Profiling a warehouse-scale computer

Svilen Kanev[†] Harvard University
Juan Pablo Darago[†] Universidad de Buenos Aires
Kim Hazelwood[†] Yahoo Labs
Parthasarathy Ranganathan Google
Tipp Moseley Google
Gu-Yeon Wei Harvard University
David Brooks Harvard University

Abstract

With the increasing prevalence of warehouse-scale (WSC) and cloud computing, understanding the interactions of server applications with the underlying microarchitecture becomes ever more important in order to extract maximum performance out of server hardware. To aid such understanding, this paper presents a detailed microarchitectural analysis of live datacenter jobs, measured on more than 20,000 Google machines over a three year period, and comprising thousands of different applications.

We first find that WSC workloads are extremely diverse, breeding the need for architectures that can tolerate application variability without performance loss. However, some patterns emerge, offering opportunities for co-optimization of hardware and software. For example, we identify common building blocks in the lower levels of the software stack. This "datacenter tax" can comprise nearly 30% of cycles across jobs running in the fleet, which makes its constituents prime candidates for hardware specialization in future server systems-on-chips. We also uncover opportunities for classic microarchitectural optimizations for server processors, especially in the cache hierarchy. Typical workloads place significant stress on instruction caches and prefer memory latency over bandwidth. They also stall cores often, but compute heavily in bursts. These observations motivate several interesting directions for future warehouse-scale computers.

1. Introduction

Recent trends show computing migrating to two extremes: software-as-a-service and cloud computing on one end, and more functional mobile devices and sensors ("the internet of things") on the other end. Given that the latter category is often supported by back-end computing in the cloud, designing next-generation cloud and datacenter platforms is among the most important challenges for future computer architectures.

Computing platforms for cloud computing and large internet services are often hosted in large data centers, referred to as warehouse-scale computers (WSCs) [4]. The design challenges for such warehouse-scale computers are quite different from those for traditional servers or hosting services, and emphasize system design for internet-scale services across thousands of computing nodes for performance and cost-efficiency at scale. Patterson and Hennessy, for example, posit that these warehouse-scale computers are a distinctly new class of computer systems that architects must design to [19]: "the datacenter is the computer"[41].

At such scale, understanding performance characteristics becomes critical – even small improvements in performance or utilization can translate into immense cost savings. Despite that, there has been a surprising lack of research on the interactions of live, warehouse-scale applications with the underlying microarchitecture. While studies on isolated datacenter benchmarks [14, 49], or system-level characterizations of WSCs [5, 27], do exist, little is known about detailed performance characteristics of at-scale deployments.

This paper presents the first (to the best of our knowledge) profiling study of a live production warehouse-scale computer. We present detailed quantitative analysis of microarchitecture events based on a longitudinal study across tens of thousands of server machines over three years running workloads and services used by billions of users. We highlight important patterns and insights for computer architects, some significantly different from common wisdom for optimizing SPEC-like or open-source scale-out workloads.

Our methodology addresses key challenges to profiling large-scale warehouse computers, including breakdown analysis of microarchitectural stall cycles and temporal analysis of workload footprints, optimized to address variation over the 36+ month period of our data (Section 2). Even though extracting maximal performance from a WSC requires a careful concert of many system components [4], we choose to focus on server processors (which are among the main determinants of both system power and performance [25]) as a necessary first step in understanding WSC performance.

From a software perspective, we show significant diversity in workload behavior with no single "silver-bullet" application to optimize for and with no major intra-application hotspots (Section 3). While we find little hotspot behavior within appli-

[†] The work was done when these authors were at Google.

ISCA'15, June 13-17, 2015, Portland, OR USA
ACM 978-1-4503-3402-0/15/06.
http://dx.doi.org/10.1145/2749469.2750392

cations, there are common procedures *across applications* that constitute a significant fraction of total datacenter cycles. Most of these hotspots are in functions unique to performing computation that transcends a single machine – components that we dub "datacenter tax", such as remote procedure calls, protocol buffer serialization and compression (Section 4). Such "tax" presents interesting opportunities for microarchitectural optimizations (e.g., in- and out-of-core accelerators) that can be applied to future datacenter-optimized server systems-on-chip (SoCs).

Optimizing tax alone is, however, not sufficient for radical performance gains. By drilling into the reasons for low core utilization (Section 5), we find that the cache and memory systems are notable opportunities for optimizing server processors. Our results demonstrate a significant and growing problem with instruction-cache bottlenecks. Front-end core stalls account for 15-30% of all pipeline slots, with many workloads showing 5-10% of cycles completely starved on instructions (Section 6). The instruction footprints for many key workloads show significant growth rates (≈30% per year), greatly exceeding the current growth of instruction caches, especially at the middle levels of the cache hierarchy.

Perhaps unsurprisingly, data cache misses are the largest fraction of stall cycles, at 50% to 60% (Section 7). Latency is a significantly bigger bottleneck than memory bandwidth, which we find to be heavily over provisioned for our workloads. A typical datacenter application mix involves access patterns that indicate bursts of computations mixed with bursts of stall times, presenting challenges for traditional designs. This suggests that while wide, out-of-order cores are necessary, they are often used inefficiently. While simultaneous multithreading (SMT) helps both with hiding latency and overlapping stall times (Section 8), relying on current-generation 2-wide SMT is not sufficient to eliminate the bulk of overheads we observed.

Overall, our study suggests several interesting directions for future microarchitectural exploration: design of more general-purpose cores with additional threads to address broad workload diversity, with specific accelerators for "datacenter tax" components, improved emphasis on the memory hierarchy, including optimizations to trade-off bandwidth for latency, as well as increased emphasis on instruction cache optimizations (partitioning i-cache/d-cache, etc). Each of these areas deserves further study in the quest of more performant warehouse-scale computers.

2. Background and methodology

This study profiles a production warehouse-scale computer at large, aggregating performance data across thousands of applications and identifying architectural bottlenecks at this scale. The rest of this section describes a typical WSC software environment and then details the methodology that enables such analysis.

Background: WSC software deployment We begin with a brief description of the software environment of a modern warehouse-scale computer as a prerequisite to understanding how processors perform under a datacenter software stack. While the idioms described below are based on our experience at Google, they are typical for large-scale distributed systems, and pervasive in other platform-as-a-service clouds.

Datacenters have bred a software architecture of distributed, multi-tiered services, where each individual service exposes a relatively narrow set of APIs.[1] Communication between services happens exclusively through remote procedure calls (RPCs) [17]. Requests and responses are serialized in a common format (at Google, protocol buffers [18]). Latency, especially at the tail end of distributions, is the defining performance metric, and a plethora of techniques aim to reduce it [11].

One of the main benefits of small services with narrow APIs is the relative ease of testability and deployment. This encourages fast release cycles – in fact, many teams inside Google release weekly or even daily. Nearly all of Google's datacenter software is stored in a single shared repository, and built by one single build system [16]. Consequently, code sharing is frequent, binaries are mostly statically linked to avoid dynamic dependency issues. Through these transitive dependences, binaries often reach 100s of MBs in size. Thus, datacenter CPUs are exposed to varied and diverse workloads, with large instruction footprints, and shared low-level routines.

Continuous profiling We collect performance-related data from the many live datacenter workloads using Google-Wide-Profiling (GWP) [44]. GWP is based on the premise of low-overhead random sampling, both of machines within the datacenter, and of execution time within a machine. It is inspired by systems like DCPI [2].

In short, GWP collectors: (i) randomly select a small fraction of Google's server fleet to profile each day, (ii) trigger profile collection remotely on each machine-under-test for a brief period of time (most often through `perf` [10]), (iii) symbolize the collected sample's callstacks (such that they are tagged with corresponding code locations) and (iv) aggregate a large number of such samples from many machines in a Dremel [37] database for easy analysis. The GWP collection pipeline has been described in detail by Ren et al. [44].

GWP has been unobtrusively sampling Google's fleet for several years, which makes it a perfect vehicle for longitudinal studies that answer where cycles have been spent over large periods of time. We perform several such studies with durations of 12-36 months in the following sections.

We focus these studies on code written in C++, because it is the dominant language that consumes CPU cycles. This is not necessarily the case in terms of popularity. A large amount of code (measured in lines-of-code) is written in other

[1] Recently the term "microservices" [38] has been coined to describe such a system architecture. The concept itself predates the term [40].

languages (mostly Java, Python and Go), however such code is responsible for a small fraction of cycles overall. Focusing on C++ also simplifies symbolizing callstacks with each collected sample. The aggregated set of these symbolized callstacks enables analyses that transcend application boundaries, and allows us to search for hotspots at true warehouse scale.

Architecture-specific collection To analyze more subtle interactions of warehouse-scale applications with the underlying hardware, we use processor performance counters that go beyond attributing cycles to code regions. We reuse the majority of GWP's infrastructure to collect performance counters and ask microarchitecture-specific questions. Since counters are intricately tied to a specific microarchitecture, we limit such studies to machines with Intel Ivy Bridge processors.

In more detail, for each such dedicated collection, we randomly select $\approx 20,000$ Ivy Bridge machines, and profile all jobs running on them to gather 1-second samples of the respective performance counters. For per-thread measurements, we also collect the appropriate metadata to attribute the samples to the particular job executing the thread, and its respective binary (through `perf`'s container group support). We also take special care to validate the performance counters that we use with microbenchmarks (errata in more exotic performance counters can be common), and to only use counter expressions that can fit constraints of a core's performance monitoring unit (PMU) in a single measurement (time-multiplexing the PMU often results in erroneous counter expressions). The last requirement limits the analyses that we perform. A common practice for evaluating complex counter expressions that do not fit a single PMU is to simply collect the necessary counters during multiple runs of the same application. In a sampling scenario, this is not trivially applicable because different parts of the counter expression can come from different samples, and would require special normalization to be comparable to one another.

All expressions that we do collect in single-PMU chunks are ratios (normalized by cycles or instructions) and do not require such special treatment. Their individual samples can be compared against each other and aggregated without any additional normalization. We typically show the distributions of such samples, compressed in box plots. Boxes, drawn around the median value, represent the 25-th and 75-th percentiles of such distributions, while whiskers (in the plots where shown) – the 10-th and 90-th.

Binary	Description
`ads`	Content ad targeting – matches ads with web pages based on page contents.
`bigtable`	Scalable, distributed, storage [7].
`disk`	Low-level distributed storage driver.
`flight-search`	Flight search and pricing engine.
`gmail`	Gmail back-end server.
`gmail-fe`	Gmail front-end server.
`indexing1`, `indexing2`	Components of search indexing pipelines [5].
`search1`, `search2`, `search3`	Search leaf nodes [36].
`video`	Video processing tasks: transcoding, feature extraction.

Table 1: Workload descriptions

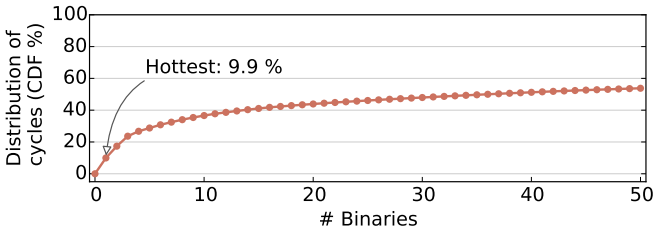

Figure 1: There is no "killer application" to optimize for. The top 50 hottest binaries only cover ≈60% of WSC cycles.

Performance counter analysis We use a performance analysis methodology, called Top-Down, recently proposed by Yasin [48]. Top-Down allows for reconstructing approximate CPI stacks in modern out-of-order processors, a task considered difficult without specialized hardware support [13]. The exact performance counter expressions that we use are identical with the ones listed in the Top-Down work [48].

Similar to other cycle counting methodologies [6, 13], Top-Down calculates the cost of microarchitectural stall events in cycles, as opposed to in more conventional metrics (e.g. miss rates, misses per kilo-instruction – MPKI), quantifying the end cost in performance for such events. This is especially important for modern complex out-of-order processors which have a wide range of latency hiding mechanisms. For example, a high value for MPKI in the L1 instruction cache can raise a false alarm for optimizing instruction footprint. In fact, a modern core's front end has sufficient buffering, and misses in the L1 alone cause very little end-performance impact.

Workloads While we do make the observation that workloads are getting increasingly diverse, we focused on 12 binaries (Table 1) for in-depth microarchitectural analysis. The main selection criterion was diversity. Thus, we ended up with jobs from several broad application classes – batch (`video`, `indexing`) vs. latency-conscious (the rest); low-level services (`disk`, `bigtable`) through back-ends (`gmail`, `search`) to front-end servers (`gmail-fe`). We strived to include varied microarchitectural behaviors – different degrees of data cache pressure, front-end bottlenecks, extracted IPC, etc. We also report averages aggregated over a significantly larger number of binaries than the hand-picked 12.

Finally, we make the simplifying assumption that one application equals one binary and use the two terms interchangeably (Kambadur et al. [24] describe application delineation tradeoffs in a datacenter setting). This has no impact on any results for the 12 workloads described above, because they are composed of single binaries.

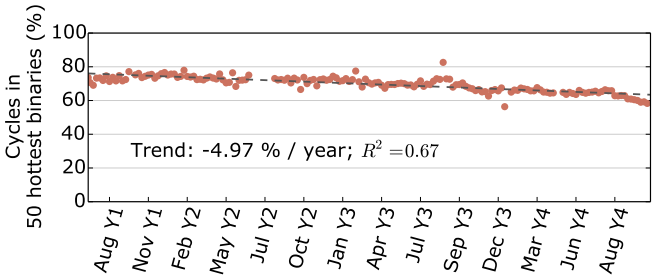

Figure 2: Workloads are getting more diverse. Fraction of cycles spent in top 50 hottest binaries is decreasing.

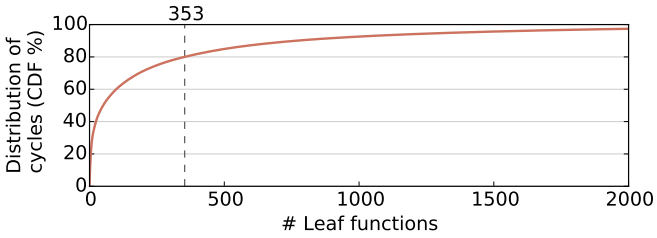

Figure 3: Individual binaries are already optimized. Example binary without hotspots, and with a very flat execution profile.

3. Workload diversity

The most apparent outcome of this study is the diversity of workloads in a modern warehouse-scale computer. While WSCs were initially created with a "killer application" in mind [5], the model of "the datacenter is the computer" has since grown and current datacenters handle a rapidly increasing pool of applications.

To confirm this point, we performed a longitudinal study of applications running in Google's warehouse-scale computers over more than 3 years. Figure 1 shows the cumulative distribution of CPU cycles among applications for the last week of the study. It is clear that no single application dominates the distribution – the hottest one accounts for ≈10% of cycles. Furthermore, it takes a tail of 50 different applications to build up to 60% of cycles.

Figure 1 is just a single slice in time of an ongoing diversification trend. We demonstrate that in Figure 2, which plots the fraction of CPU cycles spent in the 50 hottest binaries for each week of the study. While at the earliest periods we examined, 50 applications were enough to account for 80% of execution time, three years later, the same number (not necessarily the same binaries) cover less than 60% of cycles. On average, the coverage of the top 50 binaries has been decreasing by 5 percentage points per year over a period of more than 3 years. Note that this data set does not include data related to public clouds, which give orders of magnitude more programmers access to warehouse-scale resources, further increasing application diversity.

Applications exhibit diversity as well, having very flat execution profiles themselves. We illustrate this point with a CPU profile from `search3`, aggregated over a week of execution on a typically-sized cluster for that particular application. Figure 3 shows the distribution of CPU cycles over leaf functions – the hottest single function is responsible for only 6.3% of cycles, and it takes 353 functions to account for 80% of cycles. This tail-heavy behavior is in contrast with previous observations. For example, another scale-out workload, `Data analytics` from CloudSuite has been shown to contain significant hotspots – with 3 functions responsible for 65% of execution time [49].

From a software engineer's perspective, the absence of immediately apparent hotspots, both on the application and function levels, implies there is no substitute for datacenter-wide profiling. While there is value in optimizing hotspots on a per-application basis, the engineering costs associated with optimizing flat profiles are not always justified. This has driven Google to increasingly invest effort in automated, compiler-driven feedback-directed optimization [8]. Nevertheless, targeting the right common building blocks across applications can have significantly larger impact across the datacenter.

From an architect's point of view, it is similarly unlikely to find a single bottleneck for such a large amount of codes. Instead, in the rest of the paper, after aggregating over many thousands of machines running these workloads, we point out several smaller-scale bottlenecks. We then tie them back to suggestions for designing future WSC server systems.

4. Datacenter tax

Despite the significant workload diversity shown in Section 3, we see common building blocks once we aggregate sampled profile data across many applications running in a datacenter. In this section, we quantify the performance impact of the **datacenter tax**, and argue that its components are prime candidates for hardware acceleration in future datacenter SoCs.

We identify six components of this tax, detailed below, and estimate their contributions to all cycles in our WSCs. Figure 4 shows the results of this characterization over 11 months – "tax cycles" consistently comprise 22-27% of all execution. In a world of a growing number of applications (Figure 2), optimizing such inter-application common building blocks can lead to significant performance gains, more so than hunting for hotspots in individual binaries. We have observed services that spend virtually all their time paying tax, and would benefit disproportionately from reducing it.

The components that we included in the tax classification are: protocol buffer management, remote procedure calls (RPCs), hashing, compression, memory allocation and data movement. In order to cleanly attribute samples between them we only use leaf execution profiles (binning based on program counters, and not full call stacks). With leaf profiles, if the sample occurs in `malloc()` on behalf of RPC calls, that sample will be attributed to memory allocation, and not to RPC. This also guarantees that we always under-estimate the fraction of cycles spent in tax code.

While some portions of the tax are more specific to WSCs (protobufs and RPCs), the rest are general enough to be used

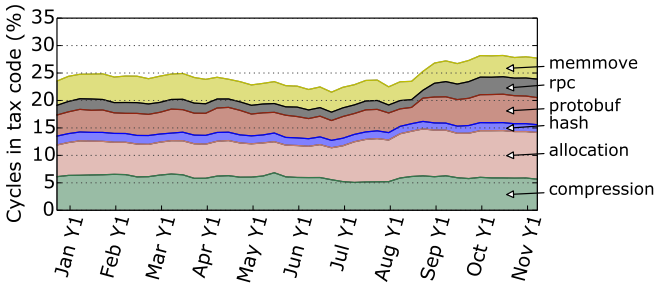

Figure 4: 22-27% of WSC cycles are spent in different components of "datacenter tax".

in various kinds of computation. When selecting which inter-application building blocks to classify as tax, we opted for generally mature low-level routines, that are also relatively small and self-contained. Such small, slowly-changing, widely-used routines are a great match for hardware specialization. In the following paragraphs, we sketch out possible directions for accelerating each tax component.

Protobuf management Protocol buffers [18] are the lingua franca for data storage and transport inside Google. One of the the most common idioms in code that targets WSCs is serializing data to a protocol buffer, executing a remote procedure call while passing the serialized protocol buffer to the remote callee, and getting a similarly serialized response back that needs deserialization. The serialization/deserialization code in such a flow is generated automatically by the protobuf compiler, so that the programmer can interact with native classes in their language of choice. Generated code is the majority of the `protobuf` portion in Figure 4.

The widespread use of protocol buffers is in part due to the encoding format's stability over time. Its maturity also implies that building dedicated hardware for protobuf (de)serialization in a server SoC can be successful, similarly to XML parsing accelerators [9, 46]. Like other data-intensive accelerators [28], such dedicated protobuf hardware should probably reside closer to memory and last-level caches, and get its benefits from doing computation close to memory.

Remote procedure calls (RPCs) RPCs are ubiquitous in WSCs. RPC libraries perform a variety of functions, such as load balancing, encryption, and failure detection. In our tax breakdown, these are collectively responsible for approximately a third of RPC tax cycles. The rest are taken up by simple data movement of the payloads. Generic data movement accelerators have been proposed [12] and would be beneficial for the latter portion.

Data movement In fact, RPCs are by far not the only code portions that do data movement. We also tracked all calls to the `memcpy()` and `memmove()` library functions to estimate the amount of time spent on explicit data movement (i.e., exposed through a simple API). This is a conservative estimate because it does not track inlined or explicit copies. Just the variants of these two library functions represent 4-5% of datacenter cycles. Recent work in performing data movement in DRAM [45] could optimize away this piece of tax.

Compression Approximately one quarter of all tax cycles are spent compressing and decompressing data.[2] Compression is spread across several different algorithms, each of which stresses a different point in the compression ratio/speed spectrum. This need not be the case for potential hardware-accelerated compression. For example, the `snappy` algorithm was designed specifically to achieve higher (de)compression speeds than `gzip`, sacrificing compression ratios in the process. Its usage might decrease in the presence of sufficiently fast hardware for better-compressing algorithms [30, 39].

Memory allocation Memory allocation and deallocation make up a substantial component of WSC computation (as seen by `allocation` in Figure 4), despite significant efforts in optimizing them in software [15, 29]. Current software implementations are mostly based on recursive data structures, and interact with the operating system, which makes them non-trivial to implement in hardware. However, the potential benefits suggest that an investigation in this direction is worthwhile.

Hashing We also included several hashing algorithms in our definition of tax cycles to estimate the potential for cryptography accelerators. Hashing represents a small, but consistent fraction of server cycles. Due to the large variety of hashes in use, this is a conservative estimate.

Kernel The kernel as a shared workload component deserves additional mention. It is obviously ubiquitous, and it is not surprising that WSC applications spend almost a fifth of their CPU cycles in the kernel (Figure 5). However, we do not consider it acceleratable tax – it is neither small, nor self-contained, and certainly not easy to replace with hardware. This is not to say it would not be beneficial to further optimize it in software. As an example, consider the scheduler in Figure 5, which has to deal with many diverse applications, each with even more concurrent threads (a 90-th percentile machine is running about 4500 threads concurrently [50]). Even after

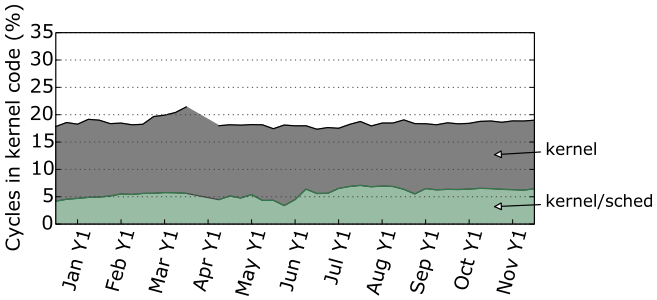

Figure 5: Kernel time, especially time spent in the scheduler, is a significant fraction of WSC cycles.

[2]We only include general-purpose lossless compression in this category, not audio/video coding.

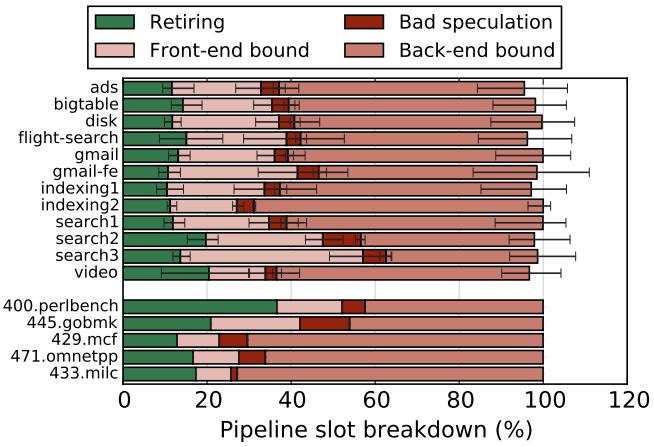

Figure 6: Top-level bottleneck breakdown. SPEC CPU2006 benchmarks do not exhibit the combination of low retirement rates and high front-end boundedness of WSC ones.

extensive tuning [47], the scheduler alone accounts for more than 5% of all datacenter cycles.

5. Microarchitecture analysis

Similar to the smaller components of the "datacenter tax" that together form a large fraction of all cycles, we expect multiple performance bottlenecks on the microarchitectural level. In order to easily identify them, we use the Top-Down [48] performance analysis methodology, which we incorporated in our fleet measurement infrastructure.

Top-Down chooses the micro-op (μop) queue of a modern out-of-order processor as a dividing point between a core's front-end and back-end, and uses it to classify μop pipeline slots (i.e. potentially committed μops) in four broad categories: `Retiring`, `Front-end bound`, `Bad speculation`, `Back-end bound`. Out of these, only `Retiring` classifies as "useful work" – the rest are sources of overhead that prevent the workload from utilizing the full core width.

Because of this single point of division the different components of this overhead are additive, very much like the components of a traditional CPI stack. The detailed methodology recursively breaks each overhead category into more concrete subcategories (e.g., `Back-end bound` into `Core-bound`, `L1-bound`, etc.), driving profiling in the direction of increasingly specific microarchitectural bottlenecks. We mostly focus on the top-level breakdown and several of its direct descendants – deeper subcategories require more complex counter expressions that are harder to collect accurately in sampled execution, as described in Section 2.

The breakdown in the four top categories can be summarized in a simple decision tree. If a μop leaves the μop queue, its slot can be classified as either `Retiring` or `Bad speculation`, depending on whether it makes it through to the commit stage. Similarly, if a μop-queue slot does not become empty in a particular cycle, there can be two reasons: it was either empty to begin with (`Front-end bound`), or the back-end was not ready for another μop (`Back-end bound`). These can be distinguished simply by a back-end stall signal. Intuitively, `Front-end bound` captures all overheads associated with fetching, instruction caches, decoding and some shorter-penalty front-end resteers, while `Back-end bound` is composed of overheads due to the data cache hierarchy and the lack of instruction-level parallelism.

We apply this approach to the overheads of datacenter workloads in Figure 6. It includes several SPEC CPU2006 benchmarks with well-known behaviors as reference points: `400.perlbench` – high IPC, largest i-cache working set; `445.gobmk` – hard-to-predict branches, highest IL1 MPKI; `429.mcf`, `471.omnetpp` – memory-bound, stressing memory latency; `433.milc` – memory-bound, stressing memory bandwidth.

The first immediate observation from Figure 6 is the small fraction of `Retiring` μops– similar, or often lower, than the lowest seen in SPEC (`429.mcf`). This implies that most datacenter workloads spend cores' time stalled on various bottlenecks. The majority of these stall slots are clearly due to back-end pressures – except for `search2` and `search3`, more than 60% of μop slots are held up due to the back-end. We will examine these more closely in Section 7. `Bad speculation` slots are within the range defined by the SPEC suite. Examining more closely, the 12 WSC applications show branch misprediction rates in a wide range from $0.5\times$ to $2\times$ those of `445.gobmk` and `473.astar`, with the rest of SPEC below the lower bound of that interval.

Finally, one type of behavior that clearly stands out in comparison with SPEC benchmarks is the large fraction of stalls due to front-end pressure. We investigate them in the next section.

6. Instruction cache bottlenecks

The Top-Down cycle breakdown shown in Figure 6 suggests that WSC applications spend a large portion of time stalled in the front-end. Indeed, `Front-end waste` execution slots are in the 15-30% range across the board (most often $2-3\times$ higher than those in typical SPEC benchmarks) Note that these indicate instructions *under-supplied* by the front-end – after the back-end has indicated it is able to accept more. We trace these to predominantly instruction cache problems, due to lots of lukewarm code. Finally, extrapolating i-cache working set trends from historical data, we see alarming growth rates for some applications that need to be addressed.

For a more detailed understanding of the reasons for front-end stalls, we first measure front-end starvation cycles – those when the μop queue is delivering 0 μops to the back-end. Figure 7 shows them to typically exceed 5% of all cycles. This is especially significant in the presence of deep (40+ instructions) front-end buffers, which absorb minor fetch bubbles. The most probable cause is a non-negligible fraction of long-latency instruction miss events – most likely instruction misses in the L2 cache.

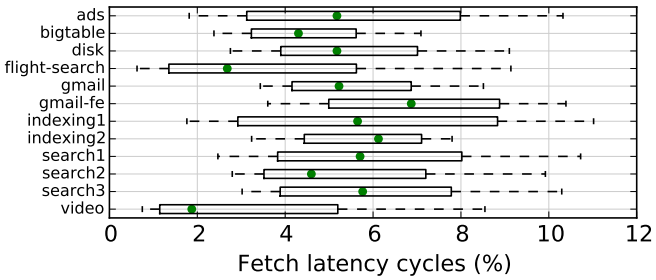

Figure 7: Cycles completely starved on front-end bottlenecks account for more than 5% of execution.

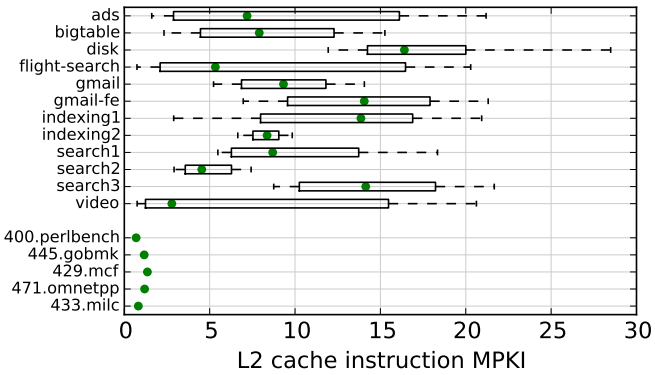

Figure 8: Instruction misses in the L2 cache are usually high.

Such a hypothesis is confirmed by the high exhibited L2 instruction miss rates from Figure 8. WSC applications typically miss in the range of 5-20 MPKI, an order of magnitude more frequently than the worst cases in SPEC CPU2006, and, at the high end of that interval, 50% higher than the rates measured for the scale-out workloads of CloudSuite [14].

The main reason for such high miss rates is simply the large code footprint of WSC applications. Binaries of 100s of MB are common and, as seen in Section 3, without significant hotspots. Thus, instruction caches have to deal with large code working sets – lots of "lukewarm instructions". This is made worse in the L2 cache, where instructions have to compete for cache capacity with the data stream, which typically also has a large working set.

There are several possible directions for architects to address instruction cache bottlenecks. Larger instruction caches are an obvious one, although higher capacity has to be balanced with increased latency and die constraints. More complex instruction prefetchers are another, which have been successful for private i-caches under non-trivial instruction miss rates [3, 26]. Finally, cache partitioning is another alternative, especially in light of high miss rates in the L2 and lukewarm code. While partitioning has been extensively studied for multiple applications' access streams in shared last-level caches (Qureshi and Patt [43], among many others), relatively little attention has been paid to treating the instruction and data streams differently, especially in private, mid-level caches. Recently, Jaleel et al. proposed modifying replacement policies to prioritize code over data [21], and the SPARC M7 design team opted for an architecture with completely separate L2 instruction and data caches [30].

A problem in the making Large instruction working sets are a widespread and growing issue. To demonstrate that, we use profiling data to estimate i-cache footprints of datacenter binaries over a period of 30 months. For some applications, such estimates grow by more than 25% year-over-year, significantly out-pacing i-cache size growth.

The canonical method to estimate a workload's working set size is simulation-based. It involves simply sweeping the cache size in a simulator, and looking for the "knee of the curve" – the size at which the miss rate drops to near zero. This is cumbersome, especially if performed over a large number of versions of the same binary to capture a growth trend. Instead, we developed a different, non-invasive approach to estimate it.

With only profiling data available, one can use unordered instruction pointer samples, and measure how many unique cache lines cover a large fraction (e.g. 99%) of all samples, when ranked by hotness. The rationale behind such a metric is that an infinite-size cache would obviously contain all unique lines. In a limited-size one, over a large enough time window, the LRU replacement policy eventually kicks out less frequently-used lines, until the hottest lines are left.

However, such a strategy is contingent on consistent sampling over time. In a long-term historical study, both the fraction of machines that get profiled and the amount of machines serving the same application can vary significantly, often in ways that are hard to control for. Under such variance, it is unfair to compare the absolute number of cache lines that cover a fixed fraction of samples for two time periods – 99% of 10× more samples are more likely to capture larger portions of the tail of instruction cache lines.

We compensate with yet another layer of sampling. For a particular binary version, we select a fixed number of random samples in post-processing a week's worth of data (in results shown below, this number is 1 million), and count the absolute number of unique cache lines that cover that new sample set. This is the equivalent of constructing a hotness ranking with a stable number of samples across measurement periods.

Figure 9 shows the results of applying this approach to 30 months of instruction pointer samples. It plots our estimate of the instruction cache working set size – the number of unique cache lines in 1M randomly-selected weekly samples for a specific binary. For calibration, we include `400.perlbench`, which has the largest measured i-cache working set in SPEC CPU2006 (≈172 KB) [20].

First, compared to SPEC, all workloads demonstrated several fold larger i-cache working sets. Figure 9 illustrates that for `search2` and `bigtable` – their i-cache footprints are 4× those of `400.perlbench`, which amounts to 688 KB or more. Note that such a size is significantly larger than the L2 cache in current architectures (Intel: 256 KB, AMD: 512 KB, IBM: 512 KB), which also has to be shared with the data stream.

More importantly, this estimate is growing over time, at

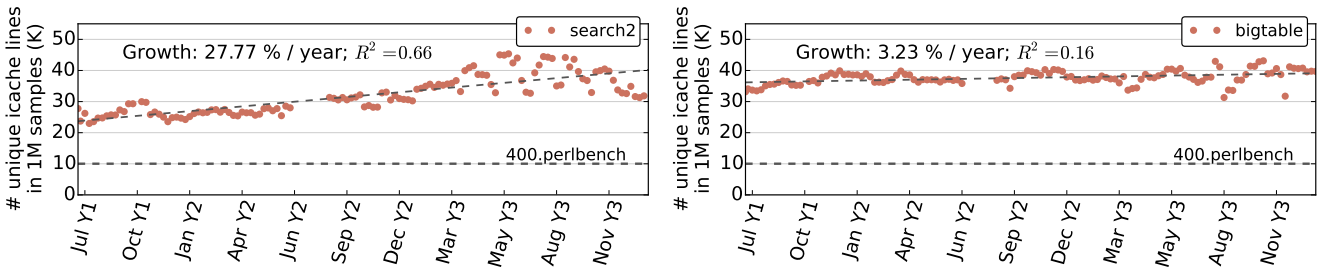

Figure 9: Large instruction cache footprints. Getting progressively larger for some applications.

alarming rates for some applications. Consider `search2` in Figure 9, whose footprint has almost doubled during the duration of the study, at 27% per year. Other workloads are more stable – for example, `bigtable` only sees a 3% year-to-year growth.

While the exact reasons for this disparity are unclear, we hypothesize it is related to development velocity. Products like `search` are under constant development, and often see a variety of new features added, which leads to simply more code. `bigtable`, on the other hand, is a relatively mature code base with a well-defined feature set that sees less development. A more quantitative study, correlating development speed with instruction footprint would make for interesting future work.

7. Core back-end behavior: dependent accesses

While the negative impact of large instruction working sets is likely to continue growing, the current dominant source of overhead identified by the Top-Down decomposition (Figure 6) is clearly in the core's back-end.

Overall, the combined influence of a large amount of front-end and back-end stalls results in very few instructions per cycle (IPC) on average (Figure 10) – almost 2x lower than the SPECint geomean and close to that of the most memory-bound benchmarks in SPEC (`429.mcf`, `471.omnetpp`, `433.milc`). This result is in line with published data on classical datacenter workloads [22], and has led researchers to investigate the potential of small cores for warehouse-scale applications [1, 22, 31]. We show a more nuanced picture, with bimodal *extracted* ILP, frequently low, but also with periods of more intense computation.

As a reminder, there are two very broad reasons for `Back-end bound` μop slots: time spent serving data cache requests, and lack of instruction-level parallelism (ILP). Of the two, data cache behavior is the dominant factor in our measurements. This is somewhat unsurprising, given the data-intensive nature of WSC workloads. Figure 11 serves as confirmation, showing the amount of back-end cycles, stalled due to pending loads in the cache hierarchy, or due to insufficient store buffer capacity. At 50-60% of all cycles, they account for more than 80% of `Back-end bound` pipeline slots shown earlier (Figure 6).

However, not all cycles are spent waiting on data caches. We demonstrate this in Figure 12, which measures the distribution of extracted ILP. By *extracted* ILP, we refer to the number

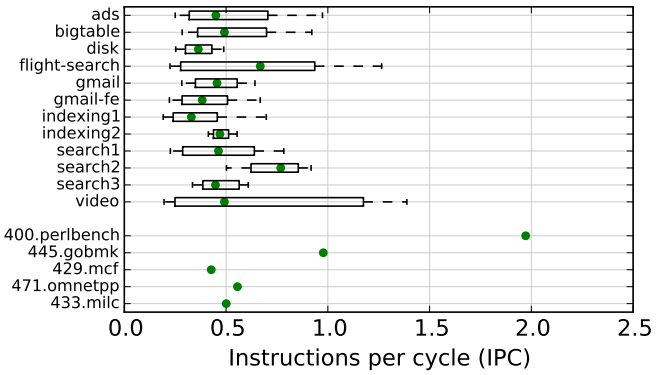

Figure 10: IPC is universally low.

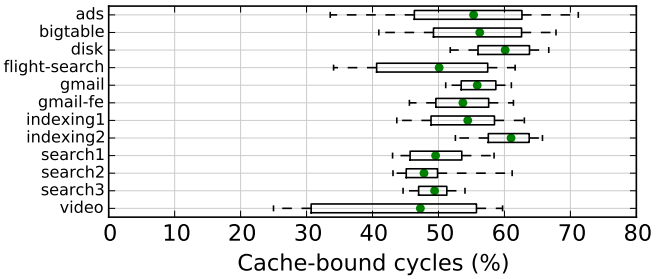

Figure 11: Half of cycles are spent stalled on caches.

of simultaneously executing μops at each cycle when some μops are issued from the out-of-order scheduler to execution units. We see that 72% of execution cycles exhibit low ILP (1 or 2 on a 6-wide Ivy Bridge core), consistent with the fact that the majority of cycles are spent waiting on caches. However, for the other 28% of cycles, 3 or more functional units are kept busy each cycle.

One explanation consistent with such behavior is that WSC applications exhibit a fine-grained mix of dependent cache accesses and bursty computation. The bursts of computation can either be dependent on the cache references, or independent and extractable as ILP. The difference between these two variants – whether intense compute phases are on the critical path of execution – could be detrimental for the amount of end performance degradation of "wimpier" cores, and requires a dedicated simulation study.

Memory bandwidth utilization Notice that in the previous paragraph, we immediately diagnose dependent cache accesses. We hypothesize this because of the very low memory bandwidth utilization that we observed, shown in Figure 13.

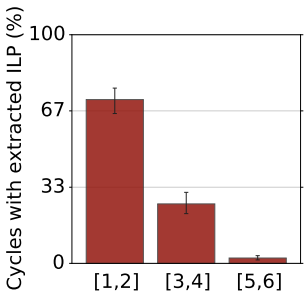

Figure 12: Extracted ILP. 28% of cycles utilize 3 or more execution ports on a 6-wide machine.

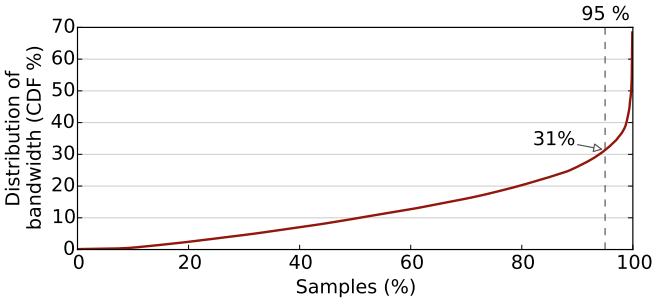

Figure 13: Memory bandwidth utilization is universally low.

The plot is a cumulative histogram of measured DRAM bandwidth across a sufficiently large number of machines.[3] The 95-th percentile of utilization is at 31%, and the maximum measured – 68%, with a heavy tail at the last percentile. Some portion of the low bandwidth usage is certainly due to low CPU utilization. However this is not a sufficient explanation – Barroso et al. show median CPU utilization in the 40%–70% range (depending on the type of cluster) [4], while we measure a significantly lower median bandwidth utilization at 10%. Note that the low bandwidth requirement is not very different from measurements on CloudSuite [14] and other emerging datacenter workloads [33].

One consequence of the low bandwidth utilization is that memory latency is more important than bandwidth for the set of the applications running in today's datacenters. In light of WSC server design, this might pose tradeoffs between memory bandwidth (or then number of memory controllers) and other uses of freed up silicon area (for example, more cores or accelerators).

Note that the large amount of unused bandwidth is also contrary to some typical benchmarking practices that focus on capacity. For example, SPECrate as commonly run (N copies on N cores) can shift several benchmarks' memory bottlenecks from latency to bandwidth [48], causing architects to optimize for a less relevant target.

8. Simultaneous multi-threading

The microarchitectural results shown so far did not account for simultaneous multi-threading (SMT), even though it is

[3]Measured through the sum of the `UNC_M_CAS_COUNT:RD` and `UNC_M_CAS_COUNT:WR` IvyTown uncore performance counters.

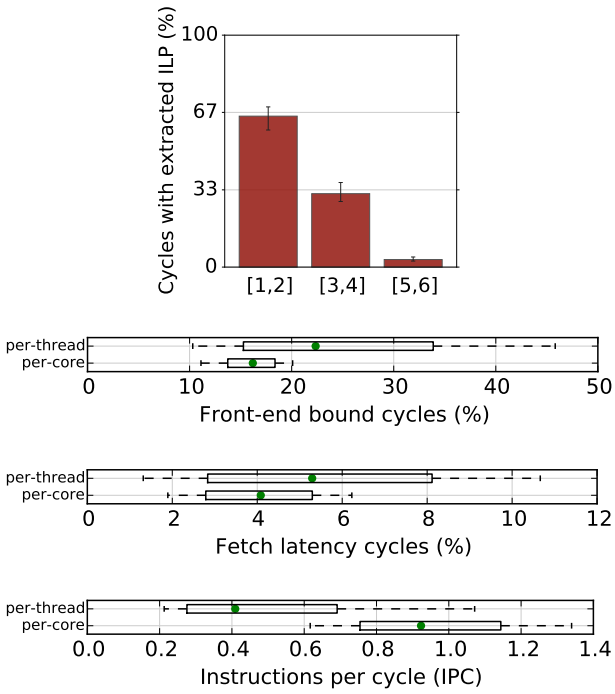

Figure 14: SMT effects on architectural behavior. From top to bottom: (i) more ILP extracted compared to Figure 12; (ii) front-end bound cycles decrease, but (iii) instruction starvation still exists; (iv) core throughput doubles with two hyperthreads.

enabled on the Ivy Bridge machines profiled. For example, the top-level cycle breakdown in Figure 6 was done on a per-hyperthread basis, assuming each hyperthread has the full machine width to issue μops.

Broadly speaking, SMT is most efficient when workloads have different performance bottlenecks, and multiple threads can complement each other's deficiencies. WSC applications, with inefficiencies in both the front-end and the back-end, as well as suspected fine-grained phase behavior, fit such a description well, and we expect them to benefit from SMT.

While we cannot perform at-scale measurements of counterfactuals without disturbing a large number of user-facing services (i.e., disabling SMT and looking at workload performance), we can at least estimate the efficacy of SMT by comparing specific per-hyperthread performance counters with ones aggregated on a per-core basis. Note that this is very different from measuring the speedup that a single application experiences from SMT. When a thread is co-run on a core, its performance naturally drops compared to when it has the full core available – mostly due to capacity effects, i.e. having to share microarchitectural units and caches. On the other hand, core utilization increases simply because multiple threads share it. While we cannot measure the first effect at-scale without turning SMT off, we can and do measure the latter.

As expected, functional unit utilization in the back-end increases when accounting for SMT. The first plot in Figure 14 shows that 3 or more of the 6 execution ports are used during

34% of cycles when counting both hyperthreads, as opposed to 28% in Figure 12, when counting each hyperthread separately.

While such improvements from SMT are expected and well-understood, the effects on front-end performance are less clear. On the one hand, SMT can increase instruction cache pressure – more instructions need to be fetched, even if hyperthreads share the same code, exacerbating an already severe instruction cache capacity bottleneck (Section 6). On the other, long-latency fetch bubbles on one hyperthread can be absorbed by fetching from another.

Our profiling data suggests that the latter effect dominates in WSC applications and SMT ends up improving front-end utilization. This is evident from the second and third plots of Figure 14. Per-core `Front-end bound` cycles are significantly lower than when measured per-hyperthread – 16% versus 22% on the medians, with drastically tighter distributions around them. Front-end starvation cycles (with no μops dispatched) also decrease from 5% to 4%, indicating that long-latency instruction cache misses are better absorbed, and SMT succeeds in alleviating some front-end inefficiencies.

Note however, that, even after we account for 2-wide SMT, 75% of collected fleet samples show an IPC value of 1.2 or less (last plot of Figure 14), compared to a theoretical machine width of 4.0. Adding this to the fact that latency bottlenecks (both due to fetching instructions from the L3 cache, and fetching data from main memory) are still far from eliminated suggests potential for wider SMT: with more threads per core, as seen in some server chips [30]. This case is strengthened by the low memory bandwidth utilization shown earlier – even with more threads per core bandwidth is unlikely to become a bottleneck. These results warrant further study of balancing the benefits of wider SMT with the potential costs, both in performance from potentially hitting capacity bottlenecks, and in power from the duplication or partitioning of core resources.

9. Related work

In recent years, research interest in developing new architectural support for datacenters has increased significantly. The concept of deploying "wimpy cores" or microservers to optimize datacenters has been well-explored [1, 22, 31], and recent efforts have investigated specialized interconnects [32] and customized hardware accelerators [42]. While our cycle breakdown finds opportunities for specialization, microarchitectural analysis suggests that "brawny" out-of-order superscalar cores provide sufficient performance to be justified, especially when coupled with wide SMT. As prior research has observed, "wimpy" cores and some forms of specialization excel in cost- and power-efficiency, often at the cost of performance.

Architecture research in datacenter processor design has spurred multiple academic efforts to develop benchmark suites for datacenter computing. Most notably, CloudSuite is a mixture of scale-out cloud service workloads, characterized on a modern server system [14]. Recent efforts have provided in-depth microarchitectural characterization of portions of CloudSuite [49]. Some of our findings (very low bandwidth utilization) are well-represented in CloudSuite benchmarks, others – to a lesser extent (large i-cache pressure), while yet others are markedly different (very flat execution profiles versus hotspots). Many follow-up architectural studies unjustly focus only on the `Web Search` portion of CloudSuite. This can lead to false conclusions, because: (i) websearch is not the sole "killer workload" in the datacenter; and (ii) CloudSuite `Web Search` is the least correlated with our findings from a live WSC (it sees very low stall times, has a tiny L2 instruction working set, and, as a result, achieves very high IPC more representative of a compute-bound workload [14]). Similarly, DCBench focuses in more depth on cloud data analytics [23]. These suites are vital for experimentation, though they cannot be as comprehensive as observing production applications evolve at scale over the years.

Other researchers have also taken the approach of profiling live datacenters. Kozyrakis et al. present data on internet-scale workloads from Microsoft – Hotmail, Cosmos, and Bing, but their study focuses more on system-level Amdahl ratios rather than microarchitectural implications [27]. Another paper [5] similarly focuses on system issues for Google websearch. While it has some discussion of microarchitecture, this study is now more than a decade old. A large body of work profiles production warehouse-scale applications with the explicit purpose of measuring [24] and reducing [35, 50] contention between co-scheduled jobs, or of scheduling them in accordance with machine characteristics [34]. Such studies can benefit from microarchitectural insights provided here.

Finally, our work builds on top of existing efforts to profile and analyze applications on modern hardware. Google-Wide-Profiling provides low-overhead performance sampling across Google's datacenter fleet and has been deployed for many years to provide the capability for longitudinal studies [44]. We also leverage recent advances in Top-Down performance analysis [48] that allow us to estimate CPI stacks without specialized hardware support [13].

10. Conclusions

To better understand datacenter software performance properties, we profiled a warehouse-scale computer over a period of several years. In this paper, we showed detailed microarchitectural measurements spanning tens of thousands of machines, running thousands of different applications, while executing the requests of billions of users.

These workloads demonstrate significant diversity, both in terms of the applications themselves, and within each individual one. By profiling across binaries, we found common low-level functions ("datacenter tax"), which show potential for specialized hardware in a future server SoC. Finally, at the microarchitectural level, we identified a common signature for WSC applications – low IPC, large instruction footprints, bimodal ILP and a preference for latency over bandwidth –

which should influence future processor designs for the datacenter. These observations motivate several interesting directions for future warehouse-scale computers. The table below briefly summarizes our findings and potential implications for architecture design.

Finding	Investigation direction
`workload diversity`	Profiling across applications.
`flat profiles`	Optimize low-level system functions.
`datacenter tax`	Datacenter specific SoCs (protobuf, RPC, compression HW).
`large (growing) i-cache footprints`	I-prefetchers, i/d-cache partitioning.
`bimodal ILP`	Not too "wimpy" cores.
`low bandwidth utilization`	Trade off memory bandwidth for cores. Do not use SPECrate.
`latency-bound performance`	Wider SMT.

Summary of findings and suggestions for future investigation.

Acknowledgments

We would like to thank the anonymous reviewers and Ahmad Yasin for their constructive feedback. We reserve special thanks for our colleagues at Google, and especially: the GWP team for developing and maintaining large-scale profiling infrastructure; David Levinthal and Stephane Eranian for their invaluable help with performance counters; and Luiz Barroso, Artur Klauser and Liqun Cheng for commenting on drafts of this manuscript.

Svilen Kanev's academic work was partially supported by C-FAR, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA. Juan Pablo Darago's academic work was supported by the LICAR lab in Departamento de Ciencias de la Computación, Universidad de Buenos Aires.

References

[1] David G Andersen, Jason Franklin, Michael Kaminsky, Amar Phanishayee, Lawrence Tan, and Vijay Vasudevan. FAWN: A fast array of wimpy nodes. In *Operating systems principles (SOSP)*, 2009.

[2] Jennifer Anderson, Lance Berc, George Chrysos, Jeffrey Dean, Sanjay Ghemawat, Jamey Hicks, Shun-Tak Leung, Mitch Lichtenberg, Mark Vandevoorde, Carl A Waldspurger, et al. Transparent, low-overhead profiling on modern processors. In *Workshop on Profile and Feedback-Directed Compilation*, 1998.

[3] Murali Annavaram, Jignesh M. Patel, and Edward S. Davidson. Call graph prefetching for database applications. *Transactions of Computer Systems*, 2003.

[4] Luiz André Barroso, Jimmy Clidaras, and Urs Hölzle. The datacenter as a computer: an introduction to the design of warehouse-scale machines. *Synthesis Lectures on Computer Architecture*, 2013.

[5] Luiz André Barroso, Jeffrey Dean, and Urs Hölzle. Web search for a planet: The google cluster architecture. *IEEE Micro*, 2003.

[6] Paolo Calafiura, Stephane Eranian, David Levinthal, Sami Kama, and Roberto Agostino Vitillo. GOoDA: The generic optimization data analyzer. In *Journal of Physics: Conference Series*, 2012.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Operating Systems Design and Implementation (OSDI)*, 2006.

[8] Dehao Chen, Neil Vachharajani, Robert Hundt, Shih-wei Liao, Vinodha Ramasamy, Paul Yuan, Wenguang Chen, and Weimin Zheng. Taming hardware event samples for FDO compilation. In *Code generation and optimization (CGO)*, 2010.

[9] Zefu Dai, Nick Ni, and Jianwen Zhu. A 1 cycle-per-byte XML parsing accelerator. In *Field Programmable Gate Arrays*, 2010.

[10] Arnaldo Carvalho de Melo. The new linux 'perf' tools. In *Slides from Linux Kongress*, 2010.

[11] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 2013.

[12] Filipa Duarte and Stephan Wong. Cache-based memory copy hardware accelerator for multicore systems. *IEEE Transactions on Computers*, 2010.

[13] Stijn Eyerman, Lieven Eeckhout, Tejas Karkhanis, and James E Smith. A top-down approach to architecting cpi component performance counters. *IEEE Micro*, 2007.

[14] Michael Ferdman, Babak Falsafi, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, and Anastasia Ailamaki. Clearing the clouds. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[15] T.B. Ferreira, R. Matias, A. Macedo, and L.B. Araujo. An experimental study on memory allocators in multicore and multithreaded applications. In *Parallel and Distributed Computing, Applications and Technologies (PDCAT)*, 2011.

[16] Google. Bazel. `http://bazel.io/`.

[17] Google. gRPC. `http://grpc.io/`.

[18] Google. Protocol buffers. `https://developers.google.com/protocol-buffers/`.

[19] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. 2012.

[20] Aamer Jaleel. Memory characterization of workloads using instrumentation-driven simulation–a Pin-based memory characterization of the SPEC CPU2000 and SPEC CPU2006 benchmark suites. *Intel Corporation, VSSAD*, 2007.

[21] Aamer Jaleel, Joseph Nuzman, Adrian Moga, Simon C Steely Jr, and Joel Emer. High Performing Cache Hierarchies for Server Workloads. In *High-Performance Computer Architecture (HPCA)*, 2015.

[22] Vijay Janapa Reddi, Benjamin C Lee, Trishul Chilimbi, and Kushagra Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. *Computer Architecture (ISCA)*, 2010.

[23] Zhen Jia, Lei Wang, Jianfeng Zhan, Lixin Zhang, and Chunjie Luo. Characterizing data analysis workloads in data centers. In *Workload characterization (IIWSC)*, 2013.

[24] Melanie Kambadur, Tipp Moseley, Rick Hank, and Martha A Kim. Measuring interference between live datacenter applications. In *High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[25] Svilen Kanev, Kim Hazelwood, Gu-Yeon Wei, and David Brooks. Tradeoffs between Power Management and Tail Latency in Warehouse-Scale Applications. In *Workload Characterization (IISWC)*, 2014.

[26] Aasheesh Kolli, Ali Saidi, and Thomas F. Wenisch. RDIP: Return-address-stack Directed Instruction Prefetching. In *Microarchitecture (MICRO)*, 2013.

[27] Christos Kozyrakis, Aman Kansal, Sriram Sankar, and Kushagra Vaid. Server engineering insights for large-scale online services. *IEEE Micro*, 2010.

[28] Snehasish Kumar, Arrvindh Shriraman, Viji Srinivasan, Dan Lin, and Jordan Phillips. SQRL: Hardware Accelerator for Collecting Software Data Structures. In *Parallel architectures and compilation (PACT)*, 2014.

[29] Sangho Lee, Teresa Johnson, and Easwaran Raman. Feedback directed optimization of tcmalloc. In *Proceedings of the workshop on Memory Systems Performance and Correctness*, 2014.

[30] Penny Li, Jinuk Luke Shin, Georgios Konstadinidis, Francis Schumacher, Venkat Krishnaswamy, Hoyeol Cho, Sudesna Dash, Robert Masleid, Chaoyang Zheng, Yuanjung David Lin, et al. A 20nm 32-Core 64MB L3 cache SPARC M7 processor. In *Solid-State Circuits Conference (ISSCC)*, 2015.

[31] Kevin Lim, Parthasarathy Ranganathan, Jichuan Chang, Chandrakant Patel, Trevor Mudge, and Steven Reinhardt. Understanding and designing new server architectures for emerging warehouse-computing environments. In *Computer Architecture (ISCA)*, 2008.

[32] Pejman Lotfi-Kamran, Boris Grot, Michael Ferdman, Stavros Volos, Onur Kocberber, Javier Picorel, Almutaz Adileh, Djordje Jevdjic, Sachin Idgunji, Emre Ozer, et al. Scale-out processors. In *Computer Architecture (ISCA)*, 2012.

[33] Krishna T Malladi, Benjamin C Lee, Frank A Nothaft, Christos Kozyrakis, Karthika Periyathambi, and Mark Horowitz. Towards energy-proportional datacenter memory with mobile DRAM. *Computer Architecture (ISCA)*, 2012.

[34] Jason Mars and Lingjia Tang. Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers. In *Computer Architecture (ISCA)*, 2013.

[35] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Microarchitecture (MICRO)*, 2011.

[36] David Meisner, Christopher M Sadler, Luiz André Barroso, Wolf-Dietrich Weber, and Thomas F Wenisch. Power management of online data-intensive services. In *Computer Architecture (ISCA)*, 2011.

[37] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. *Very Large Data Bases (VLDB)*, 2010.

[38] Dmitry Namiot and Manfred Sneps-Sneppe. On micro-services architecture. *Open Information Technologies*, 2014.

[39] Jian Ouyang, Hong Luo, Zilong Wang, Jiazi Tian, Chenghui Liu, and Kehua Sheng. FPGA implementation of GZIP compression and decompression for IDC services. In *Field-Programmable Technology (FPT)*, 2010.

[40] Mike P Papazoglou and Willem-Jan Van Den Heuvel. Service oriented architectures: approaches, technologies and research issues. *The VLDB journal*, 2007.

[41] David A Patterson. The data center is the computer. *Communications of the ACM*, 2008.

[42] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA)*, 2014.

[43] Moinuddin K Qureshi and Yale N Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Microarchitecture (MICRO)*, 2006.

[44] Gang Ren, Eric Tune, Tipp Moseley, Yixin Shi, Silvius Rus, and Robert Hundt. Google-Wide Profiling: A Continuous Profiling Infrastructure for Data Centers. *IEEE Micro*, 2010.

[45] 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. RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization. In *Microarchitecture (MICRO)*, 2013.

[46] Jan Van Lunteren, Ton Engbersen, Joe Bostian, Bill Carey, and Chris Larsson. XML accelerator engine. In *Workshop on High Performance XML Processing*, 2004.

[47] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *European Conference on Computer Systems (EuroSys)*, 2015.

[48] Ahmad Yasin. A Top-Down method for performance analysis and counters architecture. *Performance Analysis of Systems and Software (ISPASS)*, 2014.

[49] Ahmad Yasin, Yosi Ben-Asher, and Avi Mendelson. Deep-dive Analysis of the Data Analytics Workload in CloudSuite. In *Workload characterization (IISWC)*, 2014.

[50] Xiao Zhang, Eric Tune, Robert Hagmann, Rohit Jnagal, Vrigo Gokhale, and John Wilkes. CPI^2: CPU performance isolation for shared compute clusters. In *European Conference on Computer Systems (EuroSys)*, 2013.