Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale

Akshitha Sriraman*† Abhishek Dhanotia†
University of Michigan*, Facebook†
akshitha@umich.edu, abhishekd@fb.com

Abstract

At global user population scale, important microservices in warehouse-scale data centers can grow to account for an enormous installed base of servers. With the end of Dennard scaling, successive server generations running these microservices exhibit diminishing performance returns. Hence, it is imperative to understand how important microservices spend their CPU cycles to determine acceleration opportunities across the global server fleet. To this end, we first undertake a comprehensive characterization of the top seven microservices that run on the compute-optimized data center fleet at Facebook.

Our characterization reveals that microservices spend as few as 18% of CPU cycles executing core application logic (e.g., performing a key-value store); the remaining cycles are spent in common operations that are not core to the application logic (e.g., I/O processing, logging, and compression). Accelerating such common building blocks can greatly improve data center performance. Whereas developing specialized hardware acceleration for each building block might be beneficial, it becomes risky at scale if these accelerators do not yield expected gains due to performance bounds precipitated by offload-induced overheads. To identify such performance bounds early in the hardware design phase, we develop an analytical model, *Accelerometer*, for hardware acceleration that projects realistic speedup in microservices. We validate *Accelerometer*'s utility in production using three retrospective case studies and demonstrate how it estimates the real speedup with ≤ 3.7% error. We then use *Accelerometer* to project gains from accelerating important common building blocks identified by our characterization.

CCS Concepts • **Computer systems org.** → **Cloud computing**; *Heterogeneous (hybrid) systems.*

ASPLOS '20, March 16–20, 2020, Lausanne, Switzerland

ACM ISBN 978-1-4503-7102-5/20/03.
https://doi.org/10.1145/3373376.3378450

Keywords data center; microservice; acceleration; analytical model

ACM Reference Format:
Akshitha Sriraman and Abhishek Dhanotia. 2020. Accelerometer: Understanding Acceleration Opportunities for Data Center Overheads at Hyperscale. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20), March 16–20, 2020, Lausanne, Switzerland.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3373376.3378450

1 Introduction

The increasing user base and feature portfolio of web applications is driving precipitous growth in the diversity and complexity of the back-end services comprising them [63]. There is a growing trend towards microservice implementation models [1, 5, 13, 15, 124], wherein a complex application is decomposed into distributed microservices [64, 90, 110, 112, 113] that each provide specialized functionality [115], such as HTTP connection termination, key-value serving [46], protocol routing [11, 129], or ad serving [52]. At hyperscale, this deployment model uses standardized Remote Procedure Call (RPC) interfaces to invoke several microservices to serve a user's query. Hence, upon receiving an RPC, a microservice must often perform operations such as I/O processing, decompression, deserialization, and decryption, before it can execute its core functionality (e.g., key-value serving).

At global user population scale, important microservices can grow to account for an enormous installed base of physical hardware. Across Facebook's global server fleet, seven important microservices in four diverse service domains run on hundreds of thousands of servers and occupy a large portion of the compute-optimized installed base. With the end of Dennard scaling [45, 120], successive server generations running these microservices exhibit diminishing performance returns. These microservices' importance begs the question: which microservice operations consume the most CPU cycles? Are there common overheads across microservices that we might address when designing future hardware?

To this end, we undertake a comprehensive characterization of microservices' CPU overheads on Facebook production systems serving live traffic. Since microservices must invoke common *leaf functions* at the end of a call trace (e.g.

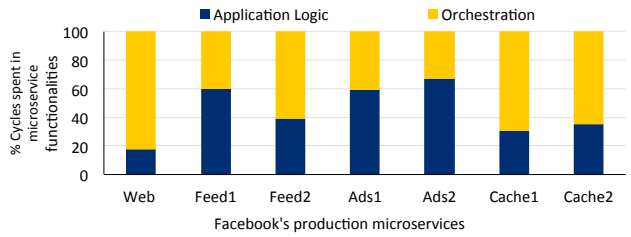

Figure 1. Breakdown of cycles spent in core application logic vs. orchestration work: orchestration overheads can significantly dominate.

`memcpy()`) [63], we first characterize their leaf function overheads. We then characterize *microservice functionalities* to determine (1) whether diverse microservices execute common types of operations (e.g., compression, serialization, and encryption) and (2) the overheads they induce. Both studies can help identify important acceleration opportunities that might be addressed in future software or hardware designs.

We find that application logic disaggregation across microservices at hyperscale has resulted in significant leaf function and microservice functionality overheads. For example, several microservices spend only a small fraction of their execution time serving core application logic, squandering significant cycles facilitating the core logic via *orchestration* work that is not core to the application logic (e.g., compression, serialization, and I/O processing), as shown in Fig.1. Accelerating the core application logic alone can yield only limited performance gains—an important Machine Learning (ML) microservice can speed up by only 49% even if its ML inference takes no time. Our `Web` microservice entails surprisingly high overheads from reading and updating logs. Caching microservices [34] can spend 52% of cycles sending/receiving I/O to support a high request rate and consequent I/O compression and serialization overheads dominate. Copying, allocating, and freeing memory can consume 37% of cycles, and kernel scheduler and network overheads are high with poor IPC scaling. Many microservices face common orchestration overheads despite great diversity in microservices' core application logic.

Such significant and common overheads might offer opportunities to accelerate common building blocks across microservices. Indeed, we report acceleration opportunities that might inform future software and hardware designs. However, introducing hardware acceleration 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 uncertainties inherent in projecting customer demand, deploying diverse custom hardware is risky at scale as the hardware might under-perform due to performance bounds from offload-induced overheads. As such, there is a need for simple analytical models that identify performance bounds early in the hardware design phase to project gains from accelerating overheads.

To this end, we develop an analytical model for hardware acceleration, *Accelerometer*, that identifies performance bounds to project microservice speedup. Whereas a few prior models [20, 38] estimate speedup from acceleration, they fall short in the context of microservices as they assume that the CPU waits while the offload operates. However, for many microservice functionalities, offload may be asynchronous; the processor may continue doing useful work concurrent with the offload. We extend prior models [20, 38] to capture such concurrency-induced performance bounds to project microservice speedup from hardware acceleration.

We demonstrate *Accelerometer*'s utility using three retrospective case studies conducted on production systems serving live traffic. First, we analyze an on-chip acceleration strategy—a specialized hardware instruction for encryption, AES-NI [8]. Second, we study an off-chip accelerator—an encryption device connected to the host CPU via a PCIe link. In the final study, we analyze a remote acceleration strategy—a general-purpose CPU that solely performs ML inference and is connected to the host CPU via a 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 important common overheads identified by our characterization—compression, memory copy, and memory allocation.

In summary, we contribute:

- A comprehensive characterization of leaf function overheads experienced by production microservices at Facebook: one of the largest social media platforms today.
- A detailed study of microservice functionality breakdowns, identifying orchestration overheads and highlighting potential design optimizations.
- *Accelerometer*[1]: An analytical model to project microservice speedup for various acceleration strategies.
- A detailed demonstration of *Accelerometer*'s utility in Facebook's production microservices using three retrospective case studies.

The rest of the paper is organized as follows: We describe and characterize the production microservices in Sec. 2. We explain the *Accelerometer* analytical model in Sec. 3. We validate and apply *Accelerometer* in Sec. 4 and Sec. 5, compare against related work in Sec. 6, and conclude in Sec. 7.

2 Understanding Microservice Overheads

We aim to identify how Facebook's important microservices spend their CPU cycles executing (1) leaf functions and (2) various microservice functionalities to determine software and hardware acceleration opportunities. We first characterize leaf functions (e.g., `memcpy()`) across diverse microservices. Whereas a leaf function study can provide insight into

[1] Available at https://github.com/akshithasriraman/Accelerometer and https://doi.org/10.5281/zenodo.3612797

common software building blocks, it does not reveal whether services share common functionalities that can be accelerated (e.g., compression). Hence, we additionally characterize service functionalities to identify common overheads that can benefit from acceleration. We also study Instructions Per Cycle (IPC) scaling for both the leaf and microservice functionality breakdowns to identify optimizations for overhead categories that scale poorly across CPU generations. In this section, we (1) describe each microservice, (2) explain our characterization approach, (3) characterize leaf functions, (4) report on microservice functionality breakdowns, and (5) summarize our characterization's key conclusions.

2.1 The Production Microservices

We study seven microservices in four diverse service domains that account for a large portion of Facebook's data center fleet. We characterize on production systems serving live traffic. We first detail each microservice's functionality.

Web. Web implements the HipHop Virtual Machine, a Just-In-Time system for PHP and Hack [17, 95, 128], to serve web requests from end-users. Web employs request-level parallelism to make frequent calls to other microservices: an incoming request is assigned to a worker thread, which services the request until completion.

Feed1 and Feed2. Feed1 and Feed2 are two microservices in our News Feed service. Feed2 aggregates various leaf microservices' responses into discrete "stories" that are then characterized into dense feature vectors by feature extractors and learned models [26, 51, 101, 132]. The feature vectors are sent to Feed1, which calculates and returns a predicted user relevance vector. Stories are then ranked and selected for display based on the relevance vectors.

Ads1 and Ads2. Ads1 and Ads2 maintain user-specific and ad-specific data, respectively [52]. When Ads1 receives an ad request, it extracts user data from the request and sends targeting information to Ads2. Ads2 maintains a sorted ad list, which it traverses to return ads meeting the targeting criteria to Ads1. Ads1 then ranks the returned ads.

Cache1 and Cache2. Cache is a large distributed-memory object caching service (similar to widely-used caching benchmarks [34, 35, 46, 123]) that reduces throughput requirements of various backing stores. Cache1 and Cache2 correspond to two tiers in each geographic region for Cache. Client services contact the Cache2 tier. If a request misses in Cache2, it is forwarded to the Cache1 tier. Cache1 misses are sent to an underlying database cluster in that region.

2.2 Characterization Approach

We characterize the seven microservices by profiling each in production while serving real-world user queries. We next describe the characterization methodology.

Hardware platforms. We characterize our microservices on 18- and 20-core Intel Skylake processors [42] (see Table 1). We run Web, Feed1, Feed2, and Ads1 on the 18-core Skylake, and Ads2, Cache1, and Cache2 on the 20-core Skylake. We study IPC scaling across three CPU generations (Table 1).

Table 1. GenA, GenB, and GenC CPU platforms' attributes.

	GenA	GenB	GenC
μarchitecture	Intel Haswell	Intel Broadwell	Intel Skylake
Cores / socket	12	16	18 or 20
SMT	2	2	2
Cache block size	64 B	64 B	64 B
L1-I$ / core	32 KiB	32 KiB	32 KiB
L1-D$ / core	32 KiB	32 KiB	32 KiB
Private L2$ / core	256 KiB	256 KiB	1 MiB
Shared LLC	30 MiB	24 MiB	24.75 or 27 MiB

Experimental setup. We measure each microservice in our production environment's default deployment—standalone with no co-runners on bare metal hardware. There are no cross-service contention or interference effects in our data. We study each system at peak load to stress performance bottlenecks.

We characterize leaf functions by first using Strobelight [14] to measure instructions and cycles spent in microservices' key leaf functions. We then feed leaf functions and their cycle counts to an internal tool that tags each leaf function's category (e.g., tagging `memcpy()` as "memory"); the tool then aggregates cycles spent in each leaf category.

To characterize microservice functionality, we use Strobelight [14] to (1) collect all function call traces of a microservice (e.g., a function call trace can be composed of a function sequence starting with cloning a thread and ending with a leaf function such as `memcpy()`) and (2) measure cycles and instructions spent in each call trace. We feed the function call traces and their cycle counts to an internal tool that buckets each function call trace into a microservice functionality category (e.g., I/O, serialization, and compression); it then aggregates cycles spent in each category. To determine a category's IPC, we determine the ratio of aggregated instruction and cycle counts for functions in that category. We contrast our measurements with some SPEC CPU2006 [54] benchmarks and Google services [63] where possible.

2.3 Leaf Function Characterization

We first present key leaf function breakdowns for our microservices and compare them with SPEC CPU2006 [54] and Google services [25, 63]. We then characterize a few key leaf functions in greater detail. Finally, we report IPC scaling measurements for Cache1's leaf functions across three CPU generations.

We define each leaf function category in Table 2. We report the fraction of overall cycles spent in each leaf category in Fig. 2 (we omit bars that consume <1% of cycles). We also omit bars for 401.bzip2, 429.mcf, 445.gobmk, 456.hmmer, 458.sjeng, 462.libquantum, 464.h264ref, and 483.xalancbmk

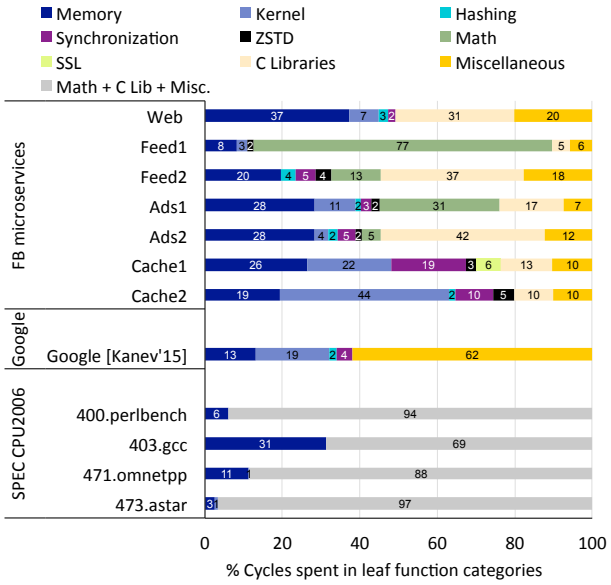

Figure 2. Breakdown of cycles spent in leaf functions: memory functions consume a significant portion of total cycles.

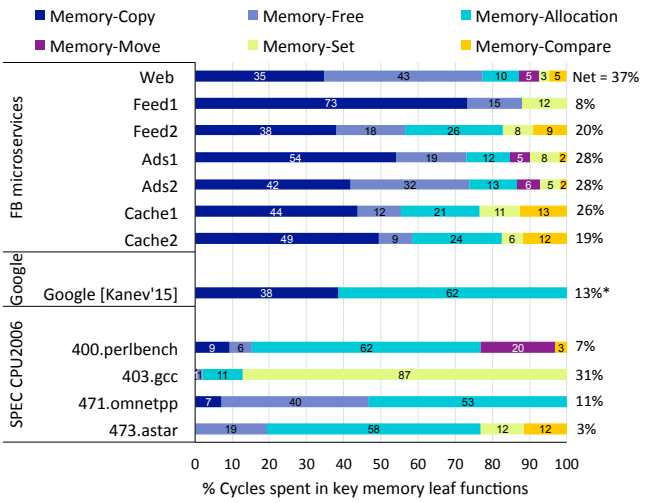

Figure 3. Breakdown of cycles spent in memory leaf functions as a fraction of total cycles: memory copy, allocation, & free consume significant cycles.

Table 2. Categorization of 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

SPEC CPU2006 benchmarks since their leaves are composed of either math functions or C libraries.

We make several observations. First, most microservices spend a significant fraction of cycles on memory functions (e.g., copy and allocation) and kernel operations. `Cache1` and `Cache2` spend more cycles in the kernel as they frequently incur context switches due to a high service throughput [111]. We further break down the memory and kernel function categories (Sec. 2.3.1 and 2.3.2) to identify specific optimizations.

Second, ML microservices such as `Ads2` and `Feed2` spend only up to 13% of cycles on mathematical operations that constitute ML inference using Multilayer Perceptrons. We find that these services can also benefit from optimizations to C libraries, which we investigate further in Sec. 2.3.4.

Third, `Cache1` and `Cache2` spend significant cycles synchronizing frequent communication between distinct thread pools. Additionally, we find that `Cache1` spends 6% of cycles in leaf encryption functions since it encrypts a high number of Queries Per Second (QPS).

Fourth, Google's breakdown across their global server fleet [63] is similar to Facebook's leaf breakdowns. In contrast, SPEC CPU2006 [54] benchmarks do not capture key leaf overheads (e.g., memory and kernel) faced by our microservices; their functions primarily belong to the math, C libraries, and miscellaneous categories.

We conclude many leaf function overheads are significant and common across services. We next investigate leaf functions in greater detail to identify acceleration opportunities.

2.3.1 Memory. In Fig. 3, we characterize cycles spent in various memory leaf functions as a fraction of total cycles spent in memory functions. The memory functions include memory copy, free, allocation, move, set, and compare. We compare our microservices with Google's services [63] and SPEC CPU2006 [54] benchmarks. Note that only the memory copy and allocation breakdowns are available for Google's services [63], and they account for 13% of total cycles (represented by an asterisk in Fig. 3).

We observe that memory copies are by far the greatest consumers of memory cycles. Google's services also spend 5% of total fleet cycles on memory copies [63]. Although `403.gcc` exhibits a high memory overhead, it spends very few cycles in copying memory. Memory copy optimizations such as (1) reducing copies in network protocol stacks [16], (2) performing dense memory copies via SIMD [4], (3) moving data in DRAM [105], (4) minimizing I/O copies using Intel's I/O Acceleration Technology (IO AT) [9], (5) processing in memory [18], (6) optimizing memory-based software libraries [92, 99], and (7) building hardware accelerators (e.g.,

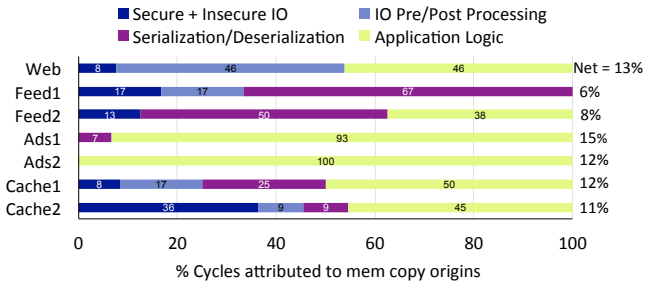

Figure 4. Breakdown of service functionalities that invoke memory copies: there is significant diversity in dominant functionalities that perform copies.

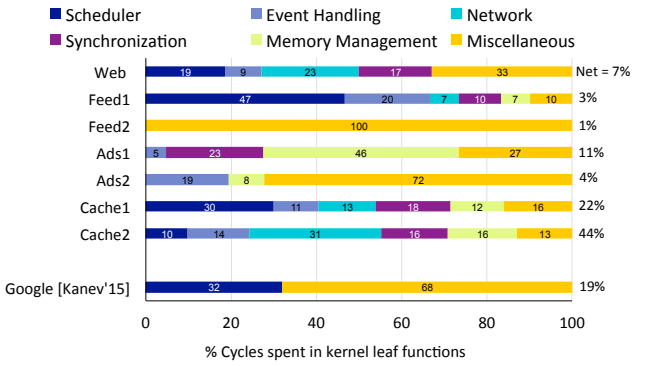

Figure 5. Breakdown of cycles spent in various kernel leaf functions: kernel scheduler, event handling, and network overheads can be high.

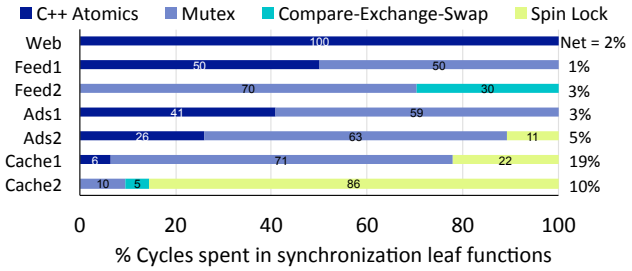

Figure 6. Breakdown of CPU cycles spent in synchronization functions: Cache frequently uses spin locks to avoid thread wakeup delays.

for memory-memory copies [82]) could minimize copy overheads. To identify optimizations, we also provide greater nuance to our memory copy characterization by attributing memory copies to various microservice functionalities.

We find that memory allocation can be a significant overhead despite using fast software allocation libraries [24]. Google's services incur a slightly greater allocation overhead. This observation suggests the need to continue to build software [3, 30, 58, 71, 74, 87, 131] and hardware optimizations [65, 70] for allocations. Of the SPEC CPU2006 [54] suite, `471.omnetpp` spends the most cycles on allocation (~5%).

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 [65]. 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 can 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 [7]. While numerous prior works focus on optimizing allocations [58, 63, 70], very few recognize that optimizing `free()` can result in significant performance wins.

Memory copy origins. In Fig. 4, we attribute memory copies to microservice functionalities defined in Table 3. We find that memory is primarily copied during (1) I/O pre- or post-processing, (2) I/O sends and receives, (3) RPC serialization/deserialization, and (4) application logic execution (e.g., executing key-value stores in Cache). We observe significant diversity in dominant service functionalities that invoke 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 post-processing [9, 105], whereas Cache2 can gain from fewer copies in network protocol stacks [16, 48].

2.3.2 Kernel. We depict the cycles spent in kernel leaf functions in Fig. 5. We make three observations: (1) Microservices with a high kernel overhead—Cache1 and Cache2—invoke scheduler functions frequently. Software/hardware optimizations [28, 29, 32, 41, 44, 60, 73, 75, 116] that reduce scheduler latency (e.g., intelligent thread switching and coalescing I/O) might considerably improve Cache performance. (2) Cache2 spends significant cycles in I/O and network interactions. Optimized systems [29, 60, 66, 77, 97, 98] that incorporate kernel-bypass and multi-queue NICs might minimize Cache2's kernel overhead. (3) Prior work [63] only reports the kernel scheduler overhead for Google's services. They typically mirror overheads seen in Cache1 and Cache2.

2.3.3 Synchronization. Microservices such as Cache oversubscribe threads to improve service throughput [111]. Hence, such microservices frequently synchronize various thread pools. We portray these synchronization overheads in Fig. 6. We find that Cache, which exhibits a high synchronization overhead, spends several cycles in spin locks that are typically deemed performance inefficient [22, 81]. However, Cache implements spin locks since it is a μs-scale microservice [111], and is hence more prone to μs-scale performance

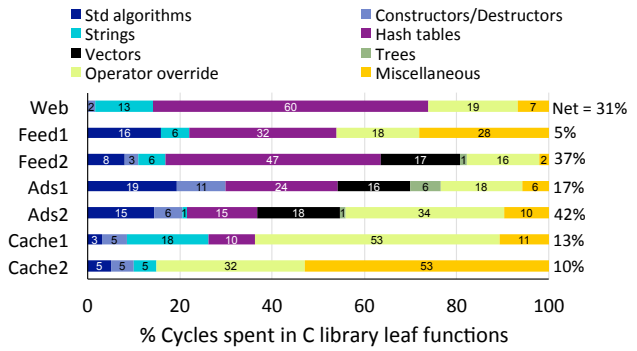

Figure 7. Breakdown of CPU cycles spent in C libraries: ML services perform several vector operations while dealing with large feature vectors.

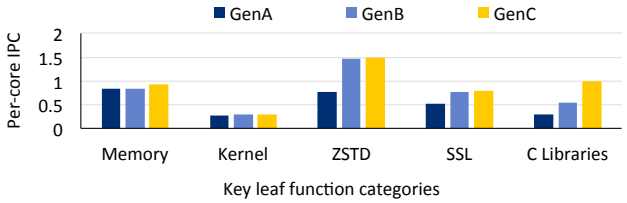

Figure 8. Cache1's IPC scaling across three CPU generations for key leaf funcs.: kernel IPC is typically low & scales poorly.

penalties that can otherwise arise from thread re-scheduling, wakeups, and context switches [113].

2.3.4 C Libraries. We characterize overheads from C libraries in Fig. 7. We observe that `Feed2`, `Ads1`, and `Ads2` perform several vector operations as they deal with large feature vectors. `Web` spends significant cycles parsing and transforming strings to process queries from the many URL endpoints it implements. `Web` also performs several hash table look-ups to (1) maintain query parameters, (2) identify services to contact, and (3) merge responses. We conclude many microservices can benefit from optimizing vector operations [72], string computations [49, 107], and hash table look-ups [108, 118].

2.3.5 IPC scaling. We show `Cache1`'s per-core IPC scaling for key leaf functions in Fig. 8. We report IPC across three CPU generations to see whether IPC scales as expected.

We make several observations: (1) Each leaf function type uses less than half of the theoretical execution bandwidth of a GenC CPU (theoretical peak IPC of 4.0). As such, simultaneous multithreading is effective for these microservices and is enabled in our CPUs. Given our production microservices' larger codebase, larger working set, and more varied memory access patterns, we do not find a lower typical IPC surprising. (2) Kernel IPC is typically low and also scales poorly. Accelerating the kernel is non-trivial as it is neither small, nor self-contained, and cannot be easily optimized

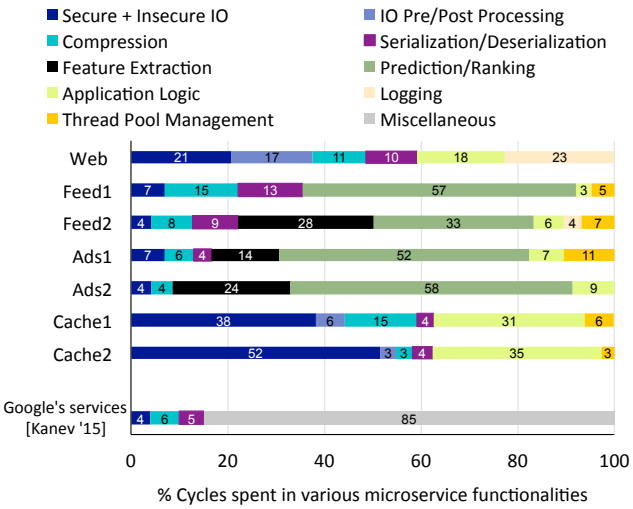

Figure 9. Breakdown of CPU cycles spent in various microservice functionalities: orchestration overheads are significant & fairly common.

Table 3. Categorization of microservice functionalities.

Functionality category	Examples of service operations
Secure and 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	Core business logic (e.g., Cache's key-value serving)
Logging	Creating, reading, updating logs
Thread pool management	Creating, deleting, synchronizing threads

in hardware. However, software optimizations that minimize scheduler, I/O, and network overheads can improve kernel IPC [28, 29, 41, 60]. (3) C libraries' IPC scales well across CPU generations. This observation is unsurprising as many hardware vendors primarily rely on open-source benchmarks [54] that heavily use C libraries (see Fig. 2) to make architecture design decisions. (4) Typically, we see only a small IPC gain from GenB to GenC. This trend suggests the need to specialize hardware for key leaf functions.

2.4 Service Functionality Characterization

We attribute CPU cycles to microservice functionalities in Fig. 9 to identify key microservice overheads (as motivated in Fig. 1). We define how we pool various functionalities in Table 3. Note that each functionality category typically includes several leaf function categories. For example, despite ML inference being heavy on math leaf functions, it can also comprise memory movement and C library leaves.

We make four observations: First, several microservices face significant orchestration overheads from performing operations that are not core to the application logic, but instead facilitate application logic such as compression, I/O, and logging. For example, the 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; (note that the "application logic" for these microservices includes core non-ML operations such as merging results). Hence, even if modern inference accelerators [23, 62, 125, 127] were to offer an infinite inference speedup, the net microservice performance would only improve by 1.49x - 2.38x. There is hence a great need for architects to accelerate the orchestration work that facilitates the core application logic.

Second, several orchestration overheads are common across microservices; accelerating them can significantly improve our global fleet's performance. For example, Web, Cache1, and Cache2 spend a significant portion of cycles executing I/O—i.e., sending and receiving RPCs. 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 [111]—they frequently invoke RPCs to communicate with mid-tier microservices. These microservices can benefit from RPC optimizations such as kernel-bypass and multi-queue NICs [29, 60, 66, 77, 97, 98]. Additionally, Web, Feed1, Feed2, and Cache1 spend several cycles in compression and serialization (similar to Google's services [63]); they can benefit from accelerating these common orchestration overheads [21, 27, 36, 47, 68, 106].

Third, Web spends only 18% of cycles in core 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.

Fourth, Ads1, Feed2, Cache1, and Feed1 incur a high thread pool management overhead. Intelligent thread scheduling and tuning [50, 59, 69, 100, 119, 126] can help these services.

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.

2.4.1 IPC scaling. In Fig. 10, we show Cache1's per-core IPC for key microservice functionalities across three CPU generations. We find that the I/O IPC remains low across CPU generations. Since I/O calls primarily invoke kernel functions, the low kernel IPC (see Fig. 8) contributes to the low I/O IPC. Additionally, there is little IPC improvement for key-value store operations. Since key-value stores are typically memory intensive [34], the low memory IPC (Fig. 8) results in a low key-value store IPC.

We summarize our characterization findings in Table 4.

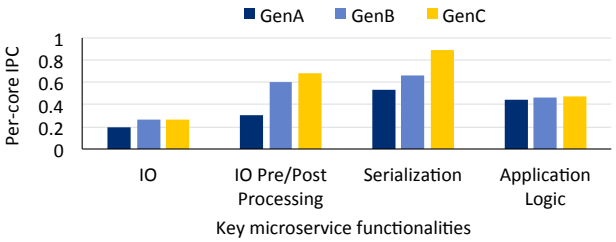

Figure 10. Cache1's IPC scaling across three CPU generations for key functionality categories: a low I/O IPC is primarily due to a low kernel IPC.

3 The Accelerometer Model

Overheads identified by our characterization can be accelerated in the hardware via CPU optimizations (e.g., specialized hardware instructions) [8, 88, 89] or custom accelerator devices (e.g., ASICs). Investing in hardware acceleration often requires (1) designing new hardware, (2) testing it, and (3) carefully planning capacity to provision the hardware to match projected load. Given the uncertainties inherent in projecting customer demand, investing in diverse custom hardware is risky at scale, as the hardware might not live up to its expectations due to performance bounds precipitated by offload-induced overheads [20].

Simple analytical models enable better hardware investments by identifying performance bounds early in the design phase. However, existing models for hardware acceleration [20, 38] fall short in the context of microservices as they are oblivious to offload overheads induced by microservice threading designs such as synchronous vs. asynchronous offload to an accelerator. For example, existing models [20, 38] assume that the CPU waits while the offload operates i.e., offload is synchronous. However, for many functionalities, offload may be asynchronous; the CPU may continue doing useful work concurrent with the offload. Extending prior models [20], we develop *Accelerometer* to capture this concurrency and realistically model microservice speedup for various hardware acceleration strategies (e.g., on-chip vs. off-chip). In this section, we (1) describe the acceleration strategies *Accelerometer* models, (2) discuss system abstractions it assumes, (3) define *Accelerometer*'s model parameters, (4) detail how it models speedup for various threading designs, and (5) highlight *Accelerometer*'s applications.

Acceleration strategies. *Accelerometer* models three kinds of hardware acceleration strategies to accelerate an algorithm or *kernel*—on-chip, off-chip, and remote.

On-chip. On-chip acceleration optimizes components on the CPU die (e.g., wider SIMD units [121], Intel's AES-NI hardware encryption instruction [8], and CPU modifications [67, 88]). Offload latencies are typically ns-scale.

Off-chip. Off-chip accelerators are typically contacted via PCIe and coherent interconnects [117] (e.g., GPUs, smart NICs, and ASICs). Offload latencies are $\sim \mu$s-scale [91].

Table 4. Summary of findings and suggestions for future optimizations.

Finding	Acceleration opportunity
Significant orchestration overheads (§2.4)	Software and hardware acceleration for orchestration rather than just app. logic
Several common orchestration overheads (§2.4)	Accelerating common overheads (e.g., compression) can provide fleet-wide wins
Poor IPC scaling for several functions (§2.3.5, §2.4.1)	Optimizations for specific leaf/service categories
Memory copies & allocations are significant (§2.3, §2.3.1)	Dense copies via SIMD, copying in DRAM, Intel's I/O AT, DMA via accelerators, PIM
Memory frees are computationally expensive (§2.3, §2.3.1)	Faster software libraries, hardware support to remove pages
High kernel overhead and low IPC (§2.3, §2.3.5)	Coalesce I/O, user-space drivers, in-line accelerators, kernel-bypass
Logging overheads can dominate (§2.4)	Optimizations to reduce log size or number of updates
High compression overhead (§2.3, §2.4)	Bit-Plane Compression, Buddy compression, dedicated compression hardware
Cache synchronizes frequently (§2.3, §2.3.3)	Better thread pool tuning and scheduling, Intel's TSX, coalesce I/O, vDSO
High event notification overhead (§2.3.2)	RDMA-style notification, hardware support for notifications, spin vs. block hybrids

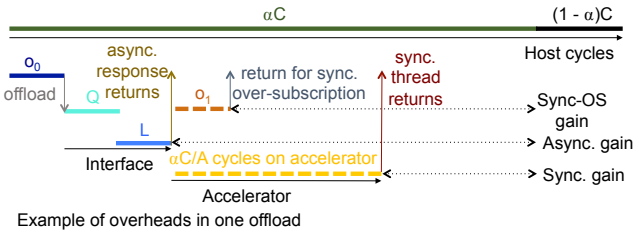

Figure 11. Example timeline of host & accelerator.

Remote. Remote accelerators are off-platform devices contacted via the network. Examples include remote ML inference units [51], network switches [78, 104], remote encryption units [31], and remote GPUs [43]. Offload latencies are typically ms-scale when using commodity ethernet [102].

System abstraction. *Accelerometer* assumes an abstract system with three components (1) *host*—a general-purpose CPU, (2) *accelerator*—custom hardware to accelerate a kernel, and (3) *interface*—the communication layer between the host and the accelerator (e.g., a PCIe link). The interface helps define overheads from dispatching work to an accelerator (e.g., preparing the kernel for offload, offload latency, and queuing delays). Hence, the interface abstraction can easily help model speedup for diverse acceleration strategies. With these system abstractions, we build the *Accelerometer* model such that it abstracts the underlying hardware architecture using parameters defined below (see Table 5).

Parameter definition. *Accelerometer* makes a few assumptions to retain model simplicity while still being able to estimate microservice speedup. Similar to LogCA [20], *Accelerometer* assumes that (1) the kernel's execution time is a function of *granularity* g—i.e., the data offload size and (2) the host and accelerator use kernels of the same complexity. It defines C as the total host cycles spent to execute both kernel and non-kernel logic in a fixed time unit; C is inversely proportional to the host's busy frequency for a time unit of one second. It uses Amdahl's law to define a constant $\alpha \leq 1$, such that the host spends $(\alpha * C)$ cycles executing the kernel and $((1-\alpha) * C)$ cycles executing the non-kernel logic (as shown in Fig. 11). *Accelerometer* assumes that data offload is unpipelined (i.e., the accelerator requires the entire block to start operating); it considers the average latency of such an offload, L. The offload latency distribution can be found by multiplying the offload latency of a single byte with g for each offload. When data offload is pipelined, L is independent of g; we do not study pipelined offloads as our existing systems use unpipelined offloads. The peak achievable accelerator speedup factor, A, helps define cycles spent in the accelerator such that cycles spent on the host to execute the kernel is cut down by the acceleration factor, or $\frac{\alpha * C}{A}$.

Modeling diverse threading designs. We develop *Accelerometer* to model the microservice throughput speedup (referred to as "speedup") and the microservice per-request latency speedup (referred to as "latency reduction") for the three acceleration strategies. Modeling both speedup and latency reduction helps ensure that acceleration enables a higher throughput (i.e., more QPS) without violating latency Service Level Objectives (SLOs). To model speedup, *Accelerometer* identifies how many fewer host cycles are needed to execute the kernel when there is acceleration—spending fewer host cycles on the kernel frees up host cycles to do more work, improving throughput. It defines speedup as the ratio of total cycles spent by the host when there is no acceleration, C, to the total cycles spent by the host when the kernel is accelerated, C_S, or $\frac{C}{C_S}$. To model per-request latency reduction, it identifies the total cycles taken to execute a request when there is acceleration; spending fewer cycles for a request due to acceleration reduces per-request latency. It defines latency reduction as the ratio of C to the total cycles spent on the host and the accelerator, C_L, or $\frac{C}{C_L}$.

Unlike LogCA [20], we find that when data is offloaded to an accelerator, the speedup $\frac{C}{C_S}$ and latency reduction $\frac{C}{C_L}$ depend on the acceleration strategy as well as the threading design used to offload (e.g., synchronous vs. asynchronous offload). For example, in a synchronous offload the host waits for the accelerator's response before resuming execution (see Fig. 11), putting the accelerator's operation cycles ($\frac{\alpha * C}{A}$) in the critical path of the host's execution (i.e., C_S), impacting speedup. Conversely, in an asynchronous offload, the host continues doing useful work concurrent with the accelerator's operation on the offload, removing $\frac{\alpha * C}{A}$ from the critical

Table 5. Description of the Accelerometer analytical model parameters.

Symbol	Parameter description	Units
C	Total cycles spent by the host to execute all logic in a fixed time unit	Cycles
g	Size of an offload	Bytes
n	Number of times the host offloads a kernel of lucrative size in a fixed time unit	N/A
o_0	Cycles the host spends in setting up the kernel prior to a single offload	Cycles
Q	Avg. cycles spent in queuing between host and accelerator for a single offload	Cycles
L	Avg. cycles to move an offload from host to accelerator across the interface, including cycles the data spends in caches/memory	Cycles
o_1	Cycles spent in switching threads (due to context switches and cache pollution) for a single offload	Cycles
A	Peak speedup of an accelerator	N/A
α	A constant ≤ 1	N/A
C_b	Cycles spent by the host per byte of offload data	Cycles

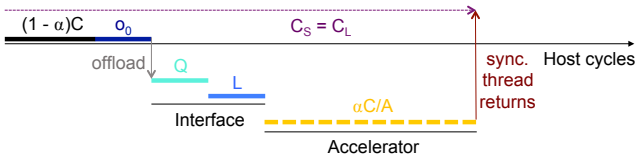

Figure 12. Modeling Sync C_S and C_L for one offload.

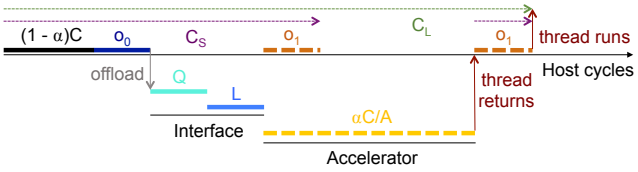

Figure 13. Modeling Sync-OS C_S and C_L for one offload.

path of C_S. We extend LogCA to model speedup and latency reduction for both synchronous and asynchronous offload.

(1) Synchronous. When a host thread offloads work to an accelerator synchronously, it waits in the `blocked` state for the accelerator's response. If the microservice runs one thread per core, the host's core waits for the accelerator's response—we refer to this scenario as Sync. Hence, C_S and C_L will include cycles spent on the accelerator $\frac{\alpha * C}{A}$, as shown in Fig. 12. Moreover, 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. Hence, C_S and C_L can also include these additional overheads per offload. Considering n offloads occur in a given time unit, *Accelerometer* defines Sync speedup $\frac{C}{C_S}$ and latency reduction $\frac{C}{C_L}$ as: $\frac{C}{(1-\alpha)C + \frac{\alpha C}{A} + n(o_0 + L + Q)}$, where C_S and C_L comprise host cycles spent in (1) non-kernel logic, (2) waiting for the accelerator's response, and (3) offload-induced overheads across the n offloads. Making this equation appear similar to Amdahl's law with offload overheads (i.e., dividing by C),

$$\text{Sync } \frac{C}{C_S} or \frac{C}{C_L} = \frac{1}{(1-\alpha) + \frac{\alpha}{A} + \frac{n}{C}(o_0 + L + Q)} \quad (1)$$

In eqn.(1), $(n * Q)$ is the mean queuing delay for n offloads; Q enables projecting speedup based on accelerator load. Replacing $(n * Q)$ with $\sum_{i=1}^{n}(Q_i)$ models the queuing distribution. Net speedup is >1 when the host spends more cycles when unaccelerated—i.e., $(\alpha * C) > \frac{\alpha C}{A} + n(o_0 + L + Q)$. In eqn. (1), we consider n kernel offloads that each improve speedup (or reduce latency). To determine whether a kernel offload improves speedup, we consider the offload granularity, g, such that the host spends C_b cycles per byte of g. A single offload improves speedup when the cycles a host would spend in executing all bytes of a g-size kernel offload is greater than the cycles spent in accelerating the kernel (i.e., the sum of cycles spent on the accelerator executing the g-size offload and the offload overheads—$o_0 + L + Q$), or:

$$C_b * g > \frac{C_b * g}{A} + o_0 + L + Q \quad (2)$$

Eqn. (2) can be extended to model the kernel's complexity (e.g., sub-linear, linear, or super-linear) using g^β [20]. For example, $\beta = 1$ for a linear complexity kernel.

In reality, several microservices (e.g., `Web` and `Cache`) oversubscribe threads to improve throughput by having more threads than available cores. Oversubscription allows a host to schedule an available thread to process new work, while the thread that offloaded work `blocks` awaiting the accelerator's response. The host continues to perform useful work instead of wasting cycles in awaiting the accelerator's response; we refer to this synchronous thread Over-Subscription as Sync-OS. Hence, the accelerator's cycles $\frac{\alpha C}{A}$ do not affect C_S, as shown in Fig. 13. Instead, C_S is affected by the OS-induced overhead to switch to an available thread, o_1. The $(L+Q)$ overhead persists when the host's device driver synchronously awaits an offload acknowledgement from an off-chip accelerator before switching threads. However, $(L+Q) = 0$ when (1) the device driver does not wait for the off-chip accelerator's acknowledgement or (2) the accelerator is remote. Hence, the speedup is:

$$\text{Sync-OS } \frac{C}{C_S} = \frac{1}{(1-\alpha) + \frac{n}{C}(o_0 + L + Q + 2o_1)} \quad (3)$$

Speedup is >1 when: $(\alpha * C) > n(o_0 + L + Q + 2o_1)$. A single offload improves throughput speedup when the cycles a host would spend in executing that offload is greater than the

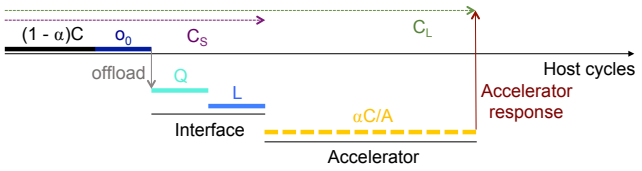

Figure 14. Modeling Async C_S and C_L for one offload.

offload-induced overhead—$o_0 + L + Q + 2o_1$, or:

$$C_b * g > o_0 + L + Q + 2o_1 \quad (4)$$

The latency reduction remains the same as eqn. (1) (since the accelerated per-request latency, C_L, will include cycles spent on the accelerator), but must now account for o_1. The μs-scale o_1 overhead [76, 122] can dominate in μs-scale microservices such as Cache [111], making it feasible to incur a throughput gain at the cost of a per-request latency slowdown. Service operators can use the following latency reduction equation to ensure that the latency SLO is not violated.

$$\text{Sync-OS } \frac{C}{C_L} = \frac{1}{(1-\alpha) + \frac{\alpha}{A} + \frac{n}{C}(o_0 + L + Q + o_1)} \quad (5)$$

Latency is reduced when: $(\alpha * C) > \frac{\alpha C}{A} + n(o_0 + L + Q + o_1)$. A single offload reduces latency when the cycles a host would spend in executing the offload dominates accelerator cycles and offload overheads, or: $(C_b * g) > \frac{C_b * g}{A} + (o_0 + L + Q + o_1)$.

(2) Asynchronous. After a host thread offloads work asynchronously, it continues to process new work without awaiting the accelerator's response. When the response arrives, it can be picked up by (1) the same thread that sent the request or (2) a distinct thread dedicated to pick up responses [114]. When a distinct thread picks up the response, the speedup equation is the same as (3) with only one thread switching overhead o_1. The latency reduction equation remains the same as (5). If the response is picked up by the same thread that sent the request, $o_1 = 0$ since the OS does not switch threads (see Fig. 14); we refer to this scenario as Async. Hence the speedup is:

$$\text{Async } \frac{C}{C_S} = \frac{1}{(1-\alpha) + \frac{n}{C}(o_0 + L + Q)} \quad (6)$$

Speedup is >1 when: $(\alpha * C) > n(o_0 + L + Q)$. A single offload improves speedup when:

$$C_b * g > o_0 + L + Q \quad (7)$$

Similarly, *Accelerometer* does not consider o_1 when modeling Async latency reduction:

$$\text{Async } \frac{C}{C_L} = \frac{1}{(1-\alpha) + \frac{\alpha}{A} + \frac{n}{C}(o_0 + L + Q)} \quad (8)$$

Latency reduces when: $(\alpha * C) > \frac{\alpha C}{A} + n(o_0 + L + Q)$. A single offload reduces latency when: $(C_b * g) > \frac{C_b * g}{A} + (o_0 + L + Q)$.

In some asynchronous designs, the host does not require the accelerator's response for further processing, eliminating o_1 (e.g., when a host sends requests to an encryption accelerator, which then sends encrypted requests to the next microservice). Hence, the speedup equation remains the same as eqn. (6). Latency reduction depends on whether acceleration is off-chip or remote since remote accelerator latencies $\frac{\alpha C}{A}$ will not affect a microservice's request latency and will instead show up in the overall application's end-to-end latency. We define the Async off-chip per-request latency reduction as eqn. (8) and the remote latency reduction as eqn. (6).

Applying the Accelerometer model. The *Accelerometer* model shows that speedup and latency reduction depend on the acceleration strategy and microservice threading design. We expect *Accelerometer* to have the following use cases: (1) Data center operators can project fleet-wide gains from optimizing key service overheads. (2) Architects can make better accelerator design decisions and estimate realistic gains by being aware of the offload overheads due to microservice design. *Accelerometer* can help determine tradeoffs between various acceleration strategies (e.g., on-chip vs. off-chip) for microservice overheads. Indeed, we validate our models in production and then apply them to project gains for on-chip vs. off-chip recommendations (see Table 4) derived from key overheads identified by our characterization.

4 Validating the Accelerometer Model

We validate *Accelerometer*'s utility in production using three retrospective case studies. With these studies, we validate all three microservice threading scenarios—Sync, Sync-OS, and Async. Each study covers a distinct acceleration strategy—i.e., on-chip, off-chip, or remote. For each study, we first describe (1) the experimental setup, (2) how we derive model parameters, and (3) how we measure speedup on production systems. We then validate *Accelerometer* by comparing model-estimated speedup with real microservice speedup. We do not compare the latency reduction since our existing production infrastructure lacks necessary support to precisely measure a microservice's per-request latency.

Validation methodology. We follow a five step process to validate the *Accelerometer* model: (1) we identify offload sizes g that improve speedup, (2) we determine the number of such offloads in one second, n, and the fraction of cycles they constitute, α, (3) we use the *Accelerometer* model to estimate speedup from these n offloads, (4) we compare *Accelerometer*-estimated speedup with real production speedup, and (5) we present a functionality breakdown for both the accelerated and unaccelerated microservices to show how throughput improves. We assume that we can use software to selectively accelerate only those kernel offloads that improve speedup (the kernel execution time can be dominated by overheads for very small offloads [20]).

Experimental setup. We perform our case studies on Intel Skylake processor platforms (Table 1). For each case study, we first measure the real production speedup using

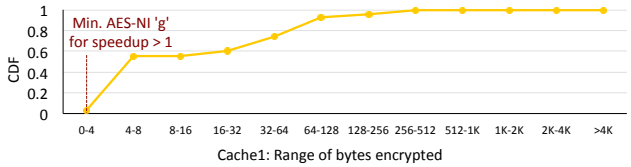

Figure 15. CDF of bytes encrypted in Cache1:<512B are frequently encrypted.

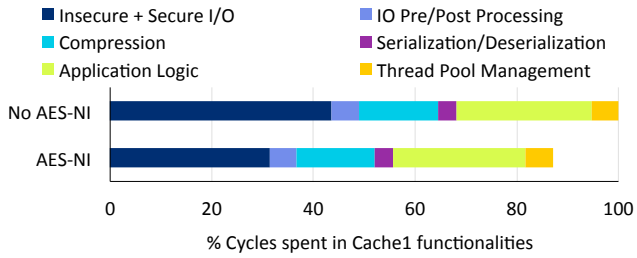

Figure 16. Breakdown of cycles spent in Cache1's functionalities for both the no-AES-NI (unaccelerated) & with-AES-NI (accelerated) cases: 12.8% of cycles are freed up with AES-NI.

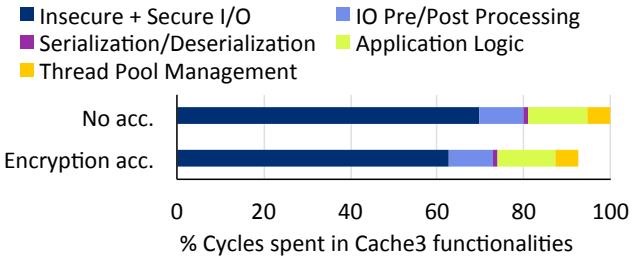

Figure 17. Breakdown of cycles spent in Cache3's functionalities when encryption is accelerated vs. not: secure IO calls are optimized with acceleration.

an internal tool called Operational Data Store (ODS) [19, 33, 96]. We measure speedup via A/B testing. A/B testing is the process of comparing two identical systems that differ only in a single variable. We conduct A/B tests by comparing the throughput (in QPS) of two identical servers (i.e., same hardware platform, same fleet, and facing the same load) that differ only in terms of whether they accelerate the kernel.

To determine the *Accelerometer*-estimated speedup, we assume a linear complexity kernel, since we cannot easily perform scaling studies on production systems to determine kernel complexity. We measure model parameters using (1) tools such as Strobelight [14], `bpftrace` [6], and `bcc-tools` [10], (2) roofline estimates from device specification sheets, and (3) micro-benchmarks that measure execution time on the host and the accelerator. Some host parameters, once calculated, can be re-used for different kernels on the same system. For each case study, we measure the unaccelerated host's busy frequency to calculate C for one second. To determine whether a specific offload improves speedup (using equations (2), (4), (7)), we use `bpftrace` [6] to measure g's size range and the number of invocations of each granularity. We compute n by aggregating invocations of those offload sizes that improve speedup. To determine α, we first use the service functionality breakdown (see Fig. 9) to estimate host cycles spent in the kernel under study. We then use n and these total host cycles to estimate the fraction of kernel cycles that must be offloaded, $(\alpha * C)$. We assume an unpipelined interface when estimating L.

Case study 1: AES-NI for Cache1. We study encryption in `Cache1` with Intel's AES-NI [8] instruction—an on-chip optimization. In this case, `Cache1` uses a `Sync` threading design. We use AES [2] from the OpenSSL [12] cryptography library to build micro-benchmarks to measure L, o_0, and A. We assume $Q = 0$, since the same host thread executes the AES-NI instruction. We show the Cumulative Distribution Function (CDF) of `Cache1`'s encryption granularities in Fig. 15. We use model parameters defined in Table 6 in eqn. (2) to determine that a specific offload improves net speedup when $g \geq 1$ Byte (B). Prior work [20] also sees wins with AES-NI for small offload granularities. From Fig. 15, we observe that `Cache1`'s encryption size is $\sim\geq$ 4 B; hence, all offloads will improve speedup. We confirm that `Cache1` offloads all encryptions in a production system as well.

We then use Table 6's parameters in eqn. (1) to estimate a speedup of 15.7%. The real production speedup is 14% (as determined via A/B testing). Hence, the *Accelerometer*-estimated speedup differs from the real speedup by only 1.7%. We compare `Cache1`'s functionality breakdown with AES-NI in Fig. 16. We observe that AES-NI accelerates the "secure IO" functionality by 73%, saving 12.8% of `Cache1`'s cycles.

Case study 2: Encryption for Cache3. We accelerate encryption in a different microservice, `Cache3`, that is similar to `Cache1` and `Cache2`; we show `Cache3`'s functionality breakdown in Fig. 17. The encryption accelerator is off-chip—the host communicates with the accelerator via a PCIe link. The host offloads the encryption kernel to the accelerator asynchronously, and does not require the accelerator to respond (`Async`). However, after offloading a kernel, the host waits for the accelerator to acknowledge receipt. We use the accelerator's specification sheets to (1) estimate L with fair queuing Q and (2) assume $o_0 = 0$.

In this study, we assume that all encryption offloads will improve speedup, since `Cache3`'s software infrastructure does not support selectively offloading only those granularities that yield speedup. We use parameters defined in Table 6 in equation (6) to estimate speedup. We observe that the PCIe transfer latency is the dominant overhead. After A/B testing, we find that the model overestimates the real speedup by 1.1%.

Table 6. Model parameters used to compare *Accelerometer*-estimated speedup with measured speedup on production systems.

Case Study	C (10^9 cycles)	α	n	o_0 (cycles)	Q (cycles)	L (cycles)	o_1 (cycles)	A	Est. Speedup	Real Speedup
AES-NI	2.0	0.165844	298,951	10	0	3	NA	6	15.7%	14%
Encryption	2.3	0.19154	101,863	0	0	2530	NA	NA	8.6%	7.5%
Inference	2.5	0.52	10	25,000,000	0	NA	12,500	NA	72.39%	68.69%

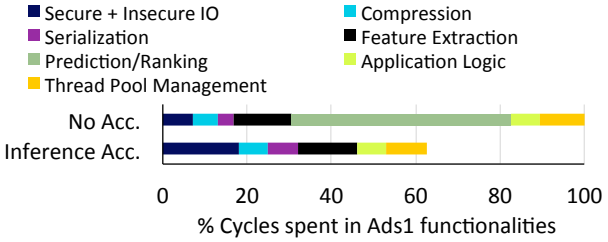

Figure 18. Breakdown of cycles spent in Ads1's functionalities for both the inference unaccelerated & accelerated cases: all inference cycles are freed up.

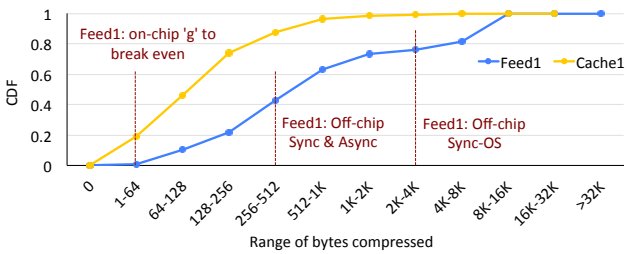

Figure 19. CDF of bytes compressed in Feed1 and Cache1: Feed1 often compresses large granularities.

In Fig. 17, we compare the functionality breakdown of an unaccelerated Cache3 instance with a Cache3 instance that accelerates encryption. We observe that acceleration improves the encryption (secure IO) overhead by 35.7%, improving Cache3's throughput by 7.5%.

Case study 3: Inference for Ads1. We deploy a remote Skylake CPU to perform Ads1's ML inference. We note that the end-to-end service throughput decreases when inference is offloaded to a remote CPU (i.e., $A = 1$). However, we expect the host CPU running Ads1 to incur a speedup, as it no longer does inference locally and uses asynchronous network APIs to offload inference to the remote "accelerator". We validate *Accelerometer* for remote acceleration using this case study.

The host picks up the accelerator's response with a distinct thread (same speedup as Sync-OS with a single thread switching overhead o_1). To estimate o_0, we use a micro-benchmark to measure (1) inference invocation counts and (2) feature vector sizes to estimate I/O overheads from offloading to a remote server. We use a micro-benchmark to measure o_1 using the BPF run queue (scheduler) latency tool [10]. We assume $L + Q = 0$ as the accelerator is remote.

In this study, we carefully batch inference operations and offload them to the remote CPU only when the batch size is large enough to overcome network overheads (as we cannot violate SLO on a production system). Hence, we assume that all of Ads1's inference offloads improve speedup. We use parameters defined in Table 6 in equation (3) (with a single o_1) to estimate speedup. Since Ads1 must invoke many more IO calls to offload inference, it incurs additional IO overheads (o_0). Due to these overheads, we estimate speedup as 72.39%. In reality, remote inference improves Ads1 throughput by 68.69%; our model over-estimates speedup by 3.7%.

In Fig. 18, we illustrate Ads1's functionality breakdown for both the remote inference and local inference cases. Although remote inference consumes additional IO cycles, it completely offloads the inference functionality, freeing up host CPU cycles to perform more work. Note that Ads1 achieves this throughput improvement at the expense of a per-request latency degradation since each request faces an additional ~10 ms network traversal delay; we ensure that the per-request latency meets SLO constraints. This result shows that Ads1's latency can be improved if the remote inference CPU (with $A = 1$) is replaced with an inference accelerator with $A > 1$ to overcome network traversal delays.

5 Applying the Accelerometer Model

We apply the *Accelerometer* model to project speedup for the acceleration recommendations derived from three key common overheads identified by our characterization: compression, memory copy, and memory allocation (see Table 4). We first apply on-chip (Chen et al. [36]) and off-chip (Simek et al. [106]) compression acceleration with Sync, Sync-OS, and Async. We then apply on-chip memory copy (AVX [4]) and allocation acceleration (Kanev et al. [65]); off-chip faces several challenges (e.g., coherence). We apply on-chip offload only with Sync as we only assume CPU core optimizations. We do not see gains from remote acceleration.

We show the model parameters for each acceleration recommendation in Table 7. We assume that all on-chip offloads yield gains as we only consider core optimizations with negligible ($o_0 + L$) overhead. We assume $Q = 0$ in all cases.

Compression. In Fig. 19, we show the compression granularities' CDF for services with high compression overheads—Feed1 and Cache1. Feed1 compresses larger granularities than Cache1; we focus on Feed1 in this study. Since Feed1

Table 7. Parameters used to model speedup and latency reduction for a few acceleration recommendations from Table 4.

Overhead	Acceleration	C (10^9 cycles)	α	n	L (cycles)	o_1 (cycles)	A
Compression	On-chip: Sync	2.3	0.15	15,008	0	NA	5
Compression	Off-chip: Sync	2.3	0.15	9,629	2,300	NA	27
Compression	Off-chip: Sync-OS	2.3	0.15	3,986	2,300	5,750	27
Compression	Off-chip: Async	2.3	0.15	9,769	2,300	NA	27
Memory Copy	On-chip: Sync	2.3	0.1512	1,473,681	0	NA	4
Memory Allocation	On-chip: Sync	2.0	0.055	51,695	0	NA	1.5

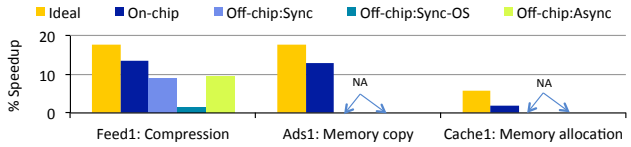

Figure 20. *Accelerometer*-estimated speedup for key overheads we identified: performance bounds from accelerator offload limit achievable speedup.

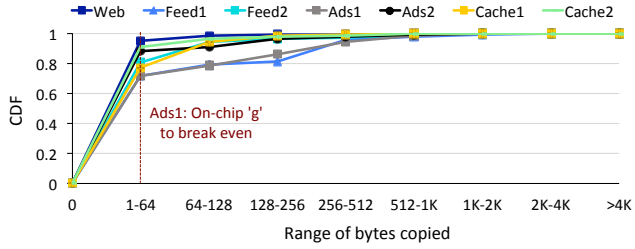

Figure 21. CDF of memory copies across microservices: most microservices frequently copy small granularities.

spends 15% of cycles in compression, it can achieve an ideal speedup of 17.6%, as shown in Fig. 20.

On-chip. We apply Table 7's model parameters in eqn. (2) to find that an offload improves speedup when $g \geq 1$ B; all of Feed1's compressions will improve speedup. We then use $n = 15,008$ in eqn. (1) to estimate a speedup of 13.6% as shown in Fig. 20, implying a latency reduction of 13.6%.

Off-chip. From Table 7 and eqn. (2), we find that a Sync offload improves speedup when $g \geq 425$ B. We note that 64.2% of compressions are ≥ 425 B (Fig. 19). Offloading these compressions improves speedup (and reduces latency) by 9% (Fig. 20). Similarly, Sync-OS and Async offloads yield speedups of 1.6% and 9.6% respectively, reducing latency by 1.4% and 9.2%. Even though on-chip yields a higher speedup, there might be value in off-chip acceleration as it is easier to design than modifying CPUs. For example, off-chip encryption accelerators can be extended to perform compression to leverage improving two kernels for the price of one offload.

Memory Copy. Fig. 21 shows memory copy granularities' CDF across services. We observe that several services often copy < 512 B (smaller than a 4K page). We apply on-chip

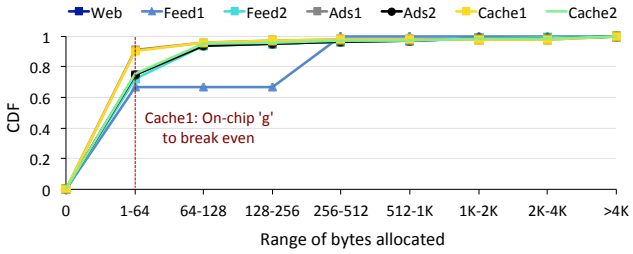

Figure 22. CDF of memory allocations across microservices: most microservices frequently allocate small granularities.

acceleration [4] for Ads1 as it incurs the highest copy overhead. We apply Table 7's parameters in eqn. (1) to project a speedup and latency reduction of 12.7% (Fig. 20). Hence, an on-chip copy optimization [4] can yield significant gains.

Memory Allocation. We show the CDF of memory allocations in Fig. 22. Most microservices perform small allocations (typically < 512 B). We analyze the microservice with the highest memory allocation overhead—Cache1. We find that offloading all of Cache1's 51,695 memory allocations to an on-chip accelerator [65], will result in a 1.86% speedup and latency reduction (Fig. 20).

6 Related Work

We discuss two categories of related work.

Data center overheads. Very few prior works study how cycles are spent in modern data centers. Kanev et al. [63] investigate the "data center tax" or the performance impact of seven types of leaf functions across Google's server fleet. Mars et al. [83–85] use key factors that impact available heterogeneity in CPUs to improve warehouse-scale performance. In contrast, we provide a deep-dive into Facebook's important microservices via leaf function, as well as service functionality breakdowns.

Analytical models. Altaf et al. developed the LogCA [20] model to estimate gains from hardware acceleration. We extend LogCA [20] to support various microservice threading designs to estimate throughput and latency improvements.

Several works develop analytical models for heterogeneous architectures. Chung et al. [37] model custom logic, FPGAs, and GPGPUs. Hempstead et al. [53] propose Navigo to determine accelerator area requirements to maintain

performance trends. Nilakantan et al. [93] estimate communication costs in heterogeneous architectures. Kumar et al. identify performance-efficient data offload granularities. These models use several parameters to accurately determine performance improvements. *Accelerometer* uses a small parameter set to build simple models for microservice speedup and latency reduction.

Several models are architecture-specific [40, 56, 57, 86, 109, 130]. Song et al. [109] predict performance and power tradeoff in GPUs. Hong et al. model GPU execution time [56] and power requirements [57]. Daga et al. [40] discuss communication overheads in APUs and GPUs. Meswani et al. [86] develop models for high performance applications. The *Accelerometer* model abstracts the underlying architecture and can be used across various accelerator types.

Apart from LogCA [20], *Accelerometer*'s simplicity is similar to the Roofline model [79]. Extensions to the Roofline model [80, 94] target specific architectures such as mobile SoCs [55], GPUs [61], vector processing units [103], and FPGAs [39]. While the Roofline model aims to aid programmability, our models seek to expose performance bounds from an accelerator's interface for hyperscale microservices.

7 Conclusion

Modern data centers face performance challenges from supporting diverse microservices in the post Dennard scaling era. We presented a detailed leaf function and service functionality characterization of important microservices used by a leading social media provider—Facebook. We highlighted common overheads and recommended suitable acceleration. To estimate a hardware accelerator's performance early in the design phase, we developed an analytical model, *Accelerometer*, that considers microservice threading designs. We validated *Accelerometer*'s utility via retrospective case studies to show that its estimates match the real production speedup with $\leq 3.7\%$ error. We then used *Accelerometer* to project speedup for the acceleration recommendations derived from key overheads identified by our characterization.

8 Acknowledgement

We acknowledge Thomas F. Wenisch for his insightful suggestions and Vaibhav Gogte for his help in developing *Accelerometer*. We thank Carlos Torres, Pallab Bhattacharya, Jonathan Haslam, Puneet Sharma, Mrinmoy Ghosh, Denis Sheahan, Banit Agrawal, Ikhwan Lee, Amlan Nayak, Jay Kamat, Shobhit Kanaujia, David Cisneros, and Parth Malani who provided valuable insights on Facebook workload characteristics and analysis. We acknowledge Murray Stokely, Vijay Balakrishnan, Hsien-Hsin Lee, Carole-Jean Wu, Udit Gupta, PR Sriraman, Amrit Gopal, Rajee & Akshay Sriraman, Brendan West, Kevin Loughlin, Akanksha Jain, Svilen Kanev, and the anonymous reviewers for their valuable suggestions.

References

[1] [n.d.]. Adopting Microservices at Netflix. https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/.
[2] [n.d.]. Advanced Encryption Standard (AES). https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197.pdf.
[3] [n.d.]. Allocation optimization with different block-sized allocation maps. https://patents.google.com/patent/US5481702A/en.
[4] [n.d.]. AVX. www.wikipedia.org/wiki/Advanced_Vector_Extensions.
[5] [n.d.]. The Biggest Thing Amazon Got Right. www.gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/.
[6] [n.d.]. BPFTrace. https://github.com/iovisor/bpftrace.
[7] [n.d.]. Hidden Costs of Memory Allocation. https://randomascii.wordpress.com/2014/12/10/hidden-costs-of-memory-allocation/.
[8] [n.d.]. Intel DPT with AES-NI & Secure Key. www.intel.com/content/www/us/en/architecture-and-technology/advanced-encryption-standard-aes/data-protection-aes-general-technology.html.
[9] [n.d.]. Intel I/O AT. www.intel.com/content/www/us/en/wireless-network/accel-technology.html.
[10] [n.d.]. Linux bcc/BPF Run Queue (Scheduler) Latency. http://www.brendangregg.com/blog/2016-10-08/linux-bcc-runqlat.html.
[11] [n.d.]. Mcrouter. https://github.com/facebook/mcrouter.
[12] [n.d.]. OpenSSL & SSL/TLS Toolkit. https://www.openssl.org/.
[13] [n.d.]. Scaling Gilt. https://www.infoq.com/presentations/scale-gilt.
[14] [n.d.]. Using tracing at Facebook scale. https://tracingsummit.org/w/images/6/6f/TracingSummit2014-Tracing-at-Facebook-Scale.pdf.
[15] [n.d.]. What is Microservices Architecture? https://smartbear.com/learn/api-design/what-are-microservices/.
[16] [n.d.]. Zero-copy TCP receive. https://lwn.net/Articles/752188/.
[17] Keith Adams, Jason Evans, Bertrand Maher, Guilherme Ottoni, Andrew Paroski, Brett Simmers, Edwin Smith, and Owen Yamauchi. 2014. The hiphop virtual machine. In *Acm Sigplan Notices*.
[18] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. 2015. PIM-enabled instructions: a low-overhead, locality-aware processing-in-memory architecture. In *Int. Symposium on Computer Architecture*.
[19] Amitanand S Aiyer, Mikhail Bautin, Guoqiang Jerry Chen, Pritam Damania, Prakash Khemani, Kannan Muthukkaruppan, Karthik Ranganathan, Nicolas Spiegelberg, Liyin Tang, and Madhuwanti Vaidya. 2012. Storage infrastructure behind Facebook messages: Using HBase at scale. *IEEE Data Eng. Bull.* (2012).
[20] Muhammad Shoaib Bin Altaf and David A. Wood. 2017. LogCA: A High-Level Performance Model for Hardware Accelerators. In *International Symposium on Computer Architecture*.
[21] Jose M Alvarez and Mathieu Salzmann. 2017. Compression-aware training of deep networks. In *Advances in Neural Information Processing Systems*.
[22] Thomas E. Anderson. 1990. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems*.
[23] Aayush Ankit, Izzat El Hajj, Sai Rahul Chalamalasetti, Geoffrey Ndu, Martin Foltin, R Stanley Williams, Paolo Faraboschi, Wen-mei W Hwu, John Paul Strachan, and Kaushik Roy. 2019. PUMA: A programmable ultra-efficient memristor-based accelerator for machine learning inference. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.
[24] Patroklos Argyroudis and Chariton Karamitas. 2012. Exploiting the jemalloc memory allocator: Owning Firefox's heap. *Blackhat USA*.
[25] Grant Ayers, Jung Ho Ahn, Christos Kozyrakis, and Parthasarathy Ranganathan. 2018. Memory Hierarchy for Web Search. In *International Symposium on High Performance Computer Architecture*.
[26] Eytan Bakshy, Solomon Messing, and Lada A Adamic. 2015. Exposure to ideologically diverse news & opinion on Facebook. *Science* (2015).
[27] Matěj Bartík, Sven Ubik, and Pavel Kubalik. 2015. LZ4 compression algorithm on FPGA. In *Electronics, Circuits, and Systems*.
[28] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access

to Privileged CPU Features. In *USENIX Symposium on Operating Systems Design and Implementation*.

[29] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A Protected Dataplane Operating System for High Throughput and Low Latency. In *USENIX Conference on Operating Systems Design and Implementation*.

[30] Emery D Berger, Benjamin G Zorn, and Kathryn S McKinley. 2002. *Reconsidering custom memory allocation*. ACM.

[31] Tom Berson, Drew Dean, Matt Franklin, Diana Smetters, and Michael Spreitzer. 2001. Cryptography as a network service. In *Network and Distributed System Security Symposium*.

[32] Christopher James Blythe, Gennaro A Cuomo, Erik A Daughtrey, and Matt R Hogstrom. 2007. *Dynamic thread pool tuning techniques*. Google Patents.

[33] Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, and Samuel Rash. 2011. Apache Hadoop goes realtime at Facebook. In *International Conference on Management of data*.

[34] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference*.

[35] Josiah L. Carlson. 2013. *Redis in Action*. Manning Shelter Island.

[36] Doris Chen and Deshanand Singh. 2013. Fractal video compression in OpenCL: An evaluation of CPUs, GPUs, and FPGAs as acceleration platforms. In *Design Automation Conference*.

[37] Eric Chung, Peter Milder, James Hoe, and Ken Mai. 2010. Single-chip heterogeneous computing: Does the future include custom logic, FPGAs, and GPGPUs?. In *International symposium on microarchitecture*.

[38] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten Von Eicken. 1993. LogP: Towards a realistic model of parallel computation. In *ACM Sigplan Notices*.

[39] Bruno Da Silva, An Braeken, Erik H D'Hollander, and Abdellah Touhafi. 2013. Performance modeling for FPGAs: extending the roofline model with high-level synthesis tools. *International Journal of Reconfigurable Computing* (2013).

[40] Mayank Daga, Ashwin M Aji, and Wu-chun Feng. 2011. On the efficacy of a fused CPU+ GPU processor (or APU) for parallel computing. In *Application Accelerators in High-Performance Computing*.

[41] Yaozu Dong, Xiaowei Yang, Jianhui Li, Guangdeng Liao, Kun Tian, and Haibing Guan. 2012. High performance network virtualization with SR-IOV. *J. Parallel and Distrib. Comput.* (2012).

[42] Jack Doweck, Wen-Fu Kao, Allen Kuan-yu Lu, Julius Mandelblat, Anirudha Rahatekar, Lihu Rappoport, Efraim Rotem, Ahmad Yasin, and Adi Yoaz. 2017. Inside 6th-generation Intel Core: new microarchitecture code-named Skylake. *IEEE Micro* (2017).

[43] Jose Duato, Antonio J Pena, Federico Silla, Rafael Mayo, and Enrique S Quintana-Orti. 2011. Performance of CUDA virtualized remote GPUs in high performance clusters. In *Parallel Processing*.

[44] Paul Emmerich, Maximilian Pudelko, Simon Bauer, and Georg Carle. 2018. User Space Network Drivers. In *Proceedings of the Applied Networking Research Workshop*.

[45] Hadi Esmaeilzadeh, Emily Blem, Renee Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon & the End of Multicore Scaling. In *International Symposium on Computer Architecture*.

[46] Brad Fitzpatrick. 2004. Distributed Caching with Memcached. *Linux*.

[47] Jeremy Fowers, Joo-Young Kim, Doug Burger, and Scott Hauck. 2015. A scalable high-bandwidth architecture for lossless compression on fpgas. In *Field-Programmable Custom Computing Machines*.

[48] Philip Werner Frey and Gustavo Alonso. 2009. Minimizing the Hidden Cost of RDMA. In *Distributed Computing Systems*.

[49] Vaibhav Gogte, Aasheesh Kolli, Michael J Cafarella, Loris D'Antoni, and Thomas F Wenisch. 2016. HARE: Hardware accelerator for regular expressions. In *International Symposium on Microarchitecture*.

[50] Md Haque, Yong hun Eom, Yuxiong He, Sameh Elnikety, Ricardo Bianchini, and Kathryn McKinley. 2015. Few-to-Many: Incremental Parallelism for Reducing Tail Latency in Interactive Services. In *Architectural Support for Programming Languages and Operating Systems*.

[51] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, and Aditya Kalro. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *International Symposium on High Performance Computer Architecture*.

[52] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, and Joaquin Quiñonero Candela. 2014. Practical Lessons from Predicting Clicks on Ads at Facebook. In *Data Mining for Online Advertising*.

[53] Mark Hempstead, Gu-Yeon Wei, and David Brooks. 2009. Navigo: An early-stage model to study power-contrained architectures & specialization. In *Workshop on Modeling, Benchmarking, and Simulations*.

[54] John L. Henning. 2006. SPEC CPU2006 Benchmark Descriptions. *SIGARCH Comp. Arch. News* (2006).

[55] Mark Hill and Vijay Janapa Reddi. 2019. Gables: A Roofline Model for Mobile SoCs. In *High Performance Computer Architecture*.

[56] Sunpyo Hong and Hyesoon Kim. 2009. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *SIGARCH Computer Architecture News*.

[57] Sunpyo Hong and Hyesoon Kim. 2010. An integrated GPU power and performance model. In *SIGARCH Computer Architecture News*.

[58] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. 2015. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *USENIX-Annual Technical Conference*.

[59] Myeongjae Jeon, Saehoon Kim, Seung-won Hwang, Yuxiong He, Sameh Elnikety, Alan L. Cox, and Scott Rixner. 2014. Predictive Parallelization: Taming Tail Latencies in Web Search. In *Conference on Research and Development in Information Retrieval*.

[60] Eun Young Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems. In *USENIX Conference on Networked Systems Design and Implementation*.

[61] Haipeng Jia, Yunquan Zhang, Guoping Long, Jianliang Xu, Shengen Yan, and Yan Li. 2012. GPURoofline: a model for guiding performance optimizations on GPUs. In *European Conference on Parallel Processing*.

[62] Norman Jouppi, Cliff Young, Nishant Patil, and David et al. Patterson. 2017. In-datacenter performance analysis of a tensor processing unit. In *International Symposium on Computer Architecture*.

[63] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a Warehouse-scale Computer. In *International Symposium on Computer Architecture*.

[64] Svilen Kanev, Kim Hazelwood, Gu-Yeon Wei, and David Brooks. 2014. Tradeoffs between power management & tail latency in warehouse-scale applications. In *Int. Symposium on Workload Characterization*.

[65] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. 2017. Mallacc: Accelerating Memory Allocation. In *Architectural Support for Programming Languages and Operating Systems*.

[66] Rishi Kapoor, George Porter, Malveeka Tewari, Geoffrey M Voelker, and Amin Vahdat. 2012. Chronos: Predictable low latency for data center applications. In *ACM Symposium on Cloud Computing*.

[67] Khubaib, M. Aater Suleman, Milad Hashemi, Chris Wilkerson, and Yale N. Patt. 2012. Morphcore: An energy-efficient microarchitecture for high performance ilp and high throughput tlp. In *International Symposium on Microarchitecture*.

[68] Jungrae Kim, Michael Sullivan, Esha Choukse, and Mattan Erez. 2016. Bit-plane compression: Transforming data for better compression in

many-core architectures. In *International Symposium on Computer Architecture*.

[69] Saehoon Kim, Yuxiong He, Seung-won Hwang, Sameh Elnikety, and Seungjin Choi. 2015. Delayed-Dynamic-Selective (DDS) Prediction for Reducing Extreme Tail Latency in Web Search. In *ACM International Conference on Web Search and Data Mining*.

[70] Taewhan Kim and Jungeun Kim. 2006. Integration of code scheduling, memory allocation, and array binding for memory-access optimization. *Computer-Aided Design of Integrated Circuits and Systems*.

[71] Kathleen Knobe, Joan D. Lukas, and Guy L. Stelle, Jr. 1990. Data Optimization: Allocation of Arrays to Reduce Communication on SIMD Machines. *J. Parallel Distrib. Comput.* (1990).

[72] Monica S Lam. 2012. *A systolic array optimizing compiler*.

[73] Maysam Lavasani, Hari Angepat, and Derek Chiou. 2013. An FPGA-based in-line accelerator for memcached. *Comp. Arch. Letters*.

[74] Doug Lea and Wolfram Gloger. 1996. *A memory allocator*. Unix/mail.

[75] Timothy R Learmont. 2001. *Fine-grained consistency mechanism for optimistic concurrency control using lock groups*. Google Patents.

[76] Chuanpeng Li, Chen Ding, and Kai Shen. 2007. Quantifying the cost of context switch. In *Workshop on Experimental computer science*.

[77] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: A Holistic Approach to Fast In-memory Key-value Storage. In *Networked Systems Design and Implementation*.

[78] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Architectural Support for Programming Languages and Operating Systems*.

[79] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *International Symposium on Computer Architecture*.

[80] Unai Lopez-Novoa, Alexander Mendiburu, and Jose Miguel-Alonso. 2014. A survey of performance modeling and simulation techniques for accelerator-based computing. *Parallel and Distributed Systems* (2014).

[81] Liang Luo, Akshitha Sriraman, Brooke Fugate, Shiliang Hu, Gilles Pokam, Chris J Newburn, and Joseph Devietti. 2016. LASER: Light, Accurate Sharing dEtection and Repair. In *International Symposium on High Performance Computer Architecture*.

[82] Howard Mao, Randy H Katz, and Krste Asanović. 2017. Hardware Acceleration for Memory to Memory Copies. (2017).

[83] Jason Mars. 2012. *Rethinking the architecture of warehouse-scale computers*. Ph.D. Dissertation.

[84] Jason Mars and Lingjia Tang. 2013. Whare-map: heterogeneity in homogeneous warehouse-scale computers. In *International Symposium on Computer Architecture*.

[85] Jason Mars, Lingjia Tang, and Robert Hundt. 2011. Heterogeneity in "homogeneous" warehouse-scale computers: A performance opportunity. *IEEE Computer Architecture Letters* (2011).

[86] Mitesh R Meswani, Laura Carrington, Didem Unat, Allan Snavely, Scott Baden, and Stephen Poole. 2013. Modeling and predicting performance of high performance computing applications on hardware accelerators. *High Performance Computing Applications* (2013).

[87] Maged M. Michael. 2004. Scalable Lock-free Dynamic Memory Allocation. In *Programming Language Design and Implementation*.

[88] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F. Wenisch. 2019. Enhancing Server Efficiency in the Face of Killer Microseconds. In *International Symposium on High Performance Computer Architecture*.

[89] Amirhossein Mirhosseini, Akshitha Sriraman, and Thomas F. Wenisch. 2019. Hiding the Microsecond-Scale Latency of Storage-Class Memories with Duplexity. In *Annual Non-Volative Memories Workshop*.

[90] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. 2016. *Microservice Arch.: Aligning Principles, Practices, & Culture*.

[91] Rolf Neugebauer, Gianni Antichi, José Fernando Zazo, Yury Audzevich, Sergio López-Buedo, and Andrew W Moore. 2018. Understanding PCIe performance for end host networking. In *ACM Special Interest Group on Data Communication*.

[92] Jarek Nieplocha and Jialin Ju. 2000. *ARMCI: A portable aggregate remote memory copy interface*. Citeseer.

[93] Siddharth Nilakantan, Steven Battle, and Mark Hempstead. 2012. Metrics for early-stage modeling of many-accelerator architectures. *IEEE Computer Architecture Letters* (2012).

[94] Cedric Nugteren and Henk Corporaal. 2012. The boat hull model: enabling performance prediction for parallel computing prior to code development. In *Conference on Computing Frontiers*.

[95] Guilherme Ottoni. 2018. HHVM JIT: A Profile-guided, Region-based Compiler for PHP & Hack. In *Programming Language Design and Implementation*.

[96] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A fast, scalable, in-memory time series database. *VLDB Endowment*.

[97] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2016. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems* (2016).

[98] George Prekas, Marios Kogias, and Edouard Bugnion. 2017. ZygOS: Achieving Low Tail Latency for Microsecond-scale Networked Tasks. In *Symposium on Operating Systems Principles*.

[99] Sarunya Pumma, Min Si, Wu-chun Feng, and Pavan Balaji. 2017. Towards scalable deep learning via I/O analysis and optimization. In *High Performance Computing and Communications*.

[100] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N Bhuyan. 2011. Thread reinforcer: Dynamically determining number of threads via OS level monitoring. In *Int. Symposium on Workload Characterization*.

[101] Emilee Rader and Rebecca Gray. 2015. Understanding user beliefs about algorithmic curation in the Facebook news feed. In *ACM conference on human factors in computing systems*.

[102] Jeff Rasley, Brent Stephens, Colin Dixon, Eric Rozner, Wes Felter, Kanak Agarwal, John Carter, and Rodrigo Fonseca. 2014. Planck: Millisecond-scale Monitoring and Control for Commodity Networks. In *ACM Conference on SIGCOMM*.

[103] Yoshiei Sato, Ryuichi Nagaoka, Akihiro Musa, Ryusuke Egawa, Hiroyuki Takizawa, Koki Okabe, and Hiroaki Kobayashi. 2009. Performance tuning and analysis of future vector processors based on the roofline model. In *Workshop on MEmory performance: DEaling with Applications, systems and architecture*.

[104] David Vincent Schuehler. [n.d.]. *Techniques for processing TCP/IP flow content in network switches at gigabit line rates*. Semantic Scholar.

[105] Vivek Seshadri, Yoongu Kim, Chris Fallin, Donghyuk Lee, Rachata Ausavarungnirun, Gennady Pekhimenko, Yixin Luo, Onur Mutlu, Phillip B. Gibbons, Michael A. Kozuch, and Todd C. Mowry. 2013. RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization. In *International Symposium on Microarchitecture*.

[106] Vaclav Simek and Ram Rakesh Asn. 2008. Gpu acceleration of 2d-dwt image compression in matlab with cuda. In *UKSIM European Symposium on Computer Modeling and Simulation*.

[107] Evangelia Sitaridi, Orestis Polychroniou, and Kenneth A Ross. 2016. SIMD-accelerated regular expression matching. In *International Workshop on Data Management on New Hardware*.

[108] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. 2005. Fast hash table lookup using extended bloom filter: an aid to network processing. In *ACM SIGCOMM Computer Comm. Review*.

[109] Shuaiwen Song, Chunyi Su, Barry Rountree, and Kirk W Cameron. 2013. A simplified and accurate model of power-performance efficiency on emergent GPU architectures. In *International Symposium on Parallel and Distributed Processing*.

[110] Akshitha Sriraman. 2019. Unfair Data Centers for Fun and Profit. In *Wild and Crazy Ideas (ASPLOS)*.

[111] Akshitha Sriraman, Abhishek Dhanotia, and Thomas F Wenisch. 2019. SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale. In *The International Symposium on Computer Architecture*.

[112] Akshitha Sriraman, Sihang Liu, Sinan Gunbay, Shan Su, and Thomas F. Wenisch. 2016. Deconstructing the Tail at Scale Effect Across Network Protocols. *The Annual Workshop on Duplicating, Deconstructing, and Debunking* (2016).

[113] Akshitha Sriraman and Thomas F. Wenisch. 2018. μSuite: A Benchmark Suite for Microservices. In *IEEE International Symposium on Workload Characterization*.

[114] Akshitha Sriraman and Thomas F Wenisch. 2018. μTune: Auto-Tuned Threading for OLDI Microservices. In *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*.

[115] Akshitha Sriraman and Thomas F. Wenisch. 2019. Performance-Efficient Notification Paradigms for Disaggregated OLDI Microservices. In *Workshop on Resource Disaggregation*.

[116] Alexei Starovoitov. 2015. BPF in LLVM & kernel. In *Linux Conference*.

[117] Jeffrey Stuecheli, Bart Blaner, CR Johns, and MS Siegel. 2015. CAPI: A coherent accelerator processor interface. *IBM Journal R&D*.

[118] Daniel Stutzbach and Reza Rejaie. 2006. Improving lookup performance over a widely-deployed DHT. In *International Conference on Computer Communications*.

[119] M. Aater Suleman, Moinuddin K. Qureshi, and Yale N. Patt. 2008. Feedback-driven threading: power-efficient and high-performance execution of multi-threaded workloads on CMPs. In *Architectural Support for Programming Languages and Operating Systems*.

[120] Michael B Taylor. 2012. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse. In *Design Automation Conference*.

[121] Xinmin Tian, Hideki Saito, Serguei V Preis, Eric N Garcia, Sergey S Kozhukhov, Matt Masten, Aleksei G Cherkasov, and Nikolay Panchenko. 2013. Practical simd vectorization techniques for intel® xeon phi coprocessors. In *Parallel & Distributed Processing*.

[122] Dan Tsafrir. 2007. The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops). In *Workshop on Experimental computer science*.

[123] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, and Jeremy Hoon. 2012. Tao: how facebook serves the social graph. In *SIG Management of Data*.

[124] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *Computing Colombian Conference*.

[125] Qingyang Wang, Chien-An Lai, Yasuhiko Kanemasa, Shungeng Zhang, and Calton Pu. 2017. A Study of Long-Tail Latency in n-Tier Systems: RPC vs. Asynchronous Invocations. In *International Conference on Distributed Computing Systems*.

[126] Zheng Wang and Michael F.P. O'Boyle. 2009. Mapping Parallelism to Multi-cores: A Machine Learning Based Approach. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*.

[127] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, and Sy et al. Choudhury. 2019. Machine learning at facebook: Understanding inference at the edge. In *High Performance Computer Architecture*.

[128] Owen Yamauchi. 2015. *Hack and HHVM: programming productivity without breaking things*.

[129] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. 2016. Treadmill: Attributing the Source of Tail Latency Through Precise Load Testing and Statistical Inference. In *International Symposium on Computer Architecture*.

[130] Yao Zhang and John D Owens. 2011. A quantitative perf. analysis model for GPU architectures. In *High Perf. Computer Architecture*.

[131] Qin Zhao, Rodric Rabbah, and Weng-Fai Wong. 2005. Dynamic Memory Optimization Using Pool Allocation and Prefetching. *SIGARCH Comput. Archit. News* (2005).

[132] Mark Zuckerberg, Ruchi Sanghvi, Andrew Bosworth, Chris Cox, Aaron Sittig, Chris Hughes, Katie Geminder, and Dan Corson. 2010. *Dynamically providing a news feed about a user of a social network*. Google Patents.

A Artifact Appendix

A.1 Abstract

Our artifact enables running *Accelerometer* to estimate speedup from hardware acceleration. Here, we include links to our source code, and offer tutorials for installing and using *Accelerometer*.

A.2 Artifact check-list (meta-information)

- **Algorithm:** New analytical model, *Accelerometer*, for projecting speedup from hardware acceleration for microservice functionalities.
- **Program:** The *Accelerometer* analytical model built using C++.
- **Compilation:** GCC (Makefile provided).
- **Data set:** Model parameters are to be provided as inputs to the program.
- **Hardware:** One CPU core.
- **Metrics:** The *Accelerometer* program estimates speedup.
- **Output:** Expected output is the *Accelerometer*-estimated speedup from hardware acceleration.
- **Publicly available?:** Yes

A.3 Description

A.3.1 How delivered

All of the source code for *Accelerometer* is open source, and can be obtained via GitHub[2] or Zenodo[3]

A.3.2 Hardware dependencies

Accelerometer requires a single CPU core to run.

A.3.3 Software dependencies

Linux OS with a recent GCC version installed.

A.3.4 Data sets

Model parameters are to be provided as inputs to the *Accelerometer* model.

A.4 Installation

The source code of these components can be found in our GitHub or DOI repository. We also provide a step-by-step tutorial (in our GitHub[1] or Zenodo[2] repositories) to help install and run *Accelerometer*.

A.5 Experiment workflow

There are three steps to run the *Accelerometer* model: (a) identify model parameters for the accelerator under test, (b) input these model parameters into a configuration file, and (c) run the *Accelerometer* model for these model parameters to estimate speedup from acceleration.

[2] https://github.com/akshithasriraman/Accelerometer
[3] https://doi.org/10.5281/zenodo.3612797

We provide the source code of our *Accelerometer* model as well as input configuration files (which were used during *Accelerometer* validation and application) in our repository.

A.6 Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging