscale web services

Akshitha Sriraman

Carnegie Mellon University, United States of America

ARTICLE INFO

Keywords:
Hyperscale computing
Computer architecture
Software systems

ABSTRACT

Modern web services such as social media, online messaging, and web search support billions of users, requiring data centers that scale to hundreds of thousands of servers, i.e., *hyperscale*. The key challenge in enabling hyperscale web services arise from (1) an unprecedented growth in data, users, and service functionality and (2) a decline in hardware performance scaling. We highlight a dissertation's contributions in bridging the software and hardware worlds to realize more efficient hyperscale services despite these challenges.

Contents

1. Introduction

Modern web services such as social media, online messaging, web search, video streaming, and online banking often support billions of users, requiring data centers that scale to hundreds of thousands of servers, i.e., *hyperscale* [2]. In fact, the world continues to expect hyperscale computing to drive more futuristic, complex applications such as virtual reality, self-driving cars, conversational AI, and the Internet of Things. This survey paper highlights technologies detailed in the author's PhD dissertation [1] that will enable tomorrow's web services to meet the world's expectations.

The key challenge in enabling hyperscale web services arises from two important trends. First, over the past few years, there has been a radical shift in hyperscale computing due to an unprecedented growth in data [3], users [4], and service functionality [5]. Second, modern hardware can no longer support this growth in hyperscale trends due to a steady decline in hardware performance scaling [6]. To enable this new hyperscale era, hardware architects must become more aware of hyperscale software requirements and software researchers can no longer expect unlimited hardware performance scaling. In short, systems researchers can no longer follow the traditional approach of building each layer of the stack separately. Instead, they must rethink the synergy between the software and hardware worlds. The dissertation [1] creates such a synergy to enable future hyperscale web services.

The dissertation [1] bridges the software and hardware worlds, demonstrating the importance of that bridge in realizing efficient hyperscale web services via solutions that span the systems stack. The specific goal is to (1) design software that is aware of new hardware constraints and (2) architect hardware that efficiently supports new software requirements. To this end, the dissertation [1] spans two broad thrusts: (1) a software and (2) a hardware thrust to analyze the complex software and hardware hyperscale design space to develop efficient cross-stack solutions for hyperscale computation.

In the software thrust, the dissertation [1] contributes *μSuite*, the first open-source benchmark suite of modern web services built with a new hyperscale software paradigm [7]. Next, we[1] use *μSuite* to study software threading design implications in light of today's hardware reality and identify new insights in the age-old research area of software threading [8]. Driven by these insights, we demonstrate how software threading models must be redesigned at hyperscale by presenting an automated approach and tool, *μTune*, that makes intelligent threading decisions during system runtime [8].

In the hardware thrust, the dissertation [1] architects both commodity and custom hardware to efficiently support hyperscale software needs. First, we study the shortcomings in *commodity hardware* running hyperscale services, revealing insights that influenced commercial CPUs [9]. Based on these insights, we present a design tool, *SoftSKU*, that enables cheap commodity hardware to efficiently support new

E-mail address: akshitha@cmu.edu.

[1] When using the "we" pronoun, we refer to the author's dissertation's contributions [1].

https://doi.org/10.1016/j.tbench.2023.100092
Received 20 December 2022; Received in revised form 4 March 2023; Accepted 12 March 2023
Available online 1 April 2023

hyperscale software paradigms, improving the efficiency of real-world services that serve billions of users, saving millions of dollars, and meaningfully reducing the carbon footprint [9].

Next, the dissertation [1] studies how *custom hardware* must be designed at hyperscale, resulting in industry-academia benchmarking efforts, commercial hardware changes, and improved software development [2]. Based on this study's insights, the dissertation presents *Accelerometer*, an analytical model that estimates realistic gains from hardware customization [10].

2. Research goals and limitations of the state-of-the-art

Current software and hardware systems were conceived when we had scarce compute resources, limited data and users, and easy hardware performance scaling. These assumptions are not true today. Today, the world is undergoing a technological revolution where web services require *hyperscale* data centers to efficiently process requests from billions of users. These hyperscale services are facing an unprecedented growth in data [3], users [4], and functionality [5]. Unfortunately, hyperscale computing is emerging at a time when hardware is facing a steady decline in performance scaling [11].

Today, to enable web services, systems researchers typically follow the traditional approach of building each systems stack layer separately. As examples, in the application layer, to support the unprecedented growth in data, users, and functionality, there is shift towards a granular, modular application architecture, with services built with distributed application paradigms like microservices and serverless [12–16]. In the software layers, there is a shift towards light-weight abstractions (e.g., containers) [17–20] in place of heavy-weight ones (e.g., virtualization) [21,22]. In the hardware layer, due to the decline in hardware performance scaling, there is a shift towards building specialized hardware for various "killer" services [23–26].

To design efficient computing systems in light of modern hyperscale service trends and today's hardware reality, systems researchers can no longer afford to build each layer of the stack separately. In short, computer architects must now be aware of software requirements, and software developers can no longer expect continued hardware performance scaling. For example, in addition to the state-of-the-art trend of building custom hardware [23–25], architects must now build hardware that is aware of new service paradigms (e.g., microservices) and software trends (e.g., new threading models). Hence, *the dissertation's [1] first research goal is to rethink the synergy between the software and hardware worlds from the ground up*.

The main challenge in establishing synergy between software and hardware is a large and complex software and hardware design space that makes it intractable to manually identify optimal designs. For example, we discovered that the software threading design space has complex implications induced by the decline of hardware performance scaling, making it impractical for an expert software developer to manually identify the best threading design [8].

Manually navigating this vast and complex design space to make efficient design decisions is often intractable at hyperscale as (1) design implications vary across service loads, (2) trial-and-error methods or experience-based intuition do not systematically capture design space implications, (3) service code evolves quickly, (4) synthetic experiments do not capture production behavior, etc. Hence, to enable futuristic web services, we must achieve *the dissertation's [1] second research goal of automatically navigating, i.e., self-navigating, the complex software and hardware hyperscale design space*.

Given the widespread need for web services, to achieve both these research goals, it is critical to devise mechanisms that can automatically (1) bring new hardware insights when designing software stack layers and (2) draw on fundamental software design principles to systematically architect the hardware layer. Hence, *the dissertation [1] bridges the software and hardware worlds, demonstrating the importance of that bridge in enabling hyperscale web services via efficient self-navigating solutions that span the systems stack. Our vision is to (1) redesign web service software based on new overheads induced by the decline in hardware performance scaling and (2) rearchitect data center commodity and custom hardware to support new software requirements due to the unprecedented growth in data, users, and services.*

To achieve this research vision in a way that self-navigates the complex software and hardware design space, the dissertation [1] spans two thrusts: (1) a software and (2) a hardware thrust. In the software thrust, we ask: how do we design hyperscale web service software based on today's hardware overheads? In the hardware thrust, we ask: how do we architect data center commodity and custom hardware to support the unprecedented growth in hyperscale software trends? It is critical to systematically answer both questions to enable tomorrow's hyperscale web services.

3. Key research contributions

We detail the dissertation's [1] key contributions below.

Enabling the study of modern web services. Modern web services are increasingly built using microservice architectures, wherein a complex web service is composed of numerous distributed microservices such as HTTP connection termination, key–value serving [27], query rewriting [28], access-control management, and protocol routing [29]. Whereas monoliths face greater than 100 ms Service Level Objectives (SLOs) (e.g., ∼300 ms for web search [30]), microservices must often achieve sub-ms SLOs (e.g., ∼100 µs for protocol routing [31]), as many microservices must be invoked serially to serve a user's query. Hence, sub-ms–scale OS/network overheads (e.g., a context switch cost of 5–20 µs [32]) are often insignificant for monoliths. However, the microservice regime differs fundamentally: OS/network overheads (e.g., context switches, network protocol delays, inefficient thread wakeups, and lock contention) that are often minor with monolithic request service times of 100s of milliseconds, can dominate microservice latency distributions. For example, even a single 20 µs context switch implies a 20% latency penalty for a request to a 100 µs-response latency protocol routing microservice [31]. Hence, it is critical to revisit prior conclusions on sub-ms–scale OS/network overheads for the microservice regime [33].

Initially, there existed no representative, open-source benchmarks to study microservices. Widely-used academic data center benchmark suites [34,35], were unsuitable for characterizing sub-ms–scale overheads in microservices as they use monolithic rather than microservice architectures and largely have request service times that are greater than 100 ms. Hence, there was a real need for open-source benchmarks that enable the study of microservices.

To study microservices, as part of the dissertation's software contributions, it introduces the first open-source benchmark suite of end-to-end modern web services composed of microservices, called *µSuite* [7]. *µSuite* includes four end-to-end web services: a content-based high dimensional search for image similarity—`HDSearch`, a replication-based protocol router for scaling fault-tolerant key–value stores—`Router`, a service for performing set algebra on posting lists for document retrieval—`Set Algebra`, and a user-based item recommender system for predicting user ratings—`Recommend`. *µSuite* has been used by researchers in academia and industry (e.g., MIT, UIUC, UT Austin, Georgia Tech, Cornell, ARM, and Intel).

The dissertation uses *µSuite* to study the OS/network performance overheads incurred by microservices. This study reveals that threading interactions with the OS and network layers introduce microsecond-scale overheads that significantly affect microservices, but are insignificant to their monolithic counterparts. Hence, intelligent thread scheduling and better threading models can greatly improve microservice performance.

Redesigning software based on underlying data center hardware constraints. The dissertation's study of OS/network performance overheads using *µSuite* showed that microservices can benefit from better threading designs. These threading-induced overheads are due to today's hardware reality, where network devices have sped up while CPU

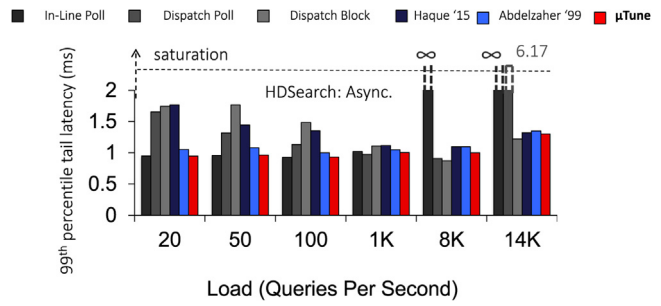

Fig. 1. *µTune*'s latency compared to existing techniques [37,38]: *µTune* lowers latency by 1.9×.

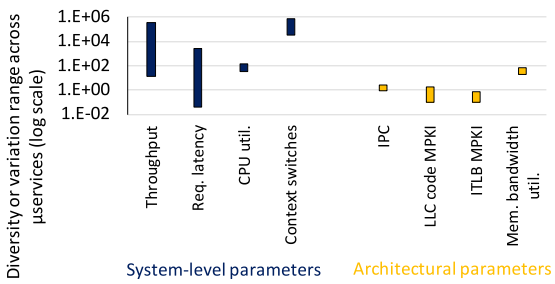

Fig. 2. Variation in system-level & architectural traits across microservices: Production microservices face diverse bottlenecks.

performance scaling has nearly stopped [36]. Today, a CPU thread's accesses to the underlying OS/network stacks cause threading-induced overheads that arise from thread contention on locks, thread wakeup delays, and context switching. Hence, analyzing software threading designs' implications and rethinking threading models for modern microservices has become a deeply important problem.

To study threading-induced software overheads that arise due to hardware constraints, there is a need to systematically analyze the sub-ms–scale OS and network overheads that arise from threading and concurrency design decisions. As part of the dissertation's software contributions [1], we use *µSuite* to systematically introduce and characterize a *taxonomy of threading models* [8]. This taxonomy is composed of software threading dimensions commonly used to build a microservice, such as synchronous or asynchronous RPCs, in-line or dispatched RPC handlers, and interrupt- or poll-based network reception. We also vary thread pool sizes dedicated to the various functionalities, i.e., network polling, RPC handling, and response execution. These threading design axes yield a rich space of microservice software threading architectures that interact with the underlying OS and hardware in starkly varied ways. Hence, this threading taxonomy and analysis enables expert and novice developers alike to guide their service threading designs.

The dissertation [1] makes the important observation that no single threading model is best across all load conditions, paving the way for an automatic load adaptation system that tunes threading models to improve performance. Specifically, our threading model study demonstrates that the relationship between optimal threading model and service load is complex—one could not expect a developer to pick the best threading model a priori. For example, at low load, models that poll for network traffic perform best, as they avoid thread wakeup delays. Conversely, at high load, models that separate network polling from RPC execution enable higher service capacity and blocking outperforms polling for incoming network traffic as it avoids wasting CPU on fruitless poll loops. Hence, exploiting these inherent threading model trade-offs during system runtime can significantly improve microservice latency.

To exploit threading trade-offs at runtime, the dissertation [1] presents and makes open source a system, *µTune* [8], that features a framework that builds upon open-source RPC platforms [39] to abstract threading model design from service code. *µTune*'s second feature is an intelligent run-time system that determines load via event-based monitoring and automatically adapts to time-varying service load by self-navigating the threading design space, i.e., tuning threading models and scaling thread pool sizes. As shown in Fig. 1, both features enable *µTune* to dynamically reduce microservice latency by 1.9× over static peak load-sustaining threading models (that an expert developer might have picked) and state-of-the-art adaptation techniques [37,38,40].

Architecting commodity hardware for new service paradigms. At global user population scale, key web services composed of numerous microservices can account for an enormous installed base of physical hardware. For example, across Facebook's global server fleet, seven key microservices in four service domains run at hyperscale, occupying a large portion of the fleet [9]. In light of this new microservice software paradigm, it is important to answer the question: do commodity server platforms serve microservices well? Are there common bottlenecks across microservices that we might address when designing future server architectures?

To identify whether commodity hardware efficiently supports microservices, the dissertation [1] undertakes comprehensive system-level and architectural analyses of Facebook's key production microservices serving live traffic. As shown in Fig. 2, we find that service functionality distribution across microservices has resulted in enormous diversity in system (e.g., request latency and CPU utilization) and architectural requirements (e.g., Instructions Per Cycle and LLC code misses per kilo instruction), with new CPU bottlenecks (e.g., high I/O processing latency and I-cache misses). Our identified bottlenecks made hardware vendors reconsider the benchmarks they used for decades to evaluate new servers.

As examples, we find that caching microservices [41] require intensive I/O and microsecond-scale response latency and frequent OS context switches constitute 18% of CPU time. In contrast, a `Feed` [42] microservice computes for seconds per request with minimal OS interaction. Facebook's `Web` [43] microservice exhibits massive instruction footprints, leading to astonishing I-cache misses and branch mispredictions, while other microservices exhibit smaller code footprints. Some microservices depend heavily on floating-point performance while others have no floating-point instructions.

The great diversity in hardware bottlenecks across microservices might suggest a strategy to specialize CPU architectures to suit each microservice's distinct needs. However, hyperscale enterprises have strong economic incentives to limit hardware platforms' diversity to (1) maintain fungibility of hardware resources, (2) preserve procurement advantages that arise from economies of scale, and (3) limit the overhead of qualifying/testing myriad hardware platforms. As such, there is an immediate need for strategies that extract greater performance from existing commodity server architectures to efficiently support diverse microservices on commodity hardware.

As part of the dissertation's hardware contributions, it introduces an automated approach and tool to improve hyperscale microservice performance on cheap commodity server architectures (often called "SKUs", short for "Stock Keeping Units") [9]. This approach called *SoftSKU* is a design-time strategy that tunes coarse-grain (e.g., boot time) OS and hardware configuration knobs available on commodity processors to help a processor platform or SKU better support its assigned microservice. OS and CPUs provide several specialization knobs; we focus on seven: (1) core frequency, (2) uncore frequency, (3) active core count, (4) code vs. data prioritization in the last-level cache ways, (5) hardware prefetcher configuration, (6) use of transparent huge pages, and (7) use of statically-allocated huge pages. The dissertation also proposes new CPU knobs (e.g., Branch Target Buffer ways) that can be made configurable to create finer-grained soft SKUs.

Manually identifying a microservice-specific *SoftSKU* is impractical as the design space is large, code evolves quickly, synthetic load tests do not often capture production behavior, and the effects of tuning a single

knob are often small. Hence, we build an automated design tool—μSKU—that self-navigates the hardware configuration design space to optimize a hardware SKU for each microservice. μSKU automatically varies configurable server knobs, by searching within a predefined design space via A/B testing, where it compares the performance of two identical servers that differ only in their knob configuration. μSKU collects copious fine-grain performance measurements while conducting automated A/B tests on production systems serving live traffic to search for statistically significant performance gains. We evaluate μSKU on hyperscale production microservices and show that the ensuing soft SKUs outperform stock and production server configurations by up to 7.2% and 4.5% respectively, with no additional hardware requirement [9].

SoftSKU demonstrates that before resorting to hardware customization, there is still significant performance to be extracted from cheap commodity CPUs by tuning their OS and hardware knobs. In this manner, soft SKUs significantly improve the performance efficiency of real-world Facebook microservices that serve billions of users, saving millions of dollars and meaningfully reduce the global carbon footprint [44]. Since this work [9], several hyperscale enterprises have dedicated teams of engineers to explore additional configurable hardware/OS soft-SKU knobs (e.g., SIMD width).

Architecting custom hardware for new service paradigms. The *SoftSKU* work [9] revealed that microservices are so diverse that they could benefit from custom hardware. In fact, to improve hardware efficiency, several architects today work on developing numerous specialized hardware accelerators for important microservice domains (e.g. Machine Learning tasks). Designing such custom hardware accelerators for each microservice operation might improve performance or energy. However, designing custom hardware for each microservice operation is prohibitively expensive at hyperscale since data center operators lose procurement advantages that arise from economies of scale and must also develop and test on myriad custom hardware platforms. Hence, an important question arises: *which microservice software operations consume the most CPU cycles and are worth accelerating in the hardware?*

To build specialized accelerators for these key microservice operations, it is important to first systematically identify which type of accelerator meets microservice requirements and is worth designing and deploying. Deploying specialized hardware is risky at hyperscale, as the hardware might under-perform due to performance bounds from the microservice's software interaction with the hardware, resulting in high monetary losses. To make well-informed hardware decisions, it is crucial to systematically answer the following question early in the design phase of a new accelerator to determine whether the new accelerator is worth designing: *how much can the accelerator realistically improve its targeted microservice overhead?*

To answer the first question posed above, we undertake a comprehensive study of how microservices spend their CPU cycles (as part of the dissertation's hardware contributions). In Fig. 3, we study seven key hyperscale Facebook microservices in four diverse service domains that run across hundreds of thousands of servers, occupying a large portion of the global server fleet. Our study reveals that microservices spend only a small fraction of CPU cycles executing their main application functionality (e.g., a Machine Learning task); the remaining cycles are spent in common *orchestration overheads*, i.e., operations that are not critical to the main microservice functionality (e.g., I/O notification, logging, and compression). Accelerating such common building blocks can greatly improve performance. Already, a few hardware vendors have used this study's insights to influence hardware customization for orchestration overheads [2] (e.g., this study's insights brought about the Intel's Infrastructure Processing Unit [45]).

Our characterization drove a hardware vendor to consider more representative benchmarks (in place of traditional ones used for decades) when evaluating hardware designs [2]. This study resulted in an industry-academia joint collaborative effort to design and open-source representative data center benchmarks. Additionally, our characterization tool has been integrated into Facebook's fleet-wide performance

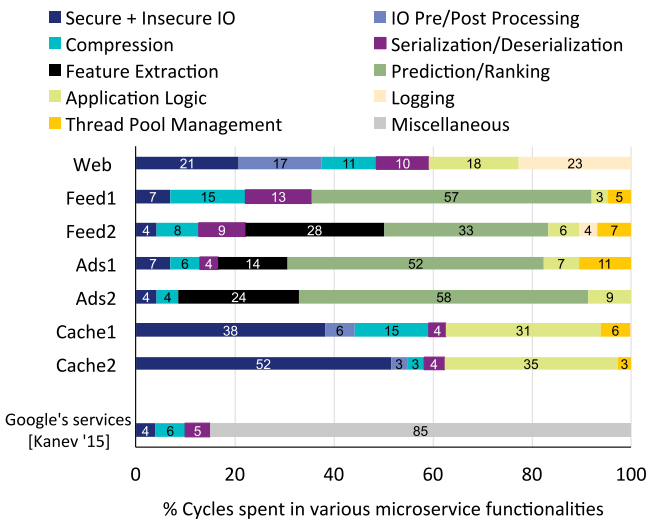

Fig. 3. Breakdown of cycles spent in Facebook production service operations: Orchestration overheads are significant and common.

monitoring infrastructure; it curates statistics from hundreds of thousands of servers to help developers visualize the performance impact of their code changes at hyperscale [2].

To answer the second question posed above, we develop *Accelerometer*,[2] an analytical model for hardware acceleration [10]. *Accelerometer* estimates realistic gains from hardware acceleration by self-navigating the various performance bounds that arise from a microservice's software interactions with the hardware. *Accelerometer* identifies performance bounds and design bottlenecks early in the hardware design cycle, and provides insight into which hardware acceleration strategies may alleviate these bottlenecks.

Accelerometer models both synchronous and asynchronous microservice software interactions for three hardware acceleration strategies—on-chip, off-chip, and remote. It assumes an abstract system with (1) a *host*: a general-purpose CPU, (2) an *accelerator*: custom hardware to accelerate a kernel, and (3) an *interface*: the communication layer between the host and the accelerator (e.g., a PCIe link). It models the microservice throughput speedup and the per-request latency reduction. We validate *Accelerometer*'s utility via three retrospective case studies conducted on production systems, by comparing model-estimated speedup with real service speedup—*Accelerometer* estimates the real microservice speedup with an error that is $\leq$3.7%. We also use *Accelerometer* to project speedup with new accelerators.

As services evolve, Accelerometer's generality makes it more suitable in determining new hardware requirements early in the design phase. Since we validated *Accelerometer* in production and made it open-source, it has been adopted by many hyperscale enterprises (e.g., with developing encryption/compression accelerators) to make well-informed hardware decisions [2].

Overall, the dissertation's primary, unique contribution is bridging the software and hardware worlds and demonstrating the importance of that bridge in realizing efficient hyperscale services via cross-stack solutions. Specifically, through the software and hardware contributions below, we realize efficient services from analytical models on paper to deployment at hyperscale.

- Software contributions.
  - The dissertation is the first to present an open-source benchmark suite of microservices that facilitates future academic and industry research [7].

[2] *Accelerometer* was recognized for its long-term impact potential with an IEEE Micro Top Picks distinction (one of top 12 computer architecture papers in 2020) [2].

- The dissertation identifies new insights in the age-old research area of software threading models that led to redesigning threading models for hyperscale web services [8].

• Hardware contributions.

- The dissertation analyzes shortcomings in commodity hardware running hyperscale services that influenced the design of commercial CPUs [9].
- The dissertation demonstrates how commodity hardware can be used efficiently to enable hyperscale services that led to real-world data centers prioritizing this approach over today's hardware customization trend [9].
- The dissertation presents a systematic understanding of hardware customization opportunities at hyperscale that enabled industry-academia joint benchmarking efforts, influenced commercial hardware design, and improved software development [2,10].
- The dissertation presents a rigorous, analytical alternative to ad hoc hardware customization approaches that enabled real-world hyperscale data centers to make well-informed hardware investments [2, 10].

4. Future directions

There are many exciting avenues of future work that follow from the research presented in the dissertation; some of these are summarized below.

Enabling cross-stack designs for emerging service paradigms. Apart from the microservice paradigm studied in the dissertation [1], modern web systems are being built with newer service paradigms such as serverless. Each new paradigm introduces unique overheads that affect hyperscale efficiency. For example, unlike microservices, serverless systems introduce new inefficiencies from container launch and warm-up delays, increased communication, and greater scalability issues. Techniques developed in the dissertation can help future systems support emerging service paradigms.

Rethinking hardware–software co-design for hyperscale overheads. The dissertation's study of real-world microservices revealed several system overheads that particularly arise at hyperscale. The dissertation's work reduced a few predominant overheads such as I-cache misses and I/O event notification. Going forward, there is a need to optimize other overheads identified in the dissertation. For example, apart from improving I/O event notification, we must optimize the end-to-end I/O processing path to efficiently (1) receive/send a large number of I/O, (2) operate the CPU when awaiting IO and (3) process large I/O just as well as small I/O transfers.

Mitigating the killer microsecond problem in modern services. As the dissertation [1] shows, modern servers have mechanisms to effectively hide nanosecond-scale stalls (e.g., OoO cores) and millisecond-scale stalls (e.g., context switching), but lack efficient support to hide microsecond-scale stalls that critically affect modern services. To mitigate microsecond-scale stalls (often called the "killer microsecond" [46]), we must study various microsecond-scale accesses' (e.g., modern networking, non-volatile memories, and accelerator accesses') impact on efficiency, to develop cross-stack solutions. For example, we must build "microsecond-aware" stacks with reduced lock contention, fast interrupts, efficient spin-polling, and better scheduling.

Using machine learning to self-navigate the hyperscale design space. As the hyperscale software/hardware design space continues to become more complex, we foresee empirical systems leveraging recent improvements in ML models to manage design complexity, to use ML techniques to self-navigate complex software/hardware design spaces such as resource allocation, request scheduling, and bottleneck identification.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] Enabling Hyperscale Web Services (Ph.D. thesis).
[2] A. Sriraman, A. Dhanotia, Understanding acceleration opportunities at hyperscale, IEEE Micro (2021).
[3] What's causing the exponential growth of data? https://insights.nikkoam.com/articles/2019/12/whats_causing_the_exponential.
[4] Digital 2020: 3.8 billion people use social media. https://wearesocial.com/blog/2020/01/digital-2020-3-8-billion-people-use-social-media.
[5] The Top 12 future web development trends in 2021. https://dev.to/adhyaswarnali/the-top-12-future-web-development-trends-in-2021-25k5.
[6] G.E. Moore, Cramming More Components onto Integrated Circuits, McGraw-Hill, New York, 1965.
[7] A. Sriraman, T.F. Wenisch, μSuite: A benchmark suite for microservices, in: IEEE International Symposium on Workload Characterization, 2018.
[8] A. Sriraman, T.F. Wenisch, μTune: Auto-tuned threading for OLDI microservices, in: USENIX Conference on Operating Systems Design and Implementation, 2018.
[9] A. Sriraman, A. Dhanotia, T.F. Wenisch, SoftSKU: Optimizing server architectures for microservice diversity @scale, in: The International Symposium on Computer Architecture, 2019.
[10] A. Sriraman, A. Dhanotia, Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale, in: International Conference on Architectural Support for Programming Languages and Operating Systems, 2020.
[11] M.M. Waldrop, The chips are down for Moore's law, Nat. News 530 (7589) (2016) 144.
[12] A brief history of microservices. https://www.dataversity.net/a-brief-history-of-microservices/.
[13] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, C. Delimitrou, An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems, in: International Conference on Architectural Support for Programming Languages and Operating Systems, 2019.
[14] S. Kanev, K. Hazelwood, G.-Y. Wei, D. Brooks, Tradeoffs between power management and tail latency in warehouse-scale applications, in: IEEE International Symposium on Workload Characterization, 2014.
[15] N. Dmitry, S.-S. Manfred, On micro-services architecture, Int. J. Open Inf. Technol. (2014).
[16] I. Nadareishvili, R. Mitra, M. McLarty, M. Amundsen, Microservice architecture: Aligning principles, practices, and culture, 2016.
[17] About DPDK. https://www.dpdk.org/about/.
[18] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, E. Bugnion, IX: A protected dataplane operating system for high throughput and low latency, in: USENIX Conference on Operating Systems Design and Implementation, 2014.
[19] M. Marty, M. de Kruijf, J. Adriaens, C. Alfeld, S. Bauer, C. Contavalli, M. Dalton, N. Dukkipati, W.C. Evans, S. Gribble, N. Kidd, R. Kononov, G. Kumar, C. Mauer, E. Musick, L. Olson, E. Rubow, M. Ryan, K. Springborn, P. Turner, V. Valancius, X. Wang, A. Vahdat, Snap: A microkernel approach to host networking, in: ACM Symposium on Operating Systems Principles, 2019.
[20] Key components of a software defined data center. https://www.evolvingsol.com/2018/04/17/components-software-defined-data-center/.
[21] Dawn of the data center operating system. https://www.infoworld.com/article/2906362/dawn-of-the-data-center-operating-system.html.
[22] Containers. https://a16z.com/2015/01/22/containers/.
[23] A. Putnam, A.M. Caulfield, E.S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G.P. Gopal, J. Gray, M. Haselman, S. Hauck, S. Heil, A. Hormati, J.-Y. Kim, S. Lanka, J. Larus, E. Peterson, S. Pope, A. Smith, J. Thong, P.Y. Xiao, D. Burger, A reconfigurable fabric for accelerating large-scale datacenter services, in: International Symposium on Computer Architecuture, 2014.
[24] N.P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T.V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C.R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, D.H. Yoon, In-datacenter performance analysis of a tensor processing unit, in: International Symposium on Computer Architecture, 2017.

[25] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K.S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. El Husseini, T. Juhasz, K. Kagi, R.K. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Yi Xiao, D. Zhang, R. Zhao, D. Burger, Serving DNNs in real time at datacenter scale with project brainwave, IEEE Micro 38 (2) (2018) 8–20.

[26] B. Abali, B. Blaner, J. Reilly, M. Klein, A. Mishra, C.B. Agricola, B. Sendir, A. Buyuktosunoglu, C. Jacobi, W.J. Starke, H. Myneni, C. Wang, Data compression accelerator on IBM POWER9 and Z15 processors, in: International Symposium on Computer Architecture, 2020.

[27] B. Fitzpatrick, Distributed caching with memcached, Linux J. (2004).

[28] M. Barhamgi, D. Benslimane, B. Medjahed, A query rewriting approach for web service composition, IEEE Trans. Serv. Comput. (2010).

[29] Mcrouter. https://github.com/facebook/mcrouter.

[30] B. Vamanan, J. Hasan, T. Vijaykumar, Deadline-aware datacenter TCP (D2TCP), in: ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, 2012.

[31] Y. Zhang, D. Meisner, J. Mars, L. Tang, Treadmill: Attributing the source of tail latency through precise load testing and statistical inference, in: International Symposium on Computer Architecture, 2016.

[32] D. Tsafrir, The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops), in: Workshop on Experimental Computer Science, 2007.

[33] L. Barroso, M. Marty, D. Patterson, P. Ranganathan, Attack of the killer microseconds, Commun. ACM (2017).

[34] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A.D. Popescu, A. Ailamaki, B. Falsafi, Clearing the clouds: A study of emerging scale-out Workloads on modern hardware, in: International Conference on Architectural Support for Programming Languages and Operating Systems, 2012.

[35] PerfKit benchmarker. https://github.com/GoogleCloudPlatform/PerfKitBenchmarker.

[36] A. Danowitz, K. Kelley, J. Mao, J.P. Stevenson, M. Horowitz, CPU DB: Recording microprocessor history, Commun. ACM (2012).

[37] M.E. Haque, Y.h. Eom, Y. He, S. Elnikety, R. Bianchini, K.S. McKinley, Few-to-many: Incremental parallelism for reducing tail latency in interactive services, in: International Conference on Architectural Support for Programming Languages and Operating Systems, 2015.

[38] T.F. Abdelzaher, N. Bhatti, Web server QoS management by adaptive content delivery, in: International Workshop on Quality of Service, 1999.

[39] gRPC. https://github.com/heathermiller/dist-prog-book/blob/master/chapter/1/gRPC.md.

[40] K. Langendoen, J. Romein, R. Bhoedjang, H. Bal, Integrating polling, interrupts, and thread management, in: Symposium on the Frontiers of Massively Parallel Computing, 1996.

[41] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H.C. Li, et al., TAO: Facebook's distributed data store for the social graph, in: USENIX Annual Technical Conference, 2013.

[42] M. Zuckerberg, R. Sanghvi, A. Bosworth, C. Cox, A. Sittig, C. Hughes, K. Geminder, D. Corson, Dynamically providing a news feed about a user of a social network. https://patents.google.com/patent/US7669123B2/en.

[43] G. Ottoni, HHVM JIT: A profile-guided, region-based compiler for PHP and hack, in: Conference on Programming Language Design and Implementation, 2018.

[44] Accelerometer & SoftSKU: Improving HW performance for diverse microservices. https://engineering.fb.com/data-center-engineering/accelerometer-and-softsku/.

[45] P. Kummrow, The IPU: A new, strategic resource for cloud service providers, 2021, https://itpeernetwork.intel.com/ipu-cloud/. [Online; accessed 22-August-2021].

[46] A. Mirhosseini, A. Sriraman, T.F. Wenisch, Enhancing server efficiency in the face of killer microseconds, in: International Symposium on High Performance Computer Architecture, 2019.