Understanding Acceleration Opportunities at Hyperscale

Akshitha Sriraman, *University of Michigan, Ann Arbor, MI, 48109, USA*
Abhishek Dhanotia, *Facebook, Menlo Park, CA, 94025, USA*

Modern web services run across hundreds of thousands of servers in a data center, i.e., at hyperscale. With the end of Moore's Law and Dennard scaling, successive server generations running these web services exhibit diminishing performance returns, resulting in architects adopting hardware customization. An important question arises: Which web service software operations are worth building custom hardware for? To answer this question, we comprehensively analyze important Facebook production services and identify key acceleration opportunities. We then develop an open-source analytical model, Accelerometer, to help make well-informed hardware decisions for the acceleration opportunities we identify.

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*. Whereas hyperscale web services once had largely monolithic software architectures, modern web services are composed of numerous independent, specialized, distributed microservices (e.g., key-value serving in a social media service).[1,2] Several companies such as Amazon, Netflix, Gilt, LinkedIn, Facebook, and SoundCloud have adopted microservice architectures to improve web service development and scalability.[3]

While at face value, hyperscale web systems seem instantaneously available at the touch of a button, existing microservices barely meet performance requirements. In reality, microservices have much more stringent performance constraints than their monolithic counterparts, since numerous microservices must be invoked, often serially, to serve a user's query. For example, a Facebook news feed service query may flow through a pipeline of many microservices invoked via remote procedure calls (RPCs), such as 1) Sigma: a spam filter; 2) McRouter: a protocol router; 3) Feed: a news feed stories extractor; 4) Tao: a distributed social graph data store; and 5) MyRocks: a user database. Complex microservice interactions place stringent performance constraints on individual microservices.

As hyperscale computing grows to drive more sophisticated applications (e.g., virtual reality and conversational AI), existing microservice systems will face greater efficiency challenges due to these more complex tasks. Increasingly complex microservices can be efficiently supported if the hardware rises to meet efficiency requirements. However, with the end of Moore's Law and Dennard scaling, a key challenge to realizing microservice efficiency is that successive server generations running microservices exhibit diminishing performance returns.

To improve hardware efficiency, several architects today work on developing numerous specialized hardware accelerators for important microservice domains [e.g. machine learning (ML) tasks]. However, large-scale internet operators have strong economic incentives to limit hardware platform diversity to: 1) maintain fungibility of hardware resources; 2) preserve procurement advantages that arise from economies of scale; and 3) limit the overhead of developing and testing on myriad specialized hardware platforms. Hence, an important question arises: Which microservice operations consume the most CPU cycles and are worth accelerating?

To build specialized accelerators for these key microservice operations, it is important to first identify which type of accelerator meets microservice requirements and is worth designing and

Digital Object Identifier 10.1109/MM.2021.3066615
Date of publication 30 March 2021; date of current version 25 May 2021.

deploying. Deploying specialized hardware is risky at hyperscale, as the hardware might underperform 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 answer the following question early in the design phase of a new accelerator: How much can the accelerator realistically improve its targeted microservice overhead?

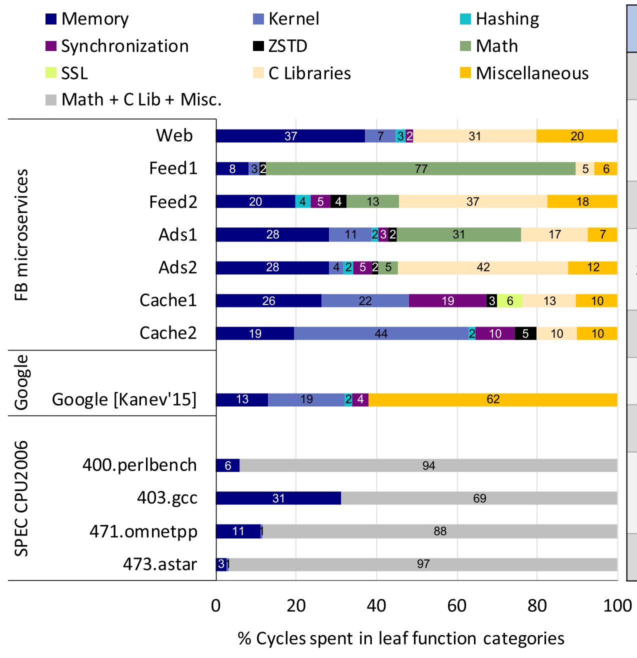
To answer both the above questions, our article,[4] presented at ASPLOS 2020, first undertakes a comprehensive characterization of how microservices spend their CPU cycles. We study seven important hyperscale Facebook microservices in four diverse service domains that run across hundreds of thousands of servers, occupying a large portion of Facebook's global server fleet. Our detailed breakdown of CPU cycles consumed by various microservice operations identifies key overheads and potential design optimizations. To make well-informed hardware decisions for these microservice overheads, we contribute *Accelerometer*, an analytical model that projects realistic microservice speedup for various hardware acceleration strategies. We also demonstrate *Accelerometer*'s utility in Facebook's production microservices via three retrospective case studies conducted when serving live user traffic.

HOW DO MICROSERVICES SPEND THEIR CPU CYCLES?

To identify microservice overheads, we comprehensively characterize how seven important Facebook production microservices spend their CPU cycles when serving live user traffic. Very few prior works study how cycles are spent in data centers. Kanev *et al.*[1] investigate the "data center tax" across Google's server fleet by studying cycles spent in seven types of *leaf functions* invoked at the end of a call trace [e.g., `memcpy()`]. However, a leaf function study alone does not holistically provide insight into whether acceleration might improve a *microservice functionality* (e.g., encryption).

To analyze *microservice functionalities*, we must comprehensively characterize a microservice's entire call stack to measure the CPU cycles spent in each phase of the microservice's operation after it receives a request. Characterizing microservice functionalities helps determine 1) whether diverse microservices execute common types of operations (e.g., compression, serialization, and encryption) and 2) the overheads such operations induce. Analyzing *both* leaf functions and microservice functionalities helps identify key acceleration opportunities that might inform future software and hardware designs.

We characterize the CPU cycles spent by Facebook's production microservices in both *leaf functions*



Leaf category	Examples of leaf functions
Memory	Memory copy, allocation, free, compare
Kernel	Task scheduling, interrupt handling, network communication, memory management
Hashing	SHA & other hash algorithms
Synchronization	User-space C++ atomics, mutex, spin locks, CAS
ZSTD	Compression, decompression
Math	Intel's MKL, AVX
SSL	Encryption, decryption
C Libraries	C/C++ search algorithms, array & string compute
Miscellaneous	Other assorted function types

Categorization of leaf functions

FIGURE 1. Breakdown of cycles spent in various leaf functions (leaf categories defined in the table to the right): Memory functions consume a significant portion of total cycles.

and *microservice functionalities*. We study 1) Web: a front-end microservice that implements PHP and Hack; 2) Feed1 and Feed2: news feed microservices that aggregate, rank, and display stories; 3) Ads1 and Ads2: advertisement microservices that compute user-specific and ad-specific data; and 4) Cache1 and Cache2: large distributed-memory object caching microservices.

Leaf Function Characterization

We present key leaf function breakdowns for Facebook's microservices in Figure 1, comparing them with Google's services[1] and SPEC CPU2006 benchmarks.[5] We find that many leaf function overheads are significant and common across microservices. We detail our observations for dominant leaf categories.

Memory. Most microservices spend a significant fraction of cycles on memory functions that include memory copy, free, allocation, move, set, and compare. Memory copies are by far the greatest consumers of memory cycles. Data is primarily copied during microservice operations such as 1) I/O pre- or postprocessing, 2) I/O sends and receives, 3) RPC serialization/deserialization, and 4) microservice business logic execution (e.g., executing key-value stores in Cache).

We observe significant diversity in dominant service functionalities that invoke memory copies across microservices. This diversity suggests a strategy to specialize copy optimizations to suit each microservice's distinct needs. For example, Web can benefit from reducing copies in I/O pre- or postprocessing, whereas Cache2 can gain from fewer copies in network protocol stacks.

Freeing memory incurs a high overhead for several microservices, as the free() function does not take a memory block size parameter, performing extra work to determine the size class to return the block to. TCMalloc performs a hash lookup to get the size class. This hash tends to cache poorly, especially in the TLB, leading to performance losses. Although C++11 ameliorates this problem by allowing compilers to invoke delete() with a parameter for memory block size, overheads still arise from 1) removing pages faulted in when memory was written to and 2) merging neighboring freed blocks to produce a more valuable large free block. While numerous prior works optimize memory allocations,[6] very few recognize that optimizing free() can result in significant performance wins.

Kernel. Microservices with high OS kernel overhead, Cache1 and Cache2, invoke OS scheduler functions frequently. Software/hardware optimizations that reduce scheduler latency (e.g., intelligent thread switching[7] and coalescing I/O) might considerably improve their performance. Cache2 also spends significant cycles in I/O and network interactions, and can benefit from optimizations such as kernel-bypass and multiqueue NICs.

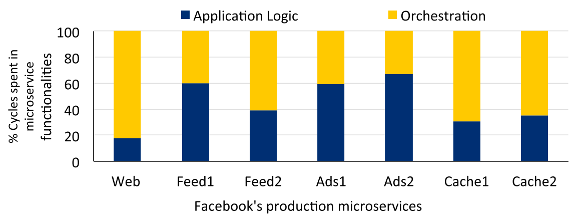


FIGURE 2. Breakdown of cycles spent in the main application logic versus orchestration work: Orchestration overheads significantly dominate.

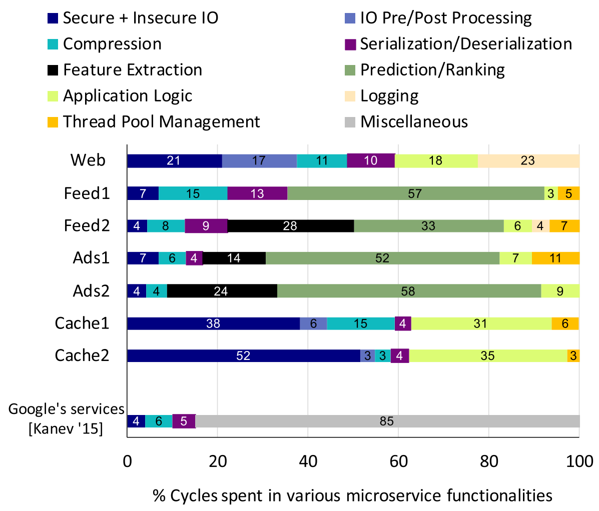
Synchronization. Microservices such as Cache oversubscribe threads to improve throughput.[8] Hence, such microservices spend significant cycles synchronizing frequent communication between distinct thread pools. Cache also spends a large fraction of cycles in spin locks that are typically deemed performance inefficient.[9] However, Cache implements spin locks since it is a microsecond-scale microservice,[8] and is hence more prone to microsecond-scale performance penalties that can otherwise arise from thread re-scheduling, wakeups, and context switches.[7]

C Libraries. We observe that Feed2, Ads1, and Ads2 invoke C libraries for vector operations, as they deal with large ML feature vectors. Web spends significant cycles parsing and transforming strings to process queries from many URL endpoints. Interestingly, unlike memory or kernel leaf functions, C libraries' instructions per cycle (IPC) scales well across CPU generations, as many hardware vendors primarily rely on open-source SPEC benchmarks that heavily use C libraries to make architecture design decisions.

Other observations. ML microservices such as Ads2 and Feed2 spend only up to 13% of cycles on mathematical operations that constitute ML inference using multilayer perceptrons. Cache2 spends 6% of cycles in leaf encryption functions since it encrypts a high number of queries per second (QPS). Additionally, Google's breakdown for a few leaf function categories, such as memory or kernel, is similar to Facebook's breakdowns. In contrast, SPEC CPU2006 benchmarks do not capture key leaf overheads faced by our microservices.

Service Functionality Characterization

We show a broad microservice functionality breakdown in Figure 2.



Functionality category	Examples of service operations
Secure & insecure I/O	Encrypted/plain-text I/O sends & receives
I/O pre/post processing	Allocations, copies, etc. before/after I/O
Compression	Compression/decompression logic
Serialization	RPC serialization/deserialization
Feature extraction	Feature vector creation in ML services
Prediction/ranking	ML inference algorithms
Application logic	Main business logic (e.g., Cache's key-value serving)
Logging	Creating, reading, updating logs
Thread pool management	Creating, deleting, synchronizing threads

Categorization of microservice functionalities

FIGURE 3. Breakdown of CPU cycles spent in various microservice functionalities (service functionality categories defined in the table to the right): Orchestration overheads are significant and fairly common across microservices.

We find that application logic disaggregation across microservices has resulted in significant microservice functionality overheads. Several microservices spend only a small fraction of their execution time serving their main application logic (e.g., ML-based ads recommendation or key-value serving), squandering significant cycles facilitating the main logic via *orchestration* work that is not critical to the main application logic (e.g., compression, serialization, and I/O processing). For example, microservices that perform ML inference—Feed1, Feed2, Ads1, and Ads2—spend as few as 33% of cycles on ML inference, consuming 42%–67% of cycles in orchestrating inference. Hence, even if modern inference accelerators[10] were to offer an infinite inference speedup, the net microservice performance would only improve by 1.49x–2.38x. There is hence an urgent need to accelerate the significant orchestration work that facilitates the main application logic.

Orchestration overheads arise since a microservice, upon receiving an RPC, must often perform operations such as I/O processing, decompression, deserialization, and decryption, before executing its main functionality. Hence, many microservices face common orchestration overheads despite great diversity in microservices' main application logic, as shown in Figure 3.

We make several observations about these significant and common orchestration overheads. First, Web, Cache1, and Cache2 spend a large portion of cycles executing I/O, i.e., sending and receiving RPCs, and consequent I/O compression and serialization overheads dominate. Web incurs a high I/O overhead since it implements many URL endpoints and communicates with a large back-end microservice pool. Cache1 and Cache2 are leaf microservices that support a high request rate[8]—they frequently invoke RPCs to communicate with mid-tier microservices. These microservices can benefit from I/O optimizations such as kernel-bypass, multiqueue NICs, and efficient I/O notification paradigms.

Second, Web spends only 18% of cycles in its main web serving logic (parsing and processing client requests), consuming 23% of cycles in reading and updating logs. It is unusual for applications to incur such high logging overheads; only few academic studies focus on optimizing them in hardware.

Third, Ads1, Feed1, Feed2, and Cache1 incur a high thread pool management overhead. Intelligent thread scheduling and tuning[7] can help these microservices.

We conclude application logic disaggregation across microservices and the consequent increase in inter-service communication at hyperscale has resulted in significant and common orchestration overheads in modern data centers. In Table 1, we report acceleration opportunities that might inform future software and hardware designs. We believe that our rich overhead characterization and taxonomy of existing optimizations will guide researchers in mitigating these overheads.

ACCELEROMETER: AN ANALYTICAL MODEL FOR HARDWARE ACCELERATION

Accelerating key common orchestration overheads in production requires 1) designing new hardware; 2) testing it; and 3) carefully planning capacity to provision the hardware to match projected load. Given the

TABLE 1. Summary of findings and suggestions for future optimizations.

Finding	Acceleration opportunity
Significant orchestration overheads	Software and hardware acceleration for orchestration rather than just app. logic
Several common orchestration overheads	Accelerating common overheads (e.g., compression) can provide fleet-wide wins
Poor IPC scaling for several functions	Optimizations for specific leaf/service categories
Memory copies & allocations are significant	Dense copies via SIMD, copying in DRAM, Intel's I/O AT, DMA via accelerators, PIM
Memory frees are computationally expensive	Faster software libraries, hardware support to remove pages
High kernel overhead and low IPC	Coalesce I/O, user-space drivers, in-line accelerators, kernel-bypass
Logging overheads can dominate	Optimizations to reduce log size or number of updates
High compression overhead	Bit-plane compression, Buddy compression, dedicated compression hardware
Cache synchronizes frequently	Better thread pool tuning and scheduling, Intel's TSX, coalesce I/O, vDSO
High event notification overhead	Hardware support for notifications (e.g., RDMA-style), spin versus block hybrids

uncertainties inherent in projecting customer demand, deploying diverse custom hardware is risky at scale as the hardware might underperform due to performance bounds from the microservice's software interactions with the hardware.

TO EASILY AND ACCURATELY MODEL WHETHER AN ACCELERATOR IS WORTH DESIGNING AND DEPLOYING FOR A MICROSERVICE OPERATION, WE DEVELOP AN ANALYTICAL MODEL, ACCELEROMETER.

To easily identify performance bounds early in the hardware design phase and estimate realistic gains from hardware acceleration, there is an urgent need to develop a simple, yet, accurate analytical model for hardware acceleration. The state-of-the-art analytical model for acceleration, LogCA,[11] falls short for microservices as it assumes that the CPU synchronously waits while the offload operates. However, for many microservice functionalities, offload is asynchronous; the processor continues doing useful work concurrent with the offload. Capturing this concurrency-induced performance bounds will help realistically model microservice speedup for various hardware acceleration strategies.

To easily and accurately model whether an accelerator is worth designing and deploying for a microservice operation, we develop an analytical model, *Accelerometer*. *Accelerometer* models both synchronous and asynchronous offloads for three hardware acceleration strategies—on-chip, off-chip, and remote.

Accelerometer assumes an abstract system with three components: 1) *host:* a general-purpose CPU; 2) *accelerator:* custom hardware to accelerate a kernel (or microservice operation); and 3) *interface:* the communication layer between the host and the accelerator (e.g., a PCIe link). *Accelerometer* models both the microservice throughput speedup (referred to as "speedup") and the per-request latency speedup (referred to as "latency reduction"). Modeling both speedup and latency reduction ensures that acceleration enables a higher throughput (i.e., more QPS) without violating microservice latency service level objectives (SLOs). When work is offloaded to an accelerator, the speedup and latency reduction depend on the acceleration strategy and the threading design used to offload, i.e., synchronous versus asynchronous offload.

Synchronous Offload

In a synchronous offload (Sync), the host awaits the accelerator's response before resuming execution

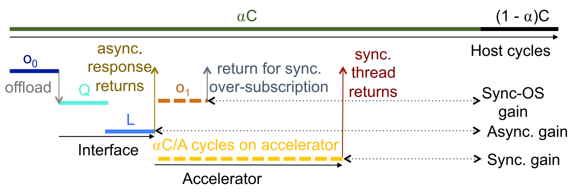


FIGURE 4. Example timeline of host and accelerator.

(see Figure 4), putting the accelerator's operation cycles ($\frac{\alpha * C}{A}$) in the critical path of the host's execution, impacting speedup and per-request latency. The host can consume additional cycles to 1) prepare the kernel for offload, o_0; 2) transfer the kernel to the accelerator, L; and 3) wait in a queue for the accelerator to become available, Q.

Synchronous Offload With Thread Oversubscription

In reality, several microservices (e.g., `Web` and `Cache`) oversubscribe threads to improve throughput by having more threads than available cores. With synchronous offload, a microservice oversubscribing threads (`Sync-OS`) allows the host to schedule an available thread to process new work, while the thread that offloaded work `blocks` awaiting the accelerator's response. Hence, the accelerator's cycles do not critically affect speedup, but impact the per-request latency. Moreover, OS overheads from the host switching to an available thread after an offload, o_1, affect both speedup and latency reduction. The microsecond-scale o_1 overhead dominates in microsecond-scale microservices (e.g., Caching), making it feasible to incur a throughput gain at the cost of a per-request latency slowdown. In such cases, *Accelerometer* can help ensure that the microservice still meets its latency SLO.

Asynchronous Offload

In an asynchronous offload (`Async`), the host does useful work concurrent with the accelerator's operation on the offload, removing the accelerator's cycles from the critical path. Depending on whether the response is picked up by the same thread that sent the request or a different thread, OS thread switch penalties can impact speedup and latency reduction.

Accelerometer models all these cases as well as other nuanced scenarios to project realistic gains early in the hardware design phase and make well-informed hardware investments. We expect *Accelerometer* to have the following use cases. 1) Data center operators can estimate fleet-wide gains from optimizing key service overheads. 2) Architects can make better accelerator design decisions and estimate realistic gains by considering offload overheads due to microservice software design.

VALIDATING AND APPLYING ACCELEROMETER

We validate *Accelerometer*'s utility via three retrospective case studies on production systems, by comparing model-estimated speedup with real microservice speedup determined via A/B testing. Each study covers a distinct microservice threading scenario (i.e., `Sync`, `Sync-OS`, and `Async`). We analyze 1) an on-chip accelerator: a specialized hardware instruction for encryption, AES-NI; 2) an off-chip accelerator: an encryption device connected to the host CPU via a PCIe link; and 3) a remote accelerator: a general-purpose CPU that solely performs ML inference and is connected to the host CPU via commodity network. In all three studies, we show that *Accelerometer* estimates the real microservice speedup with $\leq 3.7\%$ error.

Finally, we use *Accelerometer* to project speedup for the acceleration recommendations derived from three key common overheads identified by our characterization: compression, memory copy, and memory allocation.

LONG-TERM IMPLICATIONS

We discuss long-term implications, highlighting the impact this work has already had.

Accelerometer in Production

As microservices evolve, *Accelerometer*'s generality makes it even more suitable in determining new hardware requirements early in the design phase. Since we validated *Accelerometer* in production and made it open-source,[12] we are happy to report that it has been adopted by multiple hyperscale companies (e.g., with developing their encryption and compression accelerators) to make well-informed hardware decisions. We expect *Accelerometer* to trigger research in developing more complex models that account for overheads induced by offloading to specific accelerators (e.g., software batching implications on FPGA memory bandwidth versus latency).

Influence on Real Hardware Designs

In this work, we take a step back and answer the Amdahl's Law question of: Which overheads prevail even after offloading a microservice's main functionality to accelerators? Our comprehensive study of real-world microservices definitively indicates the need for a qualitatively different approach to future accelerator efforts. So far, data center hardware acceleration efforts have primarily focused on the most costly operations of a few "killer" applications (e.g., ML inference). However, accelerating orchestration overheads can offer greater benefits as they are significant and common across microservices.

As web service architectures grow more fragmented (e.g., deeper microservice pipelines and serverless architectures), it becomes more important to optimize the increasingly ubiquitous orchestration overheads. However, accelerating orchestration overheads is nontrivial as 1) orchestration libraries are already well-optimized in software and 2) orchestration function invocations are frequent, involve small data granularity, and are interspersed between other microservice code. Hence, accelerating orchestration overheads will require different techniques than those used in throughput-based specialization blocks with coarse-grained offloads (e.g., video processing).

ALTHOUGH ACCELEROMETER PROVIDES THE FIRST STEP IN DETERMINING REQUIRED ACCELERATION STRATEGIES, WE EXPECT SIGNIFICANT ACADEMIC AND INDUSTRIAL INTEREST IN RETHINKING ACCELERATORS FOR FINE-GRAINED ORCHESTRATION OPERATIONS.

Although *Accelerometer* provides the first step in determining required acceleration strategies, we expect significant academic and industrial interest in rethinking accelerators for fine-grained orchestration operations. Already, a few hardware vendors have used our study's insights to influence hardware customization for orchestration operations.

Characterization Approach and Tool

While it is relatively simple to measure the CPU cycles spent in leaf functions, it is extremely difficult to categorize every path's functionality in a microservice's entire call stack, as microservices have deep, complex software stacks that are hard to parse and classify. We developed a methodology to systematically classify each call trace path: We applied expert insights to identify service functionality classification rules that we then used to categorize cycles spent in various microservice functionalities.

We integrated this characterization tool into our fleet-wide performance monitoring infrastructure; it currently assimilates statistics from hundreds of thousands of servers from around the world to help developers visualize the performance impact of their code changes at hyperscale. With the decline of hardware performance scaling, there is a greater need for researchers to develop such tools for performance monitoring and optimization at all levels of the systems stack.

Industry-Academia Collaborative Benchmarking Efforts

Many hardware vendors rely on open-source benchmarks such as SPEC that heavily use C libraries to make architecture decisions. Hence, in our characterization, we observe that only C libraries' IPC scales well across CPU generations, but the other overheads (e.g., memory movement and encryption) show little to no improvement.

There is immense value in validating commonly used benchmarks with real-world application behaviors. Our characterization drove a hardware vendor to consider more representative benchmarks (in place of traditional ones they used for decades) when evaluating hardware designs. This work has resulted in an industry–academia joint collaborative effort to design and open-source scale-out cloud benchmarks that represent the hyperscale behaviors identified in our characterization. We expect our comprehensive study to drive continued benchmarking efforts that represent the severity of overheads in production-grade software.

End-to-End Thinking in Accelerator Design

Oftentimes, when designing accelerators, architects tend to miss the end-to-end picture, i.e., overheads that might arise from other system parts. When trying to adopt these accelerators at hyperscale, we have often found that they degrade performance due to overlooked offload-induced overheads. *Accelerometer* is a simple, powerful tool to help architects analytically estimate offload-induced overheads that arise from the end-to-end path, projecting realistic gains early in the hardware design phase.

REFERENCES

1. S. Kanev *et al.*, "Profiling a warehouse-scale computer," in *Proc. Int. Symp. Comput. Archit.*, 2015, pp. 158–169, doi: 10.1145/2749469.2750392.
2. A. Mirhosseini, A. Sriraman, and T. F. Wenisch, "Enhancing server efficiency in the face of killer microseconds," in *Proc. Int. Symp. High Perform. Comput. Archit.*, 2019, pp. 185–198, doi: 10.1109/HPCA.2019.00037.

3. A. Sriraman and T. F. Wenisch, “μSuite: A benchmark suite for microservices,” in *Proc. IEEE Int. Symp. Workload Characterization*, 2018, pp. 1–12, doi: 10.1109/IISWC.2018.8573515.
4. A. Sriraman and A. Dhanotia, “Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale,” in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2020, pp. 733–750, doi: 10.1145/3373376.3378450.
5. J. L. Henning, “SPEC CPU2006 benchmark descriptions,” *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, 2006, pp. 1–17, doi: 10.1145/1186736.1186737.
6. S. Kanev, S. L. Xi, G.-Y. Wei and D. Brooks, “Mallacc: Accelerating memory allocation,” in *Proc. Int. Conf. Archit. Support Program. Lang. Operating Syst.*, 2017, pp. 33–45, doi: 10.1145/3037697.3037736.
7. A. Sriraman and T. F. Wenisch, “μTune: Auto-tuned threading for OLDI microservices,” in *Proc. USENIX Conf. Operating Syst. Des. Implementation*, 2018, pp. 177–194. [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/sriraman
8. A. Sriraman, A. Dhanotia, and T. F. Wenisch, “SoftSKU: Optimizing server architectures for microservice diversity @scale,” in *Proc. Int. Symp. Comput. Archit.*, 2019, pp. 513–526, doi: 10.1145/3307650.3322227.
9. L. Luo *et al.*, “LASER: Light, Accurate Sharing dEtection and Repair,” in *Proc. Int. Symp. High Perform. Comput. Archit.*, 2016, pp. 261–273.
10. N. Jouppi *et al.* “In-datacenter performance analysis of a tensor processing unit,” in *Proc. Int. Symp. Comput. Archit.*, 2017, pp. 1–12.
11. M. S. B. Altaf, and D. A. Wood, “A high-level performance model for hardware accelerators,” in *Proc. Int. Symp. Comput. Archit.*, 2017, pp. 375–388.
12. “Accelerometer,” doi: 10.5281/zenodo.3612796.

AKSHITHA SRIRAMAN is currently a Ph.D. candidate in computer science and engineering at the University of Michigan, Ann Arbor, MI, USA. Her research bridges computer architecture and software systems, demonstrating the importance of that bridge in realizing efficient hyperscale web services via solutions that span the systems stack. Sriraman received an M.S. degree in Embedded Systems from the University of Pennsylvania. She is the corresponding author of this article. Contact her at akshitha@umich.edu.

ABHISHEK DHANOTIA is currently a Performance Engineer at Facebook, Menlo Park, CA, USA, where he works on designing their next-generation systems and improving efficiency of data center workloads. His research interests include computer architecture, performance analysis, and energy efficient system architectures for data centers. Dhanotia received an M.S. degree in computer engineering from North Carolina State University. Contact him at abhishekd@fb.com.