SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale

Akshitha Sriraman†*, Abhishek Dhanotia*, Thomas F. Wenisch†
University of Michigan†, Facebook*

akshitha@umich.edu, abhishekd@fb.com, twenisch@umich.edu

ABSTRACT

The variety and complexity of microservices in warehouse-scale data centers has grown precipitously over the last few years to support a growing user base and an evolving product portfolio. Despite accelerating microservice diversity, there is a strong requirement to limit diversity in underlying server hardware to maintain hardware resource fungibility, preserve procurement economies of scale, and curb qualification/test overheads. As such, there is an urgent need for strategies that enable limited server CPU architectures (a.k.a "SKUs") to provide performance and energy efficiency over diverse microservices. 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 profound diversity in OS and I/O interaction, cache misses, memory bandwidth utilization, instruction mix, and CPU stall behavior. Whereas customizing a CPU SKU for each microservice might be beneficial, it is prohibitive. Instead, we argue for "soft SKUs", wherein we exploit coarse-grain (e.g., boot time) configuration knobs to tune the platform for a particular microservice. We develop a tool, μSKU, that automates search over a soft-SKU design space using A/B testing in production and demonstrate how it can obtain statistically significant gains (up to 7.2% and 4.5% performance improvement over stock and production servers, respectively) with no additional hardware requirements.

CCS CONCEPTS
Computer systems organization → Cloud computing

KEYWORDS
Microservice, resource fungibility, soft SKU

ACM Reference Format:
Akshitha Sriraman, Abhishek Dhanotia, Thomas F. Wenisch, 2019. SoftSKU: Optimizing Server Architectures for Microservice Diversity @Scale. In *Proceedings of ISCA '19, Phoenix, AZ, USA, June 22-26, 2019*, 14 pages. https://doi.org/10.1145/3307650.3322227

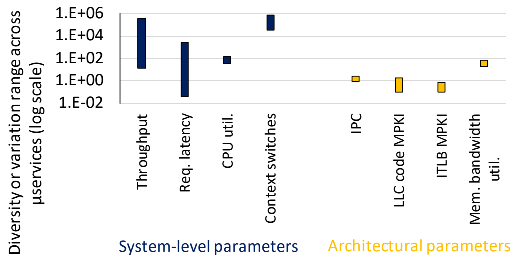

Figure 1: Variation in system-level & architectural traits across microservices: our microservices face extremely diverse bottlenecks.

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 [1]. There is a growing trend towards microservice implementation models [2–6], wherein a complex application is decomposed into distributed microservices [7–10] that each provide specialized functionality [11], such as HTTP connection termination, key-value serving [12], protocol routing [13, 14], or ad serving [15]. This deployment model enables application components' independent scalability by ramping the number of physical servers/cores dedicated to each in response to diurnal and long-term load trends [5].

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 key microservices in four service domains run on hundreds of thousands of servers and occupy a large portion of the compute-optimized installed base. These microservices' importance begs the question: do our existing server platforms serve them well? Are there common bottlenecks across microservices that we might address when selecting a future server CPU architecture?

To this end, we undertake comprehensive system-level and architectural characterizations of these microservices on Facebook production systems serving live traffic. We find that application functionality disaggregation across microservices has yielded enormous diversity in system and CPU architectural requirements, as shown in Fig. 1. For example, caching microservices [16] require intensive I/O and microsecond-scale response latency and frequent OS context switches can comprise 18% of CPU time. In contrast, a `Feed` [17] microservice computes for seconds per request with minimal

OS interaction. Our `Web` [18] microservice entails massive instruction footprints, leading to astonishing instruction cache and ITLB misses and branch mispredictions, while others execute much smaller instruction footprints. Some microservices depend heavily on floating-point performance while others have no floating-point instructions. The microarchitectural trends we discover differ markedly from those of SPEC CPU2006/2017 [19,20], academic cloud workloads [21,22], and even some of Google's major services [1,23].

Such diversity might suggest a strategy to specialize CPU architectures to suit each microservice's distinct needs. Optimizing one or more of these microservices to achieve even single-digit percent speedups can yield immense performance-per-watt benefits. Indeed, we report observations that might inform future hardware designs. However, large-scale internet operators have strong economic incentives to limit hardware platforms' diversity to (1) maintain fungibility of hardware resources, (2) preserve procurement advantages that arise from economies of scale, and (3) limit the overhead of qualifying/testing myriad hardware platforms. As such, there is an immediate need for strategies that enable a limited set of server CPU architectures (often called "SKUs," short for "Stock Keeping Units") to provide performance and energy efficiency over microservices with diverse characteristics.

Rather than diversify the hardware portfolio, we argue for "soft SKUs," a strategy wherein we exploit coarse-grain (e.g., boot time) OS and hardware configuration knobs to tune limited hardware SKUs to better support their presently assigned microservice. Unlike data centers that co-locate services via virtualization, Facebook's microservices run on dedicated bare metal servers, allowing us to easily create microservice-specific soft SKUs. As microservice allocation needs vary, servers can be redeployed to different soft SKUs through reconfiguration and/or reboot. Our OS and CPUs provide several specialization knobs; in this study, we focus on seven: (1) core frequency, (2) uncore frequency, (3) active core count, (4) code vs. data prioritization in the last-level cache ways, (5) hardware prefetcher configuration, (6) use of transparent huge pages, and (7) use of static huge pages.

Identifying the best microservice-specific soft-SKU configuration is challenging: the design space is large, service code evolves quickly, synthetic load tests do not necessarily capture production behavior, and the effects of tuning a particular knob are often small (a few percent performance change). To this end, we develop *μSKU*—a design tool that automates search within the seven-knob soft-SKU design space using A/B testing in production systems on live traffic. *μSKU* automatically varies soft-SKU configuration while collecting numerous fine-grain performance measurements to obtain sufficient statistical confidence to detect even small performance improvements. We evaluate a prototype of *μSKU* and demonstrate that the soft SKUs it designs outperform stock and production server configurations by up to 7.2% and 4.5% respectively, with no additional hardware requirement.

In summary, we contribute:

- A comprehensive characterization of system-level bottlenecks experienced by key production microservices in one of the largest social media platforms today.
- A detailed study of microservices' architectural bottlenecks, highlighting potential design optimizations.
- *μSKU*: A design tool that automatically tunes important configurable server parameters to create microservice-specific "soft" server SKUs on existing hardware.
- A detailed performance study of configurable server parameters tuned by *μSKU*.

The rest of the paper is organized as follows: We describe and measure these seven production microservices' performance traits in Sec. 2. We argue the need for Soft SKUs in Sec. 3. We describe *μSKU*'s design in Sec. 4 and we discuss the methodology used to evaluate *μSKU* in Sec. 5. We evaluate *μSKU* in Sec. 6, discuss limitations in Sec. 7, compare against related work in Sec. 8, and conclude in Sec. 9.

2 Understanding Microservice Performance

We aim to identify software and hardware bottlenecks faced by Facebook's key production microservices to see if they share common bottlenecks that might be addressed in future server CPU architectures. In this section, we (1) describe each microservice, (2) explain our characterization methodology, (3) discuss system-level characteristics to provide insights into how each microservice is operated, (4) report on the architectural characteristics and bottlenecks faced by each microservice, and (5) summarize our characterization's most important conclusions. A key theme that emerges throughout our characterization is *diversity*; the seven microservices differ markedly in their performance constraints' time-scale, instruction mix, cache behavior, CPU utilization, bandwidth requirements, and pipeline bottlenecks. Unfortunately, this diversity calls for sometimes conflicting optimization choices, motivating our pursuit of "soft SKUs" (Section 3) rather than custom hardware for each microservice.

2.1 The Production Microservices

We characterize seven microservices in four diverse service domains running on Facebook's compute-optimized data center fleet. The workloads with longer work-per-request (e.g. Feed2, Ads1) might be called "services" by some readers; we use "microservice" since none of these systems is entirely stand-alone. 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 (JIT) compilation and runtime system for PHP and Hack [18,24,25], to serve web requests originating from end-users. `Web` employs request-level parallelism: an incoming request is assigned to one of a fixed pool of PHP worker threads, which services the request until completion. If all workers are busy, arriving requests are enqueued. `Web` makes frequent requests to other microservices, and the corresponding worker thread blocks waiting on the responses.

Feed1 and Feed2. `Feed1` and `Feed2` are key microservices in our News Feed service. `Feed2` aggregates various leaf microservices' responses into discrete "stories." These stories are then characterized into dense feature vectors by feature extractors and learned models [17,26–28]. The feature vectors are then 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.

Table 1: `Skylake18`, `Skylake20`, `Broadwell16`'s key attributes.

	`Skylake18`	`Skylake20`	`Broadwell16`
Microarchitecture	Intel Skylake	Intel Skylake	Intel Broadwell
Number of sockets	1	2	1
Cores/socket	18	20	16
SMT	2	2	2
Cache block size	64 B	64 B	64 B
L1-I$ (per core)	32 KiB	32 KiB	32 KiB
L1-D$ (per core)	32 KiB	32 KiB	32 KiB
Private L2$ (per core)	1 MiB	1 MiB	256 KiB
Shared LLC (per socket)	24.75 MiB	27 MiB	24 MiB

Ads1 and Ads2. `Ads1` and `Ads2` maintain user-specific and ad-specific data, respectively [15]. 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 (like, e.g., [12, 16, 29, 30]) that reduces throughput requirements of various backing stores. `Cache1` and `Cache2` correspond to two tiers within each geographic region for this service. Client microservices contact the `Cache2` tier. If a request misses in `Cache2`, it is forwarded to the `Cache1` tier. `Cache1` misses are then 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 perform our characterization on 18- and 20-core Intel Skylake processor platforms [31], `Skylake18` and `Skylake20`. Characteristics of each are summarized in Table 1. `Web`, `Feed1`, `Feed2`, `Ads1`, and `Cache2` run on `Skylake18`. `Ads2` and `Cache1` are deployed on `Skylake20`. Both platforms support Intel Resource Director Technology (RDT) [32]. RDT facilitates tunable Last-Level Cache (LLC) size configurations using Cache Allocation Technology (CAT) [33], and allows prioritizing code vs. data in the LLC ways using Code Data Prioritization (CDP) [34].

Experimental setup. We measure each microservice in Facebook's production environment's default deployment—stand-alone with no co-runners on bare metal hardware. Therefore, there are no cross-service contention or interference effects in our data. We measure each system at peak load to stress performance bottlenecks and characterize the system's maximum throughput capabilities. Facebook's production microservice codebases evolve rapidly; we repeat experiments across updates to ensure that results are stable.

We collect most system-level performance data using an internal tool called Operational Data Store (ODS) [35–37]. ODS enables retrieval, processing, and visualization of sampling data collected from all machines in the data center. ODS provides functionality similar to Google-Wide-Profiling [38].

To analyze microservices' interactions with the underlying hardware, we use myriad processor performance counters. We collect data with Intel's EMON [39]—a performance monitoring and profiling tool that time multiplexes sampling of a vast number of processor-specific hardware performance counters with minimal error. For each experiment, we use this tool to collect tens of thousands of hardware performance events. We report 95% confidence intervals on mean results.

Table 2: Avg. request throughput, request latency, & path length across microservices: we observe great diversity across services.

μservice	Throughput (QPS)	Req. latency	Insn./query
Web	O (100)	O (ms)	O (10^6)
Feed1	O (1000)	O (ms)	O (10^9)
Feed2	O (10)	O (s)	O (10^9)
Ads1	O (10)	O (ms)	O (10^9)
Ads2	O (100)	O (ms)	O (10^9)
Cache1	O (100K)	O (μs)	O (10^3)
Cache2	O (100K)	O (μs)	O (10^3)

We contrast our measurements with some CloudSuite [21], SPEC CPU2006 [19], SPEC CPU2017 [20], and Google services [1, 23] where possible. We measured SPEC CPU2006 performance on `Skylake20`. We reproduce selected data from published reports on SPEC CPU2017 [20], CloudSuite [21], and Google's services [1,23] measured on Haswell, Westmere, and Haswell, respectively. These results are not directly comparable with our measurements as they are measured on different hardware. Nevertheless, they provide context for the greater bottleneck diversity we observe in our microservices relative to commonly studied benchmark suites.

We present our characterization in two parts. We first discuss system-level characteristics observed over the entire fleet. We then present performance-counter measurements and their implications on architectural bottlenecks.

2.3 System-Level Characterization

We first present key system-level metrics, such as request latency, achieved throughput, and path length (instructions per query), to provide insight into how the microservices behave and how these traits may impact architectural bottlenecks. Throughout, we call attention to key axes of diversity.

2.3.1 Request throughput, request latency, and path length. We report approximate peak-load throughput, average request latency, and path length (instructions per query) in Table 2. The amount of work per query varies by six orders of magnitude across the microservices, resulting in throughputs ranging from tens of Queries Per Second (QPS) to 100,000s of QPS with average request latencies ranging from tens of microseconds to single-digit seconds.

Microservices' differing time scales imply that per-query overheads that may pose major bottlenecks for some microservices are negligible for others. For example, microsecond-scale overheads that arise from accesses to Flash [40], emerging memory technologies like 3D XPoint by Intel and Micron [41–43], or 40-100 Gb/s Infiniband and Ethernet network interactions [44] can significantly degrade the request latency of microsecond-scale microservices [45–48] like `Cache1` or `Cache2`. However, such microsecond-scale overheads have negligible impact on the request latency of seconds-scale microservices like `Feed2`. The request latency diversity motivates our choice to include several microservices in our detailed performance-counter investigation.

2.3.2 Request latency breakdown. We next characterize request latency in greater detail to determine the relative contribution of computation and queuing/stalls on an average request's end-to-end latency. We report the average fraction of time a request is "running" (executing instruc-

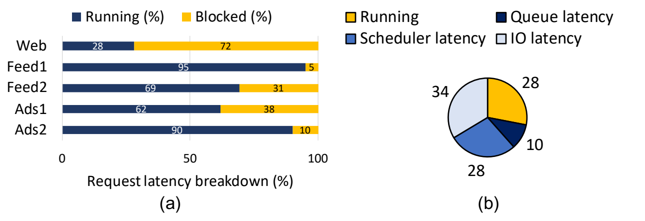

Figure 2: (a) A single request's latency breakdown for each μservice: few μservices block for a long time, (b) Web's request latency breakdown: thread over-subscription causes scheduling delays.

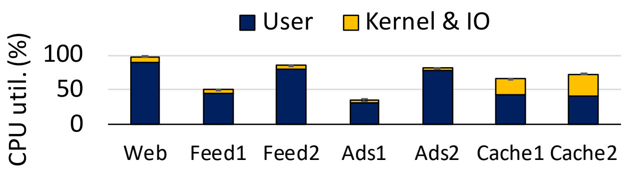

Figure 3: Max. achievable CPU utilization in user- and kernel-mode across μservices: utilization can be low to avoid QoS violations.

tions) vs. "blocked" (stalled, e.g., on I/O) in Fig. 2 (a). We omit Cache1 and Cache2 from this measurement since their queries follow concurrent execution paths and time cannot easily be apportioned as "running" or "blocked".

Feed1 and Ads2 are almost entirely compute-bound throughout a request's life as they are leaves and do not block on requests to other microservices in the common case. They will benefit directly from architectural features that enhance instruction throughput. In contrast, Web, Feed2, and Ads1 emit requests to other microservices and hence their queries spend considerable time blocked. These can benefit from architectural/OS features that support greater concurrency [11, 49], fast thread switching, and better I/O performance [50, 51].

We further break down Web's "blocked" component in Fig. 2 (b) into queuing latency (while a query awaits a worker thread's availability), scheduler latency (where a worker is ready but not running), and I/O latency (where a query is blocked on a request to another microservice). Although Web's scheduler delays are surprisingly high, these delays are not due to inefficient system design, and are instead triggered by thread over-subscription. To improve Web's throughput, load balancing schemes continue spawning worker threads until adding another worker begins degrading throughput.

2.3.3 CPU utilization at peak load. The microservices also vary in their CPU utilization profile. Fig. 3 shows the CPU utilization and its user- and kernel-mode breakdown when each microservice is operated at the maximum load it can sustain without violating Quality of Service (QoS) constraints. We make two observations: (1) CPU resources are not always fully utilized. (2) Most microservices exhibit a relatively small fraction of kernel/IO wait utilization. Each microservice faces latency, quality, and reliability constraints, which impose QoS requirements that in turn impose constraints on how high CPU utilization may rise before a constraint is violated. Our load balancers modulate load to ensure constraints are met. More specifically, Cache1, Cache2, Feed1, Feed2, Ads1, and Ads2 under-utilize the CPU due to strict latency constraints enforced to maintain user experience. These services might benefit from tail latency optimizations, which might allow them to operate at higher CPU utilization. Cache1 and Cache2 exhibit higher kernel-mode utilization due to frequent context switches, which we inspect next.

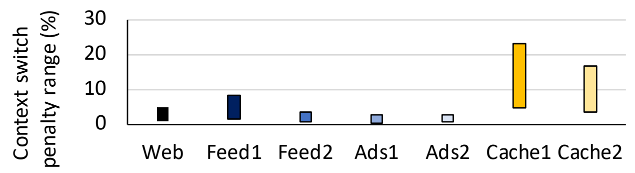

Figure 4: Fraction of a second spent context switching (range): Cache1 & Cache2 can benefit from context switch optimizations.

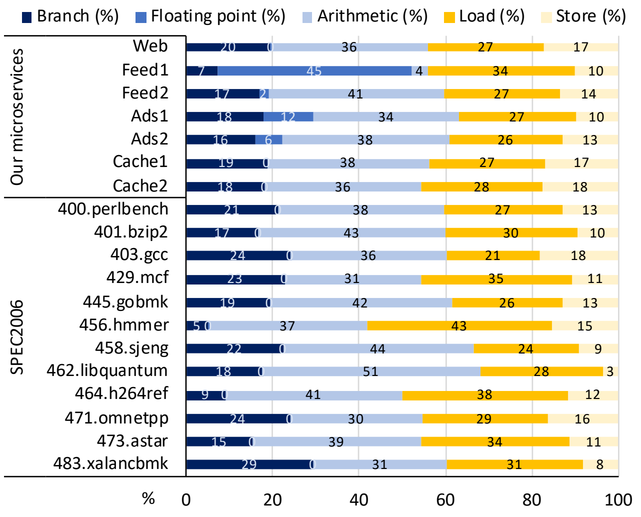

Figure 5: Instruction type breakdown across μservices: instruction mix ratios vary substantially across μservices.

2.3.4 Context switch penalty. We report the fraction of a CPU-second each microservice spends context switching in Fig. 4. We estimate context switch penalty by first aggregating non-voluntary and voluntary context switch counts reported by Linux's time utility. We then estimate upper and lower context switch penalty bounds using switching latencies reported by prior works [52, 53].

Cache1 and Cache2 incur context switches far more frequently than other microservices, and may spend as much as 18% of CPU time in switching. These frequent context switches also lead to worse cache locality, as we will show in our architectural characterization. Software/hardware optimizations [54–62] that reduce context switch latency or counts might considerably improve Cache performance.

2.3.5 Instruction mix. We report our microservices' instruction mix and contrast with SPEC CPU2006 benchmarks in Fig. 5. Instruction mix varies substantially across our microservices, especially with respect to store-intensity and the presence/absence of floating-point operations. The microservices that include ranking models that operate on real-valued feature vectors, Ads1, Ads2, Feed1, and Feed2, all include floating-point operations, and Feed1 is dominated by them. These microservices can likely benefit from optimizations for dense computation, such as SIMD instructions.

Prior work has reported that key-value stores, like Cache1 and Cache2, are typically memory intensive [16]. However,

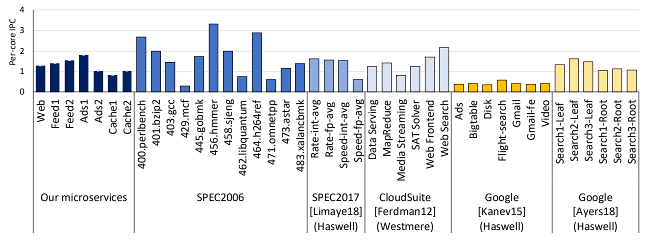

Figure 6: Per-core IPC across our μservices & prior work (IPC measured on other platforms): our μservices have a high IPC diversity.

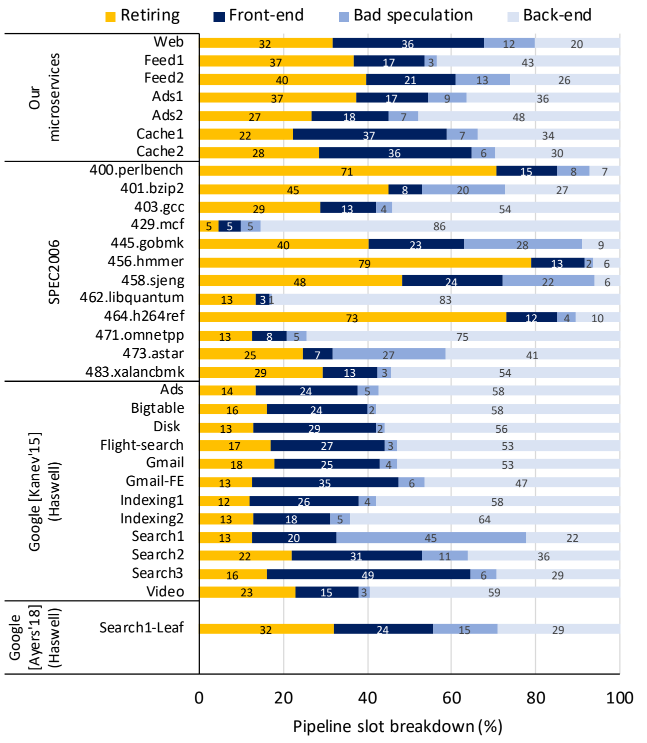

Figure 7: Top-down bottleneck breakdown: several of our microservices face high front-end stalls.

we note that `Cache` requires substantial arithmetic and control flow instructions for parsing requests and marshalling or unmarshalling data; their load-store intensity does not differ from other services as much as the literature might suggest.

2.4 Architectural Characterization

We next turn to performance-counter-based analysis of the architectural bottlenecks of our microservice suite, and examine opportunities it reveals for future hardware SKU design.

2.4.1 IPC and stall causes. We report each microservice's overall Instructions Per Cycle (IPC) in Fig. 6. We contrast our results with IPCs for commonly studied benchmark suites [20, 21] and published results for comparable Google services [1, 23]. Prior works' IPCs are measured on other platforms as shown in Fig. 6; although absolute IPCs may not be directly comparable, it is nevertheless useful to compare variability and spreads.

None of our microservices use more than half of the theoretical execution bandwidth of a Skylake CPU (theoretical peak IPC of 5.0), and `Cache1` uses only 20%. As such, simultaneous multithreading is effective for these services and is enabled in our platforms. Relative to alternative benchmarks, our microservices exhibit (1) a greater IPC diversity than Google's services [1] and (2) a lower IPC than most widely-studied SPEC CPU2006 benchmarks. Given our production workloads' larger codebase, larger working set, and more varied memory access patterns, we do not find our lower typical IPC surprising. When accounting for Skylake's enhanced performance over Haswell, we find the range of IPC values we report to be comparable to the Google services [23].

We provide insight into the root causes of relatively low IPC using the Top-down Microarchitecture Analysis Method (TMAM) [63] to categorize processor pipelines' execution stalls, as reported in Fig. 7. TMAM exposes architectural bottlenecks despite the many latency-masking optimizations of modern out-of-order processors. The methodology reports bottlenecks in terms of "instruction slots"—the fraction of the peak retirement bandwidth that is lost due to stalls each cycle. Slots are categorized as: *front-end* stalls due to instruction fetch misses, *back-end* stalls due to pipeline dependencies and load misses, *bad speculation* due to recovery from branch mispredictions, and *retiring* of useful work.

As suggested by the IPC results, our microservices retire instructions in only 22%-40% of possible retirement slots. However, the nature of the stalls in our applications varies substantially across microservices and differs markedly from the other suites. We make several observations.

First, our microservices tend to have greater front-end stalls than SPEC workloads. In particular, `Web`, `Cache1`, and `Cache2` lose $\sim$37% of retirement slots due to front-end stalls; only Google's `Gmail-FE` and `search` exhibit comparable front-end stalls. In `Web`, front-end stalls arise due to its enormous code footprint due to a rich feature set and the many URL endpoints it implements. In `Cache`, frequent context switches and OS activity cause high front-end stalls. As we will show, these microservices could benefit from larger I-cache and ITLB and other techniques that address instruction misses [64, 65]. In contrast, microservices like `Ads1`, `Ads2`, or `Feed1` do not stand to gain much from greater instruction capacity, leading to conflicting SKU optimization goals.

Second, mispredicted branches make up $3\% - 13\%$ of wasted slots. Branch mispredictions are more rare in data-crunching microservices like `Feed1` and more common when instruction footprint is large, as in `Web`, where aliasing in the Branch Target Buffer contributes a large fraction of branch misspeculations. SKU optimization goals diverge, with some microservices calling for simple branch predictors while others call for higher capacity and more sophisticated prediction.

Third, back-end stalls, largely due to data cache misses, occupy up to 48% of slots, implying that several microservices can benefit from memory hierarchy enhancements. However, microservices like `Web` or `Feed2`, which have fewer back-end stalls, likely gain more from chip area/power dedicated to additional computation resources rather than cache.

2.4.2 Cache misses. We provide greater nuance to our front-end and back-end stall breakdown by measuring instruction and data misses in the cache hierarchy. We present code and data Misses Per Kilo Instruction (MPKI) across all cache levels—L1, L2, and LLC in Figs. 8 and 9, to analyze the overall effectiveness of each cache level. We also show

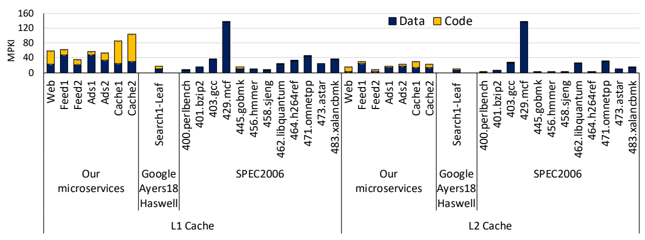
Figure 8: L1 & L2 code & data MPKI: our microservices typically have higher L1 MPKI than comparison applications.

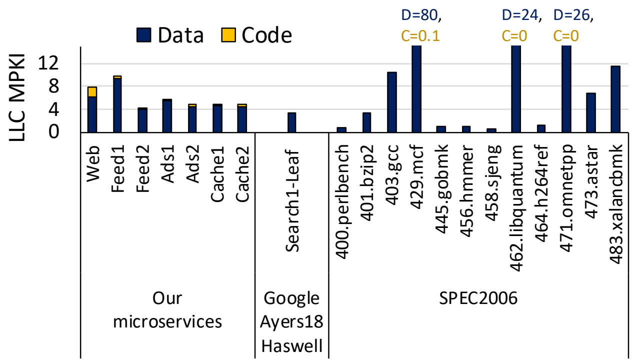

Figure 9: LLC code & data MPKI: LLC data MPKI is high across microservices and Web incurs a high code LLC MPKI.

cache MPKI reported by prior work [23] for Google search and our measurements of SPEC CPU2006 on `Skylake20`.

We make the following observations: (1) Our L1 MPKI are drastically higher than the comparison applications, especially for code, and particularly for `Cache1` and `Cache2`. (2) LLC data misses are commonly high in all microservices, especially in `Feed1`, which traverses large data structures. (3) `Web` incurs 1.7 LLC instruction MPKI. These misses are quite computationally expensive, since out-of-order mechanisms do not hide instruction stalls. It is unusual for applications to incur non-negligible LLC instruction misses at all in steady state; few such applications are reported in the academic literature.

Prior works [1, 21, 23, 66] typically find current LLC sizes to be sufficient to encompass server applications' entire code footprint. In `Web`, the large code footprint and high instruction miss rates arise due to the large code cache, frequent JIT code generation, and a large and complex control flow graph. `Cache1` and `Cache2` incur frequent context switches (see Fig. 4) among distinct thread pools executing different code, which leads to code thrashing in L1 and, to a lesser degree, L2. We conclude many microservices can benefit from larger I-caches, instruction prefetching, or prioritizing code over data in the LLC using techniques like Intel's CDP [34, 67].

2.4.3 LLC capacity sensitivity. Using CAT [34], we inspect sensitivity to LLC capacity. We vary capacity by enabling LLC ways two at a time, up to the maximum of 11 ways. We report LLC MPKI broken down by code and data in Fig. 10. We omit `Cache` as it fails to meet QoS constraints with reduced LLC capacity. For most microservices, a knee (8 ways) emerges where the LLC is large enough to capture a primary working set without degrading IPC, and further

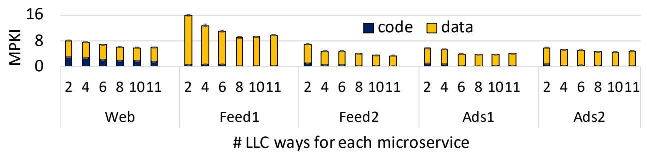
Figure 10: LLC code and data MPKI vs. LLC size: some microservices may benefit from trading LLC capacity for more cores.

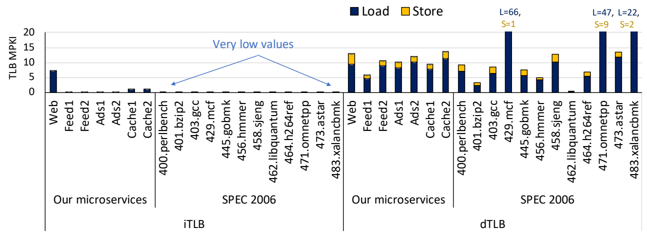

Figure 11: ITLB & DTLB (load & store) MPKI breakdown: some microservices can benefit from huge page support.

capacity increases provide diminishing returns. For some microservices (e.g., `Ads2` and `Feed1`), the largest working set is too large to be captured. Hence, some services might benefit from trading LLC capacity for additional cores [68].

2.4.4 TLB misses. We report instruction and data TLB MPKI in Fig. 11. For the DTLB, we break down misses due to loads and stores. The ITLB miss trends mirror our LLC code miss observations: `Web`, `Cache1`, and `Cache2` incur substantial ITLB misses, while the miss rates are negligible for the remaining microservices. The drastically higher miss rate in `Web` illustrates the impact of its large JIT code cache.

DTLB miss rates are more variable across microservices. They typically follow the LLC MPKI trends shown in Fig. 9 with the exception of `Feed1`—despite a relatively high LLC MPKI of 9.3 it incurs a relatively low DTLB MPKI of 5.8. `Feed1`'s main data structures are dense floating-point feature vectors and model weights, leading to good page locality despite a high LLC MPKI. However, the other microservices might benefit from software (like static or transparent huge pages) and hardware (e.g., [69–75]) paging optimizations.

2.4.5 Memory bandwidth utilization. We inspect memory bandwidth utilization and its attendant effects on latency due to memory system queuing for each microservice in Fig. 12. We first characterize the inherent bandwidth vs. latency trade-off of our two platforms—`Skylake18` in the blue dots and `Skylake20` in the yellow crosses—using a memory stress test [76]. These curves show the characteristic horizontal asymptote at the unloaded memory latency and then exponential latency growth as memory system load approaches saturation. We then plot each microservice's measured average latency and bandwidth, using dots and crosses, respectively, to indicate the service platform.

Microservices like `Web` or `Feed1` have high memory bandwidth utilization relative to the platform capability. Nevertheless, our microservices cannot push memory bandwidth utilization above a certain threshold—operating at higher bandwidth causes exponential memory latency increase, triggering service latency violations. `Ads1` and `Ads2` operate at higher latency than the characteristic curve predicts due to memory

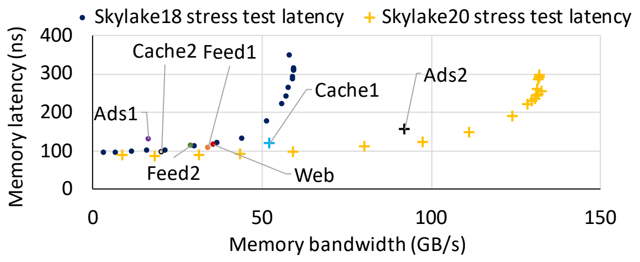

Figure 12: Memory bandwidth vs. latency: microservices under-utilize memory bandwidth to avoid latency penalties.

traffic burstiness. The curves also reveal why it is necessary to run `Cache1` and `Ads2` on the higher-peak-bandwidth `Skylake20` platform to keep memory latency low. Nevertheless, several microservices under-utilize available bandwidth, and hence might benefit from optimizations that trade bandwidth to improve latency, such as hardware prefetching [77].

We summarize our findings in Table 3.

3 "Soft" SKUs

Our microservices exhibit profound diversity in system-level and architectural traits. For example, we demonstrated diverse OS and I/O interaction, code/data cache miss ratios, memory bandwidth utilization, instruction mix ratios, and CPU stall behavior. One way to address such distinct bottlenecks is to specialize CPU architectures by building custom hardware server SKUs to suit each service's needs. However, such hardware SKU diversity is impractical, as it requires testing and qualifying each distinct SKU and careful capacity planning to provision each to match projected load. Given the uncertainties inherent in projecting customer demand, investing in diverse hardware SKUs is not effective at scale.

Data center operators aim to maintain hardware resource fungibility to preserve procurement advantages that arise from economies of scale and limit the effort of qualifying myriad hardware platforms. To preserve fungibility, we seek strategies that enable a few server SKUs to provide performance and energy efficiency over diverse microservices. To this end, we propose exploiting coarse-grain (e.g., boot time) parameters to create "soft SKUs", tuning limited hardware SKUs to better support their assigned microservice. However, manually identifying microservice-specific soft-SKUs is impractical since the design space is large, code evolves quickly, synthetic load tests do not necessarily capture production behavior, and the effects of tuning a single knob are often small (a few percent performance change). Hence, we build an automated design tool—*µSKU*—that searches the configuration design space to optimize for each microservice.

4 μSKU: System Design

µSKU is a design tool for quick discovery of performant and efficient "soft" SKUs. *µSKU* automatically varies configurable server parameters, or "knobs," by searching within a predefined design space via A/B testing. A/B testing is the process of comparing two identical systems that differ only in a single variable. *µSKU* conducts A/B tests by comparing the performance of two identical servers (i.e., same hardware platform, same fleet, and facing the same load) that differ only in their knob configuration. *µSKU* collects copious

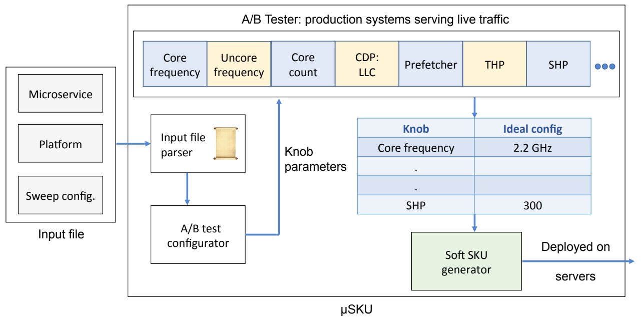

Figure 13: *µSKU*: system design

fine-grain performance measurements while conducting automated A/B tests on production systems serving live traffic to search for statistically significant performance changes. We aim to ensure that *µSKU* has a simple design so that it can be applied across microservices and hardware SKU generations while avoiding operational complexity. Key design challenges include: (1) identifying performance-efficient soft-SKU configurations in a large design space, (2) dealing with frequent code evolution, (3) capturing behavior in production systems facing diurnal or transient load fluctuations, and (4) differentiating actual performance variations from noise through appropriate statistical tests. We discuss how *µSKU*'s design meets these challenges.

We develop a *µSKU* prototype that explores a soft-SKU design space comprising seven configurable server knobs. *µSKU* accepts a few input parameters and then invokes its components—A/B test configurator, A/B tester, and soft SKU generator, as shown in Fig. 13. We describe each component below.

Input file. The user provides an input file with the following three input parameters.

(1) *Target Microservice.* Several aspects of *µSKU*'s behavior must be tuned for the specific target microservice. *µSKU* reboots the server while performing certain A/B tests (e.g., core count scaling). Some microservices may not tolerate reboots on live traffic and hence *µSKU* disables these knobs in such cases. Furthermore, *µSKU* disables knobs that do not apply to a microservice. For example, Statically-allocated Huge Pages (SHPs) are inapplicable to `Ads1`, since it does not use the APIs to allocate them. Our current *µSKU* prototype estimates performance by measuring the Millions of Instructions per Second (MIPS) rate via EMON [39], which we have confirmed is proportional to several key microservices' throughput (e.g., `Web` and `Ads1`). However, we anticipate the performance metric that *µSKU* measures to determine whether a particular soft SKU has improved performance to be microservice specific. In particular, MIPS may be insufficient to measure `Cache`'s throughput, since `Cache`'s code is introspective of performance. (It executes exception handlers when faced with knob configurations that engender QoS violations, which make instructions-per-query vary with performance.) *µSKU* can be extended to perform A/B tests using microservice-specific performance metrics.

(2) *Processor platform.* The available settings in several *µSKU* design space dimensions, such as specific core and uncore frequencies, core counts, and hardware prefetcher options, are hardware platform specific.

Table 3: Summary of findings and suggestions for future optimizations.

Finding	Opportunity
Diversity among microservices (§2.3, §2.4)	"Soft" SKUs
Some μservices are compute-intensive (§2.3.2)	Enhance instruction throughput (e.g., more cores, wider SMT, etc.)
Some μservices emit frequent requests (§2.3.2)	Features that support greater concurrency, fast thread switching, and faster I/O
CPU under-utilization due to QoS constraints (§2.3.3)	Mechanisms to reduce tail latency, enabling higher utilization
High context switch penalty (§2.3.4)	Coalesce I/O, user-space drivers, vDSO, in-line accelerators, thread pool tuning
Substantial floating-point operations (§2.3.5)	Optimizations for dense computation (e.g., SIMD)
Large front-end stalls & code footprint (§2.4.1-2)	AutoFDO, large I-cache, CDP, prefetchers, ITLB optimizations, better decode
Branch mispredictions (§2.4.1)	"Wider" hardware branch predictors, sophisticated prediction algorithms
Low data LLC capacity utilization (§2.4.1-3, §2.4.5)	Trade-off LLC capacity for additional cores
Low memory bandwidth util. (§2.4.5)	Optimizations that trade bandwidth for latency (e.g., prefetching)

(3) *Sweep configuration.* μSKU's A/B tester measures the performance implications of sweeping server knobs either (1) *independently*, where individual knobs are scaled one-by-one and their effects are presumed to be additive when creating a soft SKU, or (2) *exhaustively*, where the design space sweep explores the cross product of knob settings. Note that some microservices receive code updates so frequently (O(hours)) that an *exhaustive* μSKU sweep cannot be completed between code pushes. In practice, the gains from μSKU's knobs are not strictly additive. Nevertheless, the knobs do not typically co-vary strongly, so we have had success in tuning knobs *independently*, as the exhaustive approach requires an impractically large number of A/B tests.

A/B test configurator. The A/B test configurator sets up the automatic A/B test environment by specifying the sweep configuration and knobs to be studied.

A/B tester. The A/B tester is responsible for independently or exhaustively varying configurable hardware and OS knobs to measure ensuing performance changes. Our μSKU prototype varies seven knobs (suggested by our earlier characterization), but can be extended easily to support more. It varies (1) core frequency, (2) uncore frequency, (3) core count, (4) CDP in the LLC ways, (5) prefetchers, (6) Transparent Huge Pages (THP), and (7) SHPs.

The A/B tester sweeps the design space specified by the A/B test configurator. For each point in the space, the tester suitably sets knobs and then launches a hardware performance counter-based profiling tool [39] to collect performance observations. For each knob configuration, the A/B tester first discards observations during a warm-up phase that typically lasts for a few minutes to avoid cold start bias [78]. Next, the A/B tester records performance counter samples via EMON [39] with sufficient spacing to ensure independence. Finally, when the desired 95% statistical confidence is achieved, the A/B tester outputs mean estimates, which it records in a design space map. It then proceeds to the next knob configuration. The A/B tester typically achieves 95% confidence estimates with tens of thousands of performance counter samples (minutes to hours of measurement). If 95% confidence is not reached after collecting $\sim 30,000$ observations, μSKU concludes there is no statistically significant performance difference and proceeds to the next knob configuration. The final design space map helps identify (with a 95% confidence) the most performant knob configurations.

Soft SKU generator. The A/B tester's design space map is fed to the soft SKU generator, which selects the most performant knob configurations. It then applies this configuration to live servers running the microservice. Once the selected soft SKU is deployed, μSKU performs further A/B tests by comparing the QPS achieved (via ODS) by soft-SKU servers against hand-tuned production servers for prolonged durations (including across code updates and under diurnal load) to validate that the soft SKU offers a stable advantage.

5 Methodology

We discuss the methodology we use to evaluate μSKU.

Microservices. We focus our prototype μSKU evaluation on the `Web` service on two generations of hardware platforms and on the `Ads1` microservice on a single platform. These two microservices differ drastically in our characterization results while both being amenable to the use of MIPS rate as a performance metric. Moreover, the surrounding infrastructure for these services is sufficiently robust to tolerate failures and disruptions we might cause with the μSKU prototype, allowing us to experiment on production traffic.

Hardware platforms. To evaluate μSKU, we run `Web` on two hardware platforms—`Broadwell16` and `Skylake18`, and `Ads1` on `Skylake18` (see Table 1). We evaluate `Web` on both `Skylake18` and `Broadwell16` to analyze the configurable server knobs' sensitivity to the underlying hardware platform. Henceforth, we refer to `Web` running on `Skylake18` as `Web` (Skylake) and `Broadwell16` as `Web` (Broadwell).

Experimental setup. We compare μSKU's A/B test knob scaling studies against default production server knob configurations. Some default knob configurations arise from arduous manual tuning, and therefore differ from stock server configurations. We next describe how μSKU implements A/B test scaling studies for each configurable knob.

(1) *Core frequency.* Our servers enable Intel's Turbo Boost technology [79]. μSKU scales core frequency from 1.6 GHz to 2.2 GHz (default) by overriding core frequency-controlling Model-Specific Registers (MSRs).

(2) *Uncore frequency.* μSKU varies uncore (LLC, memory controller, etc.) frequency from 1.4 GHz to 1.8 GHz (default) by overriding uncore frequency-controlling MSRs [80].

(3) *Core count.* μSKU scales core count from 2 physical cores to the platform-specific maximum (default), by directing the boot loader to incorporate the `isolcpus` flag [81] specifying cores on which the OS may not schedule. μSKU then reboots the server to operate with the new core count.

(4) *LLC Code Data Prioritization.* μSKU uses Intel RDT [34] to prioritize code vs. data in the LLC ways. Our servers' OS kernels have extensions that support Intel RDT via the `Resctrl` interface [82]. μSKU leverages these kernel extensions to vary CDP from one dedicated LLC way for data and the

rest for code, to one dedicated way for code and the rest for data. Default production servers share LLC ways between code and data without CDP prioritization.

(5) *Prefetcher.* Our servers support four prefetchers [83]: (a) *L2 hardware prefetcher* that fetches lines into the L2 cache, (b) *L2 adjacent cache line prefetcher* that fetches a cache line in the same 128-byte-aligned region as a requested line, (c) *DCU prefetcher* that fetches the next cache line into L1-D cache, and (d) *DCU IP prefetcher* that uses sequential load history to determine whether to prefetch additional lines. *µSKU* considers five configurations: (a) all prefetchers off, (b) all prefetchers on (default on Web (Skylake) and Ads1), (c) only *DCU prefetcher* and *DCU IP prefetcher* on, (d) only *DCU prefetcher* on, and (e) only *L2 hardware prefetcher* and *DCU prefetcher* on (default on Web (Broadwell)). *µSKU* adjusts prefetcher settings via MSRs.

(6) *Transparent Huge Pages (THP)*: THP is a Linux kernel mechanism that automatically backs virtual memory allocations with huge pages (2MB or 1GB) when contiguous physical memory is available and defragments memory in the background to coalesce free space [84]. *µSKU* considers three THP configurations (a) *madvise*—THP is enabled only for memory regions that explicitly request huge pages (default), (b) *always ON*—THP is enabled for all pages, and (c) *always OFF*—THP is not used even if requested. *µSKU* configures THP by writing to kernel configuration files.

(7) *Statically-allocated Huge Pages (SHP)*: SHPs are huge pages (2MB or 1GB) reserved explicitly by the kernel at boot time and must be explicitly requested by an application. Once reserved, SHP memory can not be repurposed. *µSKU* varies SHP counts from 0 to 600 in 100-step increments by modifying kernel parameters [85]. *µSKU* can be extended to conduct a binary search to identify optimal SHP counts.

Performance metric. *µSKU* estimates performance in terms of throughput by measuring MIPS rate via EMON [39]. We have verified that MIPS is proportional to Web and Ads1's throughput (QPS). We do not measure QPS directly as QPS reported by ODS is not sufficiently fine-grained. We aim to eventually have *µSKU* replace tedious manual knob tuning for each microservice. Hence, we evaluate *µSKU*-generated soft SKUs against (a) stock off-the-shelf and (b) hand-tuned production server configurations.

6 Evaluation

We first present *µSKU*'s A/B test results for all seven configurable server knobs. We then compare the throughput of "soft" server SKUs that *µSKU* discovers against (a) hand-tuned production and (b) stock server configurations.

6.1 Knob Characterization

We present *µSKU*'s A/B test results for each knob and compare it against the current production configuration, indicated by thick red bar/point outlines or red axis lines in our graphs. For each graph, we report mean throughput and 95% confidence intervals under peak-load production traffic. For the first three knobs, we find that *µSKU* matches expert manual tuning decisions. However, for the next four knobs, *µSKU* identifies configurations that outperform production settings.

(1) Core frequency. We illustrate *µSKU*'s core frequency scaling analysis in Fig. 14 (a). *µSKU* varies core frequency

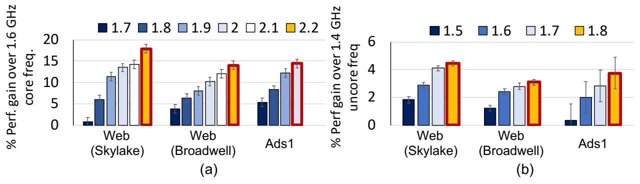

Figure 14: Perf. trend with (a) core frequency scaling, (b) uncore frequency scaling: the max. frequency offers the best performance.

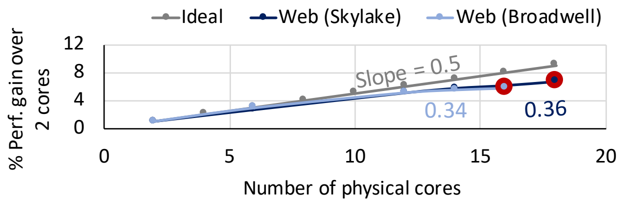

Figure 15: Perf. trend with core count scaling: Web is core-bound.

from 1.6 GHz to 2.2 GHz. We report relative throughput (MIPS) gains over cores operating at 1.6 GHz. Our production systems have a fixed CPU power budget that is shared between the core and uncore (e.g., LLC, memory and QPI controller, etc.) CPU components. The current production configuration enables Turbo Boost [79] and runs Web (Skylake and Broadwell) at 2.2 GHz and Ads1 at 2.0 GHz (as indicated by the thick red bar outlines in Fig. 14 (a)). Ads1 must operate at slightly lower frequency because its use of AVX operations consumes part of the CPU power budget.

µSKU aims to (1) identify whether there is a minimum core frequency knee below which throughput degrades rapidly and (2) diagnose if core frequency trends suggest that the microservice may be uncore bound. Web's and Ads1's throughputs increase precipitously from 1.6 GHz to 1.9 GHz, beyond which *µSKU* reports continued but diminishing throughput gains. These microservices are all sensitive to core frequency, hence, operating at the maximum and enabling Turbo Boost are sensible tuning decisions. *µSKU* configures soft SKUs that operate at 2.2 GHz core frequency for Web (Skylake and Broadwell) and 2.0 GHz for Ads1, matching experts' tuning.

(2) Uncore frequency. *µSKU* varies the frequency of uncore CPU power domain (including LLC, QPI controller, and memory controller), from 1.4 GHz to 1.8 GHz. We report results normalized to 1.4 GHz uncore frequency (Fig. 14 (b)). Our default production configuration runs both microservices at 1.8 GHz uncore frequency. Uncore frequency indicates the degree to which applications are sensitive to access latency when memory and core execution bandwidth are held constant. Both of these microservices are sensitive to memory latency, though the sensitivity is greater in Ads1. As with core frequency, *µSKU* selects soft SKUs that operate at the maximum 1.8 GHz for both microservices, again matching the default production configuration.

(3) Core count. We present *µSKU*'s core count scaling results in Fig. 15, where we report throughput gain relative to execution on only two physical cores. The grey line indicates ideal linear scaling. *µSKU* scales Web (Skylake) to its maximum core count (18 cores) and Web (Broadwell) to its maximum (16). We exclude Ads1 from Fig. 15 since its load balancing design precludes *µSKU* from meeting QoS

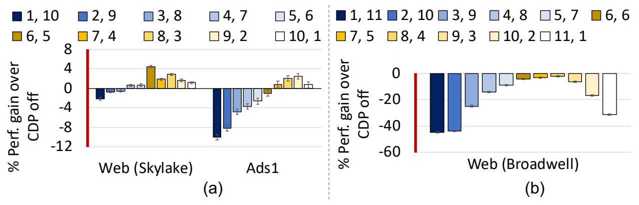

Figure 16: Perf. trend with CDP scaling: (a) Web (Skylake) & Ads1 benefit due to lower code MPKI (b) Web (Broadwell) has no gains.

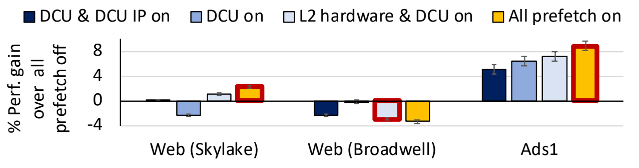

Figure 17: Perf. trends with varied prefetcher config.: turning off prefetchers can improve bandwidth utilization in Web (Broadwell).

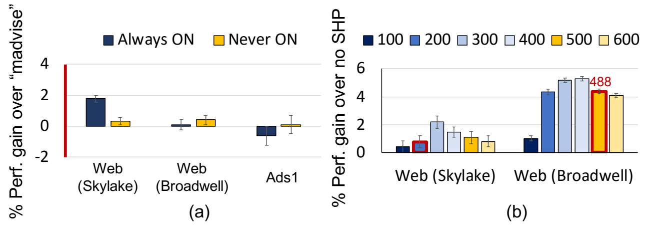

Figure 18: Perf. trends with varied (a) THP: Web (Skylake) benefits from THP ON, (b) SHP: there is a sweet spot in optimal SHP count.

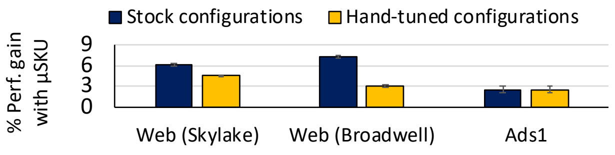

Figure 19: Perf. gain with *μSKU* over stock and hand-tuned servers: *μSKU* outperforms even hand-tuned production servers.

constraints with fewer cores. *μSKU* observes that Web's performance scales almost linearly up to ~8 physical cores. As core count increases further, interference in the LLC causes the scaling curve to bend down. As with frequency, the best soft SKU selected by *μSKU* operates with all available cores.

(4) Code Data Prioritization (CDP) in LLC ways. In our earlier characterization (Fig. 9), we noted that Web exhibits a surprising number of off-chip code misses. Hence, *μSKU* considers prioritizing code vs. data in the LLC ways. We report throughput gains over the production baseline (where CDP is not used and code and data share LLC ways) for Web (Skylake) and Ads1 in Fig. 16(a) and Web (Broadwell) in Fig. 16(b). Skylake18 and Broadwell16 have 11 and 12 LLC ways, respectively. We label each bar with {LLC ways dedicated to data, LLC ways dedicated to code}.

Here we find that Web (Skylake) achieves up to 4.5% mean throughput gain with 6 LLC ways dedicated to data and 5 LLC ways dedicated to code, a configuration that degrades LLC data misses by 0.60 MPKI but improves code misses by 0.30 MPKI. Although this configuration increases net LLC misses by almost 0.30 MPKI, it still results in a performance win because the latency of code misses is not hidden and they incur a greater penalty. Similarly, Ads1 achieves 2.5% mean throughput improvement with 9 LLC ways dedicated to data and 2 LLC ways dedicated to code, sacrificing 0.20 LLC data MPKI to improve LLC code MPKI by 0.06. *μSKU* observes no throughput improvement in Web (Broadwell) since it saturates memory bandwidth under all CDP configurations. Hence, *μSKU* can not trade-off increasing the net LLC MPKI to reduce LLC code misses. *μSKU* selects soft server SKUs for Web (Skylake) and Ads1 such that they dedicate {6, 5} and {9, 2} LLC ways for data and code, respectively, improving over the present-day hand-tuned production configuration. *μSKU* does not enable CDP in Web's (Broadwell) soft SKU.

(5) Prefetcher. We report *μSKU*'s results for prefetcher tuning in Fig. 17. Our production systems enable (1) all prefetchers on Web (Skylake) and Ads1 and (2) only the L2 hardware prefetcher and DCU prefetcher on Web (Broadwell). On Web (Broadwell), *μSKU* reveals a ~ 3% mean throughput win over the production configuration when all prefetchers are turned off. Web (Broadwell) is heavily memory bandwidth bound when prefetchers are turned on, unlike Web (Skylake) and Ads1. Turning off prefetchers reduces memory bandwidth pressure, enabling overall throughput gains. In contrast, Web (Skylake) and Ads1 are not memory bandwidth bound, and hence do not benefit from turning off prefetchers.

(6) Transparent Huge Pages (THPs). In our earlier characterization (see Fig. 11), we found that Web suffers from significant ITLB and DTLB misses. Hence, *μSKU* explores huge page settings to reduce TLB miss rates. The default THP setting on our production servers is *madvise*, where THP is enabled only for memory regions that explicitly request it. In Fig. 18(a), *μSKU* considers (1) always enabling huge pages (*always ON*) and (2) disabling huge pages even when requested (*never ON*), and compares with the default (baseline for the graph) *madvise* configuration.

μSKU identifies a mean 1.87% throughput gain on Web (Skylake) when THP is *always ON*, as it significantly reduces TLB misses compared to *madvise*. However, the *always ON* setting does not enhance Ads1 and Web (Broadwell)'s throughput as their TLB miss rates do not improve. Throughput achieved with the *never ON* configuration is comparable with *madvise*, as few allocations use the *madvise* hint.

(7) Statically-allocated Huge Pages (SHPs). We report *μSKU*'s SHP sweep results in Fig. 18(b). *μSKU* excludes Ads1 from this study as it makes no use of SHPs. Our production systems reserve 200 SHPs for Web (Skylake) and 488 SHPs for Web (Broadwell). *μSKU* shows that reserving 300 SHPs on Web (Skylake) and 400 SHPs on Web (Broadwell) can outperform our production systems by 1.4% and 1.0% respectively, due to modest TLB miss reductions.

6.2 Soft SKU Performance

μSKU creates microservice-specific soft SKUs by independently analyzing each knob and then composing their best configurations. In Fig. 19, we show the final throughput gains achieved by *μSKU*'s soft SKUs as compared to (1) hand-tuned production configurations and (2) stock server configurations (i.e., after a fresh server re-install). The stock configuration comprises (1) 2.2 GHz and 2.0 GHz core frequency for Web and Ads1 respectively, (2) 1.8 GHz uncore frequency, (3) all cores active, (4) no CDP in LLC, (5) all prefetchers turned on, (6) *always ON* for THP, and (7) no SHPs. We listed the hand-tuned configurations in Sec. 6.1.

Since these services operate on hundreds of thousands of machines, achieving even single-digit percent speedups with *µSKU* can yield immense aggregate data center efficiency benefits by reducing a service's provisioning requirement. *µSKU*'s soft SKUs outperform stock configurations by 6.2% on `Web` (Skylake), 7.2% on `Web` (Broadwell), and 2.5% on `Ads1` due to benefits enabled by CDP, prefetchers, THP, and SHP. Interestingly, *µSKU* also outperforms the hand-tuned production configurations by 4.5% on `Web` (Skylake), 3.0% on `Web` (Broadwell), and 2.5% on `Ads1`. We confirmed that the MIPS improvement reported by *µSKU*'s soft SKUs yields a corresponding QPS improvement over a prolonged period (spanning several code pushes) by monitoring fleet-wide QPS via ODS. The statistically significant throughput gains are a substantial win in data centers' efficiency.

µSKU's prototype takes 5-10 hours to explore its knob design space and arrive at the final soft-SKU configurations. Even for knob settings where *µSKU* identifies the same result as manual tuning by experts, the savings in engineering effort by relying on an automated system is significant. A key advantage of *µSKU* is that it can be applied to microservices that do not have dedicated performance tuning engineers.

7 Discussion

We discuss open questions and *µSKU* prototype limitations.

Future hardware knobs. Our architectural characterization revealed significant diversity in architectural bottlenecks across microservices. We discussed opportunities for microservice-specific hardware modifications and motivated how soft SKUs can be designed using existing hardware- and OS-based configurable knobs. However, in light of a soft-SKU strategy, we anticipate that hardware vendors might introduce additional tunable knobs. *µSKU* does not currently adjust knobs to address microservice differences in instruction mix, branch prediction, context switch penalty, and other opportunities revealed in our characterization.

QoS and perf/watt constraints. Our microservices face stringent latency, throughput, and power constraints in the form of Service-Level Objectives (SLO). *µSKU*'s prototype performs A/B testing in a coarse-grained design space and tunes configurable hardware and OS knobs to improve throughput. However, *µSKU* does not consider energy or power constraints. QoS constraints are only addressed insofar as we discard parts of the *µSKU* tuning space that lead to violations.

µSKU can be extended to consider a cluster's SLOs' full range. For example, `Cache` executes exception handlers when latency targets are violated, which makes MIPS an inappropriate metric to quantify `Cache` performance. With support for other performance metrics, *µSKU* can perform A/B tests that discount exception-handling code when measuring throughput. With support to also measure system power/energy, *µSKU* can be extended to perform energy- or power-efficiency optimization rather than optimizing only for performance. We leave such support to future work.

Exhaustive design-space sweep. We notice that throughput improvements achieved by individual knobs are not always additive when *µSKU* composes them to generate a soft SKU. This observation implies that knob configurations may have subtle dependencies on which we might capitalize. An exhaustive characterization that determines a Pareto-optimal soft SKU might identify global performance maxima that are better than those found by our independent search. However, performing an exhaustive search is prohibitive; better search heuristics (e.g., hill climbing [86]) may be required.

µSKU and co-location. Our production microservices run on dedicated hardware without co-runners. Co-location can raise interesting challenges for future work—scheduler systems that map service affinities can be designed in a *µSKU*-aware manner.

8 Related Work

Architectural proposals for cloud services. Several works propose architectures suited to a particular, important cloud service. Ayers et al. [23] characterize Google web search's memory hierarchy and propose an L4 eDRAM cache to improve heap accesses. Earlier work [87] also discusses microarchitecture for Google search. Some works [88–90] characterize low-power cores for search engines like Nutch and Bing. Trancoso et al. [91] analyze the AltaVista search engine's memory behavior and find it similar to decision support workloads; Barroso et al. [92] show that L2 caches encompass such workloads' working set, leaving memory bandwidth under-utilized. Microsoft's Catapult accelerates search ranking via FPGAs [93]. DCBench studies latency-sensitive cloud data analytics [94]. Studying a single service class can restrict the generality of conclusions, as modern data centers typically execute diverse services with varied behaviors. In contrast, we characterize diverse production microservices running in the data centers of one of the largest social medial providers. We show that modern microservices exhibit substantial system-level and architectural differences, which calls for microservice-specific optimization.

Other works [1, 95] propose architectural optimizations for diverse applications. Kanev et al. [1] profile different Google services and propose architectural optimizations. Kozyrakis et al. [95] examine Microsoft's email, search, and analytics applications, focusing on balanced server design. However, these works do not customize SKUs for particular services.

Academic efforts develop and characterize benchmark suites for cloud services. Most notably, CloudSuite [21] comprises both latency-sensitive and throughput-oriented scale-out cloud workloads. Yasin et al. [63] perform a microarchitectural characterization of several CloudSuite workloads. However, our findings on production services differ from those of academic cloud benchmark suite studies [21, 22, 63, 96, 97]. For example, unlike these benchmark suites, our microservices have large L2 and LLC instruction working sets, high stall times, large front-end pipeline stalls, and lower IPC. While these suites are vital for experimentation, it is important to compare their characteristics against large-scale production microservices serving live user traffic.

Hardware tuning. Many works tune individual server knobs, such as selective voltage boosting [98–100], exploiting multicore heterogeneity [101–103], trading memory latency/bandwidth [104–107], or reducing front-end stalls [70, 96, 108]. In contrast, we propose (1) performance-efficient soft SKUs rather than hardware changes, (2) target diverse microservices, and (3) tune myriad knobs to create customized microservice-specific soft SKUs. Other works reduce co-scheduled job interference [109–114] or schedule them in a machine characteristics-aware manner [115–118]. Such studies can benefit from architectural insights provided here.

9 Conclusion

Modern data centers face server architecture design challenges as they must efficiently support diverse microservices. We presented a detailed system-level and architectural characterization of key microservices used by a leading social media provider. We highlighted surprising and diverse bottlenecks and proposed future server architecture optimization opportunities, since each microservice might benefit from a custom server SKU. However, to avoid per-service SKU deployment challenges, we instead proposed the "soft" SKU concept, wherein we tune coarse-grain configuration knobs on a few hardware SKUs. We developed *µSKU* to automatically tune server knobs to create microservice-specific soft SKUs that outperform stock servers by up to 7.2%.

10 Acknowledgement

We acknowledge Carlos Torres, Pallab Bhattacharya, Xiaodong Wang, Joy Chaoyue Xiong, Oded Horovitz, Denis Sheahan, Ning Sun, Mark Santaniello, Amlan Nayak, Chao Li, and Yudong Guang who provided valuable insights on Facebook workload characteristics and analysis. We acknowledge Murray Stokeley, Kim Hazelwood, Bharath Muthiah, Bill Jia, Christina Delimitrou, Carole-Jean Wu, Vaibhav Gogte, Amrit Gopal, PR Sriraman, Brendan West, Amirhossein Mirhosseini, and the anonymous reviewers for their insightful suggestions.

11 References

[1] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *International Symposium on Computer Architecture*, 2015.

[2] "The biggest thing amazon got right: The platform." `https://gigaom.com/2011/10/12/419-the-biggest-thing-amazon-got-right-the-platform/`.

[3] "Adopting microservices at netflix: Lessons for architectural design." `https://www.nginx.com/blog/microservices-at-netflix-architectural-best-practices/`.

[4] "Scaling Gilt: from Monolithic Ruby Application to Distributed Scala Micro-Services Architecture." `https://www.infoq.com/presentations/scale-gilt`.

[5] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Computing Colombian Conference*, 2015.

[6] "What is microservices architecture?." `https://smartbear.com/learn/api-design/what-are-microservices/`.

[7] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks, "Tradeoffs between power management and tail latency in warehouse-scale applications," in *IEEE International Symposium on Workload Characterization*, 2014.

[8] I. Nadareishvili, R. Mitra, M. McLarty, and M. Amundsen, *Microservice Architecture: Aligning Principles, Practices, and Culture*. 2016.

[9] A. Sriraman and T. F. Wenisch, "μSuite: A Benchmark Suite for Microservices," in *IEEE International Symposium on Workload Characterization*, 2018.

[10] A. Sriraman, "Unfair Data Centers for Fun and Profit," in *Wild and Crazy Ideas (ASPLOS)*, 2019.

[11] A. Sriraman and T. F. Wenisch, "μTune: Auto-Tuned Threading for OLDI Microservices," in *Proceedings of the 12th USENIX conference on Operating Systems Design and Implementation*, 2018.

[12] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux J.*, 2004.

[13] "Mcrouter." `https://github.com/facebook/mcrouter`.

[14] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the Source of Tail Latency Through Precise Load Testing and Statistical Inference," in *International Symposium on Computer Architecture*, 2016.

[15] X. He, J. Pan, O. Jin, T. Xu, B. Liu, T. Xu, Y. Shi, A. Atallah, R. Herbrich, S. Bowers, and J. Q. n. Candela, "Practical Lessons from Predicting Clicks on Ads at Facebook," in *International Workshop on Data Mining for Online Advertising*, 2014.

[16] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. C. Li, *et al.*, "TAO: Facebook's Distributed Data Store for the Social Graph," in *USENIX Annual Technical Conference*, 2013.

[17] M. Zuckerberg, R. Sanghvi, A. Bosworth, C. Cox, A. Sittig, C. Hughes, K. Geminder, and D. Corson, "Dynamically providing a news feed about a user of a social network," 2010.

[18] G. Ottoni, "HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack," in *Conference on Programming Language Design and Implementation*, 2018.

[19] J. L. Henning, "Spec cpu2006 benchmark descriptions," *SIGARCH Comp. Arch. News*, 2006.

[20] A. Limaye and T. Adegbija, "A Workload Characterization of the SPEC CPU2017 Benchmark Suite," in *International Symposium on Performance Analysis of Systems and Software*, 2018.

[21] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2012.

[22] Y. Gan and C. Delimitrou, "The Architectural Implications of Cloud Microservices," *IEEE Computer Architecture Letters*, 2018.

[23] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory Hierarchy for Web Search," in *International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[24] O. Yamauchi, *Hack and HHVM: programming productivity without breaking things*. 2015.

[25] K. Adams, J. Evans, B. Maher, G. Ottoni, A. Paroski, B. Simmers, E. Smith, and O. Yamauchi, "The hiphop virtual machine," in *Acm Sigplan Notices*, 2014.

[26] E. Rader and R. Gray, "Understanding user beliefs about algorithmic curation in the facebook news feed," in *ACM conference on human factors in computing systems*, 2015.

[27] E. Bakshy, S. Messing, and L. A. Adamic, "Exposure to ideologically diverse news and opinion on Facebook," *Science*, 2015.

[28] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, and A. Kalro, "Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective," in *International Symposium on High Performance Computer Architecture*, 2018.

[29] V. Venkataramani, Z. Amsden, N. Bronson, G. Cabrera III, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, and J. Hoon, "Tao: how facebook serves the social graph," in *International Conference on Management of Data*, 2012.

[30] J. L. Carlson, *Redis in Action*. 2013.

[31] J. Doweck, W.-F. Kao, A. K.-y. Lu, J. Mandelblat, A. Rahatekar, L. Rappoport, E. Rotem, A. Yasin, and A. Yoaz, "Inside 6th-generation intel core: new microarchitecture code-named skylake," *IEEE Micro*, 2017.

[32] "Unlock system performance in dynamic environments." https://www.intel.com/content/www/us/en/architecture-and-technology/resource-director-technology.html.

[33] C. Intel, "Improving Real-Time Performance by Utilizing Cache Allocation Technology," *Intel Corporation, April*, 2015.

[34] "Code and Data Prioritization - Introduction and Usage Models in the Intel Xeon Processor E5 v4 Family." https://software.intel.com/en-us/articles/introduction-to-code-and-data-prioritization-with-usage-models.

[35] D. Borthakur, J. Gray, J. S. Sarma, K. Muthukkaruppan, N. Spiegelberg, H. Kuang, K. Ranganathan, D. Molkov, A. Menon, and S. Rash, "Apache Hadoop goes realtime at Facebook," in *International Conference on Management of data*, 2011.

[36] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A fast, scalable, in-memory time series database," *Proceedings of the VLDB Endowment*, 2015.

[37] A. S. Aiyer, M. Bautin, G. J. Chen, P. Damania, P. Khemani, K. Muthukkaruppan, K. Ranganathan, N. Spiegelberg, L. Tang, and M. Vaidya, "Storage infrastructure behind Facebook messages: Using HBase at scale," *IEEE Data Eng. Bull.*, 2012.

[38] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-wide profiling: A continuous profiling infrastructure for data centers," *IEEE micro*, 2010.

[39] "Emon user's guide." https://software.intel.com/en-us/download/emon-user-guide.

[40] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, "Design tradeoffs for SSD performance," in *USENIX Annual Technical Conference*, 2008.

[41] "Intel and Micron Produce Breakthrough Memory Technology." https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.

[42] V. Gogte, S. Diestelhorst, W. Wang, S. Narayanasamy, P. M. Chen, and T. F. Wenisch, "Persistency for synchronization-free regions," in *Programming Language Design and Implementation*, 2018.

[43] A. Kolli, V. Gogte, A. Saidi, S. Diestelhorst, P. M. Chen, S. Narayanasamy, and T. F. Wenisch, "Language-level Persistency," in *International Symposium on Computer Architecture*, 2017.

[44] J. Vienne, J. Chen, M. Wasi-Ur-Rahman, N. S. Islam, H. Subramoni, and D. K. Panda, "Performance analysis and evaluation of infiniband fdr and 40gige roce on hpc and cloud computing systems," in *IEEE 20th Annual Symposium on High-Performance Interconnects*, 2012.

[45] S. Cho, A. Suresh, T. Palit, M. Ferdman, and N. Honarmand, "Taming the Killer Microsecond," in *International Symposium on Microarchitecture*, 2018.

[46] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the Killer Microseconds," *Communications of the ACM*, 2017.

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

[48] A. Mirhosseini, A. Sriraman, and T. F. Wenisch, "Hiding the Microsecond-Scale Latency of Storage-Class Memories with Duplexity," in *Annual Non-Volative Memories Workshop*, 2019.

[49] L. Luo, A. Sriraman, B. Fugate, S. Hu, G. Pokam, C. J. Newburn, and J. Devietti, "LASER: Light, Accurate Sharing dEtection and Repair," in *International Symposium on High Performance Computer Architecture*, 2016.

[50] A. Sriraman and T. F. Wenisch, "Performance-Efficient Notification Paradigms for Disaggregated OLDI Microservices," in *Workshop on Resource Disaggregation*, 2019.

[51] A. Sriraman, S. Liu, S. Gunbay, S. Su, and T. F. Wenisch, "Deconstructing the Tail at Scale Effect Across Network Protocols," *The Annual Workshop on Duplicating, Deconstructing, and Debunking*, 2016.

[52] D. Tsafrir, "The context-switch overhead inflicted by hardware interrupts (and the enigma of do-nothing loops)," in *Workshop on Experimental computer science*, 2007.

[53] C. Li, C. Ding, and K. Shen, "Quantifying the cost of context switch," in *Workshop on Experimental computer science*, 2007.

[54] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan, "High performance network virtualization with SR-IOV," *Journal of Parallel and Distributed Computing*, 2012.

[55] E. Y. Jeong, S. Woo, M. Jamshed, H. Jeong, S. Ihm, D. Han, and K. Park, "mTCP: A Highly Scalable User-level TCP Stack for Multicore Systems," in *USENIX Conference on Networked Systems Design and Implementation*, 2014.

[56] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A Protected Dataplane Operating System for High Throughput and Low Latency," in *USENIX Conference on Operating Systems Design and Implementation*, 2014.

[57] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe User-level Access to Privileged CPU Features," in *USENIX Symposium on Operating Systems Design and Implementation*, 2012.

[58] P. Emmerich, M. Pudelko, S. Bauer, and G. Carle, "User Space Network Drivers," in *Proceedings of the Applied Networking Research Workshop*, 2018.

[59] M. Lavasani, H. Angepat, and D. Chiou, "An FPGA-based in-line accelerator for memcached," *IEEE Computer Architecture Letters*, 2013.

[60] T. R. Learmont, "Fine-grained consistency mechanism for optimistic concurrency control using lock groups," 2001.

[61] C. J. Blythe, G. A. Cuomo, E. A. Daughtrey, and M. R. Hogstrom, "Dynamic thread pool tuning techniques," 2007.

[62] A. Starovoitov, "BPF in LLVM and kernel," in *Linux Plumbers Conference*, 2015.

[63] A. Yasin, Y. Ben-Asher, and A. Mendelson, "Deep-dive analysis of the data analytics workload in cloudsuite," in *International Symposium on Workload Characterization*, 2014.

[64] D. Chen, D. X. Li, and T. Moseley, "AutoFDO: Automatic feedback-directed optimization for warehouse-scale applications," in *International Symposium on Code Generation & Optimization*, 2016.

[65] T. Johnson, M. Amini, and X. D. Li, "ThinLTO: scalable and incremental LTO," in *IEEE/ACM International Symposium on Code Generation and Optimization*, 2017.

[66] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki, "Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches," in *International Symposium on Computer Architecture*, 2009.

[67] I. Papadakis, K. Nikas, V. Karakostas, G. Goumas, and N. Koziris, "Improving QoS and Utilisation in modern multi-core servers with Dynamic Cache Partitioning," in *Proceedings of the Joined Workshops COSH 2017 and VisorHPC 2017*, 2017.

[68] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer, and B. Falsafi, "Scale-out Processors," in *International Symposium on Computer Architecture*, 2012.

[69] S. Bharadwaj, G. Cox, T. Krishna, and A. Bhattacharjee, "Scalable Distributed Shared Last-Level TLBs Using Low-Latency Interconnects," in *International Symposium on Microarchitecture*, 2018.

[70] R. Kumar, B. Grot, and V. Nagarajan, "Blasting Through the Front-End Bottleneck with Shotgun," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2018.

[71] A. Bhattacharjee, "Translation-Triggered Prefetching," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[72] G. Cox and A. Bhattacharjee, "Efficient Address Translation for Architectures with Multiple Page Sizes," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[73] B. Pham, A. Bhattacharjee, Y. Eckert, and G. H. Loh, "Increasing TLB reach by exploiting clustering in page translations," in *International Symposium on High Performance Computer Architecture*, 2014.

[74] B. Pham, V. Vaidyanathan, A. Jaleel, and A. Bhattacharjee, "Colt: Coalesced large-reach TLBs," in *International Symposium on Microarchitecture*, 2012.

[75] V. Karakostas, J. Gandhi, F. Ayar, A. Cristal, M. D. Hill, K. S. McKinley, M. Nemirovsky, M. M. Swift, and O. Ünsal, "Redundant Memory Mappings for Fast Access to Large Memories," in *International Symposium on Computer Architecture*, 2015.

[76] "Intel Memory Latency Checker v3.6." https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[77] B. Falsafi and T. F. Wenisch, "A primer on hardware prefetching," *Synthesis Lectures on Computer Architecture*, 2014.

[78] D. Meisner, J. Wu, and T. F. Wenisch, "BigHouse: A Simulation Infrastructure for Data Center Systems," in *International Symposium on Performance Analysis of Systems & Software*, 2012.

[79] E. Rotem, "Intel architecture, code name Skylake deep dive: A new architecture to manage power performance and energy efficiency," in *Intel Developer Forum*, 2015.

[80] D. Hackenberg, R. Schöne, T. Ilsche, D. Molka, J. Schuchart, and R. Geyer, "An energy efficiency feature survey of the intel haswell processor," in *IEEE International Parallel and Distributed Processing Symposium Workshop*, 2015.

[81] H. Akkan, M. Lang, and L. M. Liebrock, "Stepping towards noiseless linux environment," in *International workshop on runtime and operating systems for supercomputers*, 2012.

[82] "Intel resource director technology (rdt) in linux." https://01.org/intel-rdt-linux.

[83] "Disclosure of H/W prefetcher control on some Intel processors." https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.

[84] A. Arcangeli, "Transparent hugepage support," in *KVM forum*, 2010.

[85] A. S. Gadre, K. Kabra, A. Vasani, and K. Darak, "X-xen: huge page support in xen," in *Linux Symposium*, 2011.

[86] B. Selman and C. P. Gomes, "Hill-climbing search," *Encyclopedia of Cognitive Science*, 2006.

[87] L. A. Barroso, J. Dean, and U. Holzle, "Web search for a planet: The google cluster architecture," in *IEEE Micro*, 2003.

[88] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and designing new server architectures for emerging warehouse-computing environments," in *ACM SIGARCH Computer Architecture News*, 2008.

[89] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid, "Web search using mobile cores: quantifying and mitigating the price of efficiency," in *ACM SIGARCH Computer Architecture News*, 2010.

[90] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "Fawn: A fast array of wimpy nodes," in *Symposium on Operating Systems Principles*, 2009.

[91] P. Trancoso, J.-L. Larriba-Pey, Z. Zhang, and J. Torrellas, "The memory performance of DSS commercial workloads in shared-memory multiprocessors," in *International Symposium High-Performance Computer Architecture*, 1997.

[92] L. A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory system characterization of commercial workloads," in *ACM SIGARCH Computer Architecture News*, 1998.

[93] P. Andrew, M. C. Adrian, S. C. Eric, D. Chiou, and K. Constantinides, "A reconfigurable fabric for accelerating large-scale datacenter services," in *International Symposium on Computer Architecuture*, 2014.

[94] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing data analysis workloads in data centers," in *International Symposium on Workload Characterization*, 2013.

[95] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, "Server engineering insights for large-scale online services," *IEEE micro*, 2010.

[96] Y. Zhu, D. Richins, M. Halpern, and V. J. Reddi, "Microarchitectural Implications of Event-driven Server-side Web Applications," in *International Symposium on Microarchitecture*, 2015.

[97] H. M. Makrani and H. Homayoun, "MeNa: A memory navigator for modern hardware in a scale-out environment," in *International Symposium on Workload Characterization*, 2017.

[98] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, L. Tang, J. Mars, and R. Dreslinski, "Adrenaline: Pinpointing and Reining in Tail Queries with Quick Voltage Boosting," in *International Symposium on High Performance Computer Architecture*, 2015.

[99] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in *International Symposium on Microarchitecture*, 2015.

[100] G. Prekas, M. Primorac, A. Belay, C. Kozyrakis, and E. Bugnion, "Energy Proportionality and Workload Consolidation for Latency-critical Applications," in *ACM Symposium on Cloud Computing*, 2015.

[101] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, "Exploiting Heterogeneity for Tail Latency and Energy Efficiency," in *International Symposium on Microarchitecture*, 2017.

[102] S. Panneerselvam and M. Swift, "Rinnegan: Efficient Resource Use in Heterogeneous Architectures," in *International Conference on Parallel Architectures and Compilation*, 2016.

[103] C. Delimitrou and C. Kozyrakis, "Amdahl's law for tail latency," *Communications of the ACM*, 2018.

[104] K. Chang, A. Kashyap, H. Hassan, S. Ghose, K. Hsieh, D. Lee, T. Li, G. Pekhimenko, S. Khan, and O. Mutlu, "Understanding Latency Variation in Modern DRAM Chips: Experimental Characterization, Analysis, and Optimization," in *International Conference on Measurement and Modeling of Computer Science*, 2016.

[105] M. Awasthi, "Rethinking Design Metrics for Datacenter DRAM," in *International Symposium on Memory Systems*, 2015.

[106] S. Volos, D. Jevdjic, B. Falsafi, and B. Grot, "An effective dram cache architecture for scale-out servers," tech. rep., 2016.

[107] Y. Wang, A. Tavakkol, L. Orosa, S. Ghose, N. Ghiasi, M. Patel, J. S. Kim, H. Hassan, M. Sadrosadati, and O. Mutlu, "Reducing DRAM Latency via Charge-Level-Aware Look-Ahead Partial Restoration," in *International Symposium on Microarchitecture*, 2018.

[108] C. Kaynak, B. Grot, and B. Falsafi, "Confluence: Unified Instruction Supply for Scale-out Servers," in *International Symposium on Microarchitecture*, 2015.

[109] J. Li, N. K. Sharma, D. R. K. Ports, and S. D. Gribble, "Tales of the Tail: Hardware, OS, and Application-level Sources of Tail Latency," in *ACM Symposium on Cloud Computing*, 2014.

[110] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," in *International Conference on High Performance Computing, Networking, Storage and Analysis*, 2012.

[111] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *International Symposium on Microarchitecture*, 2011.

[112] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi 2: CPU performance isolation for shared compute clusters," in *European Conference on Computer Systems*, 2013.

[113] Y. Xu, Z. Musgrave, B. Noble, and M. Bailey, "Bobtail: Avoiding Long Tails in the Cloud," in *NSDI*, 2013.

[114] L. Tang, J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa, "The Impact of Memory Subsystem Resource Sharing on Datacenter Applications," in *Int. Symposium on Computer Architecture*, 2011.

[115] J. Mars and L. Tang, "Whare-map: heterogeneity in homogeneous warehouse-scale computers," in *International Symposium on Computer Architecture*, 2013.

[116] C. Delimitrou and C. Kozyrakis, "Paragon: QoS-aware Scheduling for Heterogeneous Datacenters," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[117] X. Yang, S. M. Blackburn, and K. S. McKinley, "Elfen Scheduling: Fine-Grain Principled Borrowing from Latency-Critical Workloads Using Simultaneous Multithreading," in *USENIX Annual Technical Conference*, 2016.

[118] N. Mishra, J. D. Lafferty, and H. Hoffmann, "Esp: A machine learning approach to predicting application interference," in *International Conference on Autonomic Computing*, 2017.