DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Efficiency in warehouse-scale computers: a datacenter tax study

Permanent link
http://nrs.harvard.edu/urn-3:HUL.InstRepos:37945003

Terms of Use
This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Share Your Story

The Harvard community has made this article openly available.
Please share how this access benefits you. Submit a story .

Accessibility

Efficiency in warehouse-scale computers: a datacenter tax study

A DISSERTATION PRESENTED
BY
SVILEN NIKOLAEV KANEV
TO
THE SCHOOL OF ENGINEERING AND APPLIED SCIENCES

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SUBJECT OF
COMPUTER SCIENCE

HARVARD UNIVERSITY
CAMBRIDGE, MASSACHUSETTS
SEPTEMBER 2016

©2016 – Svilen Nikolaev Kanev
all rights reserved.

Dissertation advisor: Professor David Brooks and Gu-Yeon Wei Svilen Nikolaev Kanev

Efficiency in warehouse-scale computers: a datacenter tax study

Abstract

Computation has been steadily migrating from isolated on-premise deployments to the datacenters of a small number of large-scale cloud providers. The datacenters powering the cloud, also known as warehouse-scale computers (WSCs), have a unique set of design constraints, balancing efficiency at scale with ever-growing application needs for performance. Designing next-generation server platforms for WSCs after the end of Dennard scaling is one of the most important challenges for computer architects.

In order to guide such future designs, we performed the first (to the best of our knowledge) longitudinal profiling study of a live production WSC. Our performance measurements span tens of thousands of machines over several years, while these machines serve the requests of billions of users. Even though we observe significant diversity, both in applications and architectural behaviors, patterns begin to emerge. We identify the "datacenter tax" – a set of shared low-level software components that comprise almost 30% of all processor cycles in production datacenters. The constituents of this "tax" – the necessary components to do distributed computation (data serialization, compression, etc.) – are also prime candidates for optimization, both in software and through specialized hardware. The latter case has especially high potential upside, but requires hardware accelerators that are markedly different from traditional designs. These new "broad" accelerators face a unique set of challenges: because calls to tax routines tend to be frequent, fast, and interspersed inside other application code, accelerators must be optimized for latency rather than throughput, and because each one accelerator

brings a limited amount of overall application speedup, overheads must be kept to a bare minimum. We demonstrate by construction that, while non-trivial, meeting such constraints is possible. Our memory allocation accelerator, Mallacc, reduces the latency of already fast `malloc` calls by up to 50% while occupying only 0.006% of the silicon area of a typical high-performance core.

This thesis identifies the opportunity for broad acceleration and presents first steps towards designing datacenter tax accelerators. We expect that it will spur additional interest, from industry and academia, and will help bridge the gap between research in datacenters and in specialized hardware.

Contents

Previous Work

Portions of this dissertation appear in the following:

1. S. Kanev, G. Wei and D. Brooks, "XIOSim: Power-Performance Modeling of Mobile x86 Cores", International Symposium on Low Power Electronics and Design (ISLPED), July 2012.

2. S. Campanoni, K. Brownell, S. Kanev, T. Jones, G. Wei and D. Brooks "HELIX-RC: An Architecture-Compiler Co-Design for Automatic Parallelization of Irregular Programs", International Symposium on Computer Architecture (ISCA), June 2014.

3. S. Kanev. K. Hazelwood, G. Wei and D. Brooks, "Tradeoffs between Power Management and Tail Latency in Warehouse-Scale Applications", International Symposium on Workload Characterization (IISWC), October 2014.

4. S. Kanev, J. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G. Wei and D. Brooks, "Profiling a Warehouse-Scale Computer", International Symposium on Computer Architecture (ISCA), June 2015.

5. S. Kanev, J. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G. Wei and D. Brooks, "Profiling a Warehouse-Scale Computer", IEEE Micro's Top Picks in Computer Architecture Conferences (TopPicks), June 2016.

6. S. Kanev, S. Xi, G. Wei and D. Brooks, "Mallacc: Accelerating Memory Allocation", International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 2017.

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. 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 datacenters, referred to as warehouse-scale computers (WSCs) [8]. These warehouse-scale computers have grown out of applications that cannot reasonably fit in a single machine due to either large resource requirements, or stringent performance and fault-tolerance constraints that dictate the need for a distributed system. The design challenges for warehouse-scale computers are quite different from those for traditional supercomputers or hosting farms, 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 WSCs are a distinctly new class of computer systems that architects must design to [49]: "the datacenter is the computer"[95].

Economies of scale are without a doubt the enabler for WSCs, but they can also hinder their continued improvement. Through reusing physical and power delivery infrastructure, as well as customized server designs, WSCs have reached costs per computation significantly lower than those of desktops or small rack-scale clusters [44, 114]. On top of that, even seemingly small relative improvements become significant once deployed across full server fleets reportedly in the millions of machines [48].

However, research on finding these improvements has a high startup price – at reported construction costs of $2 billion [120], one does not simply build a datacenter to experiment with.

The first two parts of this dissertation aim to address the high startup cost of research by characterizing real, production-grade WSC applications and systems. They are (mostly) observational studies on the interactions between WSC applications and server processors with regards to energy efficiency (Chapter 2) and performance (Chapter 3). While they do expose opportunities for optimization and several prototype implementations, the emphasis is on the characterization itself.

In Chapter 2, we outline the inherent tradeoff between aggressive processor power management and quality of service (QoS) – the dominant metric of performance in datacenters. We examine 15 benchmarks representing workloads from Google's datacenters on contemporary WSC servers. We show that power management techniques have brought server designs close to achieving energy-proportional computing since power became a major design constraint in the mid '00s. To achieve that, a large fraction of these applications very frequently toggle their cores between short bursts of activity and sleep. In doing so, they stress sleep selection algorithms and can cause tail latency degradation or missed potential for power savings of up to 10% in the case of web search. However, even after these additional gains are potentially realized, an "energy proportionality gap" still remains. In an effort to further reduce it, we profile datacenter applications for susceptibility to dynamic voltage and frequency scaling (DVFS). We find the largest potential in DVFS which is cognizant of latency/power tradeoffs on a workload-per-workload basis.

Chapter 3 shifts the focus to performance. With the performance gains from Dennard scaling gone and Moore's law tapering (recently dubbed "the winter of despair" [109]), understanding the interactions of server applications with the underlying microarchitecture becomes crucial, both for extracting maximum efficiency out of existing hardware, and for designing future processors for the datacenter. To aid such understanding, we set up a detailed microarchitectural study of live datacenter jobs, measured on more than 20,000 Google machines over a three year period, and comprising thousands of different applications. We find a common microarchitectural signature for WSC applications – typical workloads place significant stress on instruction caches and prefer memory latency over bandwidth; they also stall cores often, but compute heavily in bursts – and detail several general-purpose processor optimization directions that can take advantage of it.

We also first find that WSC workloads are extremely diverse, creating 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. We call the collection of these blocks the "datacenter tax" and show that it 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.

Before we demonstrate one instance of such specialization, we take a detour to methodology in Chapter 4. Due to the high cost of building hardware, virtually any architectural change is prefaced by simulation experiments. Thus, accurate and extensible simulation infrastructure is a common precondition for architecture studies. To that end, we developed XIOSim, a rich and precise x86 microarchitectural model, which was designed specifically with low-level microarchitecture studies in mind. We describe some of the major design decisions behind XIOSim which enable it to handle workloads of significant complexity, usually reserved for less detailed simulation models. XIOSim is the result of several years' worth of efforts in accurately modelling current-generation x86 hardware, and we describe the methodology for such validation. Having such a model with a realistic baseline allows us to predict the relatively small general-purpose performance changes that one can expect in the post-Moore's law era.

Finally, in Chapter 5, we present an in-depth evaluation of designing hardware specifically for the largest component of the "datacenter tax" – memory allocation. We propose Mallacc, an in-core hardware accelerator designed for broad use across a number of high-performance, modern memory allocators. The design of Mallacc is quite different from traditional throughput-oriented hardware accelerators. Because memory allocation requests tend to be very frequent, fast, and interspersed inside other application code, accelerators must be optimized for latency rather than throughput and area overheads must be kept to a bare minimum. Mallacc accelerates the three primary operations of a typical memory allocation request: size class computation, retrieval of a free memory block, and sampling of memory usage. Our simulated results show that malloc latency can be reduced by up to 50% with a hardware cost of less than 1500 μm^2 of silicon area, less than 0.006% of a typical high-performance processor core.

2

Tradeoffs between power management and tail latency

2.1 Energy proportionality in datacenters

Datacenter operators are faced with an inherent tradeoff between managing power consumption and providing predictable performance. The scale of a WSC makes it a prime candidate for techniques that reduce server power – the aggregate energy savings are beneficial from both a cost and an environmental perspective. A typical WSC workload, such as web search, or ad serving, is comprised of a tree of individual services, each with their respective service-level agreement (SLA) for performance. Applying power saving techniques on the servers responsible for those services can adversely affect performance and lead to invalidation of SLAs. Prior work on web search from Microsoft [55] and Google [87] outlines this tradeoff between power efficiency and request latency.

Power efficiency of datacenters, and WSCs in particular, has been the target of a significant body of research [7, 12, 47, 101]. It is well-established that datacenters spend a large portion of time underutilized. For example, Barroso and Hölzle show a typical CPU utilization of 5,000 Google servers in the 10-50% range over 6 months [8]. This utilization variance is caused by changing user demand and difficulties in inter-datacenter load-balancing. Figure 2.1 also demonstrates the varying user demand in a Google production cluster in North America running content ad matching. Notice that for extended

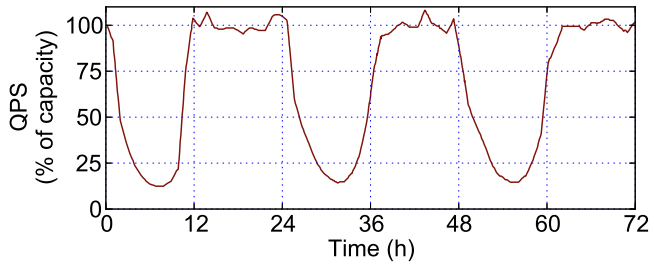


Figure 2.1: Utilization of content ad matching in a production cluster, measured in incoming queries per second (QPS), normalized by the allocated cluster capacity.

periods of time incoming requests (measured in queries per second, QPS) spawn the very wide range of 15-105% of allocated capacity.

Handling such large swings in utilization in an energy-efficient manner is the main motivation behind energy proportionality [7]. In an ideal energy-proportional system, server power consumption would perfectly track the arrival patterns of requests shown in Figure 2.1. Achieving proportionality requires that most power-hungry components be able to scale down their power consumption with usage.

Today's server systems are far from energy-proportional [8, 87]. Figure 2.3 verifies this claim on three contemporary server platforms running a websearch leaf. If these platforms were proportional, their power consumption would follow the dashed line which scales linearly from zero to maximum power. This is not the case for the platforms in question. At a utilization of 50% they consume ≈80% of the maximum system power, not 50%, creating an `energy proportionality gap` [122]. Note that, similar to observations by Wong and Annavaram [122], there is no drastic difference between the three platforms in terms of power scaling, except for the data point at 0% usage, where `Platform A` (the oldest of the three) consumes significantly more power.

In recent WSC platforms, the processor consumes the largest portion of system power [8, 55]. While DRAM power was considered as a challenger for this dominant position, recent advances in memory power efficiency (especially the broader adoption of the low-voltage DDR3L standard) have changed this trend. Figure 2.4 demonstrates that for a contemporary server platform: at full load the processors consume 78% of the system power.[1] Furthermore, just the dynamic range of processor power between fully idle and fully loaded is 67% of the maximum system power, compared to a dynamic range of 10% for all other components combined (from Figures 2.2 and 2.4). This suggests that the processor has the largest potential to bridge the energy proportionality gap in Figure 2.3, especially at mid-range

[1] Power distribution data is measured at sense resistors before component regulators on a 16-core, Intel SandyBridge platform with 256GB of DRAM.

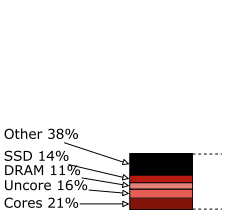


Figure 2.2: Idle server power is evenly split between components.

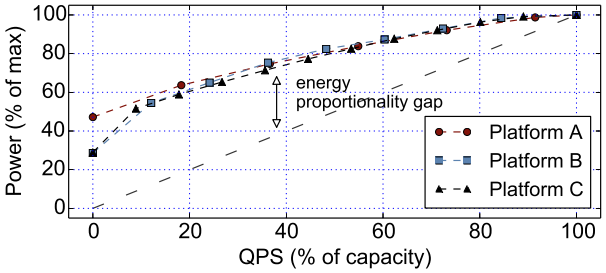


Figure 2.3: Energy proportionality (power as a function of incoming load) across three successive x86 server platforms has remained virtually unchanged.

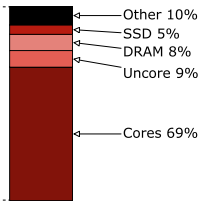


Figure 2.4: Processor power dominates on a fully loaded machine.

utilization. In the rest of this chapter, we focus on processor power management.

We characterize two complementary mechanisms for power management: idle and active. In the first one, idle power states (C-states), decrease core power when no threads have claimed a core and it is executing the operating system idle loop. Processors expose different C-states to the OS that trade sleep/wake-up latency for power savings by powering down different parts of the core and its corresponding caches. In a distributed system, the longer sleep/wake-up latencies of aggressive sleep states can fall on the critical path of incoming requests, and subsequently increase request latency. Thus, a WSC has conflicting requirements between aggressive power savings and aggravated request latency (and missing SLAs). In this light, we address the following questions:

- What are the sleep patterns of current WSC applications? Are applications' idle periods short enough to be affected by the choice of a particular sleep state?
- How much does C-state selection influence request latency and system power on the macro level? In other words, by how much can proper selection improve latency, and what are the maximal power savings from using idle periods?

We evaluate 15 benchmarks based on Google production workloads. We show that for a fraction of them, sleep activity is sufficiently coarse-grained that their latency response is not affected by the choice of a particular sleep state. However, fine-grained sleep activity does exist for certain applications. For them, choosing inappropriate sleep states can result in a latency or system power cost of up to 10%.

Aggressive idle power management, which is already included in contemporary processors, is however not sufficient to achieve an energy-proportional system. Thus, we turn our focus on active power management, which slows down cores while they are busy with execution. We evaluate the potential benefits of WSC applications from reducing voltage and frequency during memory-bound phases of execution.

We first show that the same set of WSC benchmarks is highly memory-bound on average, suggesting that active management through dynamic voltage and frequency scaling (DVFS) can be efficient. We identify a wish list of characteristics that a practical DVFS solution in the datacenter should abide by. We then show that it is unlikely to satisfy everything on this wish list simultaneously by implementing a prototype system. Finally, we identify that the directions holding the largest promise are workload-specialized and ultra-fine-grained DVFS, which warrant further study.

2.2 Idle management and latency

We briefly describe the mechanisms for idle power management, which is largely responsible for current systems' power savings at low load. While idle management saves significant energy, we show that it can also cause latency degradation, given workloads that are bursty enough.

The mechanism for processor idle power management involves shutting down cores (or whole sockets) without work to do. Figure 2.5 illustrates the process of entering a core idle state (C-state): after there is no work to be scheduled in userspace, the kernel executes a specific instruction (`mwait` on x86), with a parameter indicating the requested C-state. In Linux terminology, the logic to select the appropriate C-state is called a *governor*.

The trade-off made by C-state governors is between power savings and wake-up latency. Deeper C-states save more power by power gating larger portions of the chip, but require a longer wake-up time (and potentially more energy) to restore state [80]. The minimum idle period for a specific C-state to be profitable energy-wise is referred to as *target residency*. For example, Intel's SandyBridge microarchitecture exposes 5 core C-states [106]:

state	residency	wake-up latency
C0	*(active)*	*(active)*
C1	1 μs	1 μs
C3	106 μs	80 μs
C6	345 μs	104 μs
C7	345 μs	109 μs

The kernel governor is not the only system in charge of idle power management. Recent architectures include a shared power control unit (PCU), whose purpose is to orchestrate power management for the processor. The PCU can ignore software requests for a specific C-state, choosing to enter a shallower one, if it estimates that the residency requirement of the deeper state will not be met. This behavior is called *C-state demotion* and is controlled by a proprietary algorithm set by processor ven-

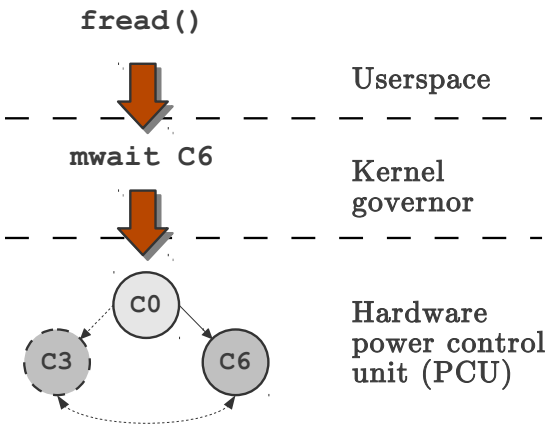


Figure 2.5: The different layers of the hardware/software stack involved in managing idle power states. Dashed lines indicate state transitions only possible in hardware.

dors. Furthermore, the PCU can choose to transition a core between different sleep states without waking it up – a knob not available to software.

Thus, idle power can be independently managed both in hardware and software. Both approaches have their benefits. The PCU has a closer and more fine-grained view of different cores' power consumption, as well as finer-grained control knobs. On the other hand, software has a global, non-processor-centric view of the full system, and can predict future events, such as incoming disk interrupts, and react accordingly. Combining management on the two layers such that they co-operate and do not get involved in "power struggles" [100] requires a detailed understanding of both of them in isolation. For the remainder of this chapter, we focus predominantly on the software layer.

Latency cost Idle power states can save a significant amount of system power [2, 67, 87]. They are largely responsible for the power scaling of current platforms demonstrated in Figure 2.3 (for example, the sharp "knee" of the curve for `Platform C` near 12% QPS is the result of a whole socket being able to go idle at the same time). However, selecting the optimal sleep state requires accurate prediction of sleep length. Predicting an idle period too short may cause missed opportunities for power savings, if a deeper sleep state is available. Similarly, predicting it too long may cause a premature wake-up, adding the state wake-up latency to the already time-critical interrupt processing.

We refer to the second effect as the *latency cost of sleep*. Figure 2.6 illustrates how significant that cost can be. It shows the median round-trip latency of a remote procedure call (RPC) layer microbenchmark at different loads. [2] The resulting workload is ideal for investigating the latency effects of idle

[2] Specifically, one server sends a small (several bytes) payload over the network and waits for a response,

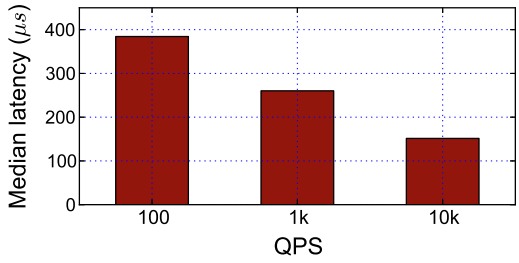


Figure 2.6: Round-trip latency degradation for a RPC transport layer for varying queries per second (QPS). Too aggressive sleep states (at low QPS) significantly degrade request latency.

states – CPU utilization is low, so cores sleep often, and request processing times are smaller than 100μs, resulting in very fine-grained sleep behavior.

Under low load (100 QPS), the cores on the critical path of computation are idle for a significant fraction of time and do enter deep sleep states. This results in an overall 2.4× increase in request latency compared to the high-load (10,000 QPS) case. Such latency degradation is unintuitive – in the canonical distributed system governed by queuing effects increasing incoming request rates leads to higher latencies, not lower.

We want to verify whether such sleep effects manifest themselves on macro-scale benchmarks. In order for them to be comparable to more typical queuing effects, request processing latencies, or query interarrival rates, must be comparable to deep C-state residencies – on the order of hundreds of μs. In other words, the workload must be "bursty". In examining C-states, LeSueur and Heiser [67] notice that an Apache web server exhibits such bursts with lengths smaller than 1ms. Furthermore, Meisner et al. [86] show that a web search cluster can be modelled with an average query interarrival rate of 300μs. This motivates a more detailed characterization of WSC benchmarks, with the aim to determine whether they exhibit sub-millisecond sleep periods, and are therefore susceptible to the latency cost of sleep.

2.3 Examining sleep patterns

In this section, we look into the idleness patterns of current Google applications. This is a necessary first step in determining whether datacenter workloads are potentially vulnerable to the latency cost of deep sleep. Since our ultimate goal is to better understand power vs. tail latency trade-offs, we select

measuring the round-trip latency. The receiver dedicates several cores to handling network interrupts and to forwarding the payload to a single core on the same chip, which immediately sends the payload back to the sender.

Name	Description	Relevant metric
latency-insensitive		
`saw`	String parsing in the Sawzall domain-specific language [96]. Test counts words in production logs.	QPS
`openssl`	Encryption test. Several standard encryption algorithms.	QPS
`flight-search`	Flight search and pricing engine.	QPS
`books`	Book scanning perspective correction.	QPS
`page-ranking`	Signaling search relevance by analyzing the hyperlink structure of web pages [91].	QPS
`ml1`	Machine learning framework.	QPS
`ml2`	Alternative machine learning framework for large dataset analysis.	QPS
latency-sensitive, IO-centric		
`sstable`	Immutable, key-value, string-based storage for BigTable data [22].	latency
`bigtable-single`	Scalable, distributed storage [22]. Local single-machine tests.	QPS; latency
`disk`	Low-level distributed storage driver. Test replays access traces from various production services.	QPS; latency
`bigtable`	Multi-machine BigTable test. More closely representative of real usage.	QPS; latency
latency-sensitive, CPU-intensive		
`search1`	Leaf node in a search cluster [87].	latency
`search2`	Alternative search leaf node.	latency
`ml3`	Machine learning framework to group text in meaningful clusters.	QPS; latency
`ads`	Content ad targeting – matches ads with web pages based on page contents.	latency

Table 2.1: Benchmark names and descriptions.

a subset of our benchmarks, which satisfy the constraints of: being latency-sensitive; and requesting idle states on a sub-ms scale. We find that popular applications, like search, or ad serving, fall into this category. For them, we measure the maximum impact that C-state selection can have on tail latency and system power – up to 10-15%.

2.3.1 Experimental setup

Hardware configuration We perform all our experiments on a 2-socket, 16-core Intel SandyBridge-based server, which has a total of 32 thread execution contexts. Its idle power states are as described in Section 2.2. We fix all cores' frequency at 2.6 GHz, disabling frequency up-scaling (referred to as

Intel TurboBoost). This is done because the additional frequency headroom is heavily dependent on idle power management, and controlled by an unknown algorithm in the processor's PCU. Enabling TurboBoost can bring significant additional variance to our measurements.

We also disable *C-state demotion* for similar reasons. Some prior work has shown that demotion can improve performance on some workloads [106]. However, in our experiments, allowing hardware to override software C-state decisions resulted in significant run-to-run variance, large enough to hide any correlation between changing software policies and overall performance results.

Software and workloads We characterize the sleep behavior of a variety of applications. To that end, we capture timing information for every C-state transition as requested by the kernel; as well as for the state's corresponding wake-up transition to the active state C0. This allows us to measure the time each logical core spends at a certain C-state. We also measure total system power at the power supply unit, and average it over the whole benchmark execution.

The requested C-state transitions are captured using `ktrace` [11] in the Linux kernel and collected with `perf` [32] during the regions of interest for the different benchmarks, after a necessary warm-up period. We capture every transition (as opposed to sampling), because we require two consecutive transitions to determine the residency in a given C-state. Since transitions rarely occur more often than once every 10μs, and collection is appropriately buffered, the characterization process does not incur significant overhead (at least significant enough to affect latency for the RPC microbenchmark described in Section 2.2).

We use a variety of workloads that represent stages of large-scale Internet services. While not completely representative of any particular WSC, these applications cover different classes of workloads from large datacenters. The application names and short descriptions are provided in Table 3.1. We split them into three groups based on their latency tolerance, and the amount of IO operations that they perform. The last column in the table lists the performance metrics used to define the quality of service (QoS) for the particular service.

Because these are internal Google binaries, our exact experiments are hard to reproduce externally. This is why we use a large number of workloads, with expected very different sleep durations and confirm which ones show the tradeoffs we expect to see. It is also worth noting that after the initial publication of this work, several researchers have observed qualitativey similar tradeoffs between sleep state management and latency for open-source workloads *with sub-ms sleep activity* (memcached [71, 128], Nginx [71] and synthetic traces [50]).

While in a real-world scenario these services are deployed on a large number of machines, for practicality of our experiments we constrain them to a single server, plus another one for load generation

for the workloads that require it. The resulting load tests retain idiosyncrasies of the live services they model (e.g. bursty and changing incoming traffic). Since single-server tests are prone to run-to-run variance, we repeat all runs at least three times, often more, based on applications' individual characteristics.

2.3.2 The typical sleep duration varies

In order to characterize idle behavior of our benchmarks, we measure the time each logical core spends at every available C-state by recording state transitions in the kernel. We later use this information to filter the benchmarks that are likely to be affected by deep sleep. Since the main purpose of this experiment is to capture the sleep patterns of applications, and not of the hardware platform, we only distinguish between "sleep" and "active" states.

Figure 2.7 shows the results of this characterization. The figures are in histogram format, with the x-axis bins representing the amount of time between transitioning in and out of sleep, and the y-axis showing the fraction of total execution time spent in sleep or active mode. The sum of `Active` bars adds up to the processor utilization of the particular service. In these plots, bursty applications tend to spend a larger fraction of time transitioning between states with shorter residency and show up on the left side of those histograms, while those with very long periods of activity/idleness cluster towards the right.

Latency-insensitive applications We first look into those throughput-oriented benchmarks for which latency is not a relevant metric. For all of them, latency is not important because either they are implemented in a throughput-centric model (such as MapReduce); or they are off the critical path of major services (`openssl`); or the quanta of work over which latency can be defined are too large from an architecture standpoint (e.g. multi-second requests for `flight-search`). Figures 2.7(a)-(g) show the sleep length distribution for such applications. For most of them (except `page-ranking` and `ml2`), idle power management activity is very coarse-grained, with sleep and active periods well longer than 1ms. Since end performance is not sensitive to individual request latency in this case, the latency effects of deep sleep are irrelevant.

Latency-sensitive IO-heavy applications The second group of benchmarks has relatively low CPU usage and generates a large number of IO requests. These benchmarks are mostly different components of BigTable – the scalable distributed storage system at Google [22]. Figures 2.7(h)-(k) show the granularity of their sleep behavior. All benchmarks in this group show a significant fraction of sleeps and bursts of activity with sub-millisecond lengths. In the case that is closest to real usage, the

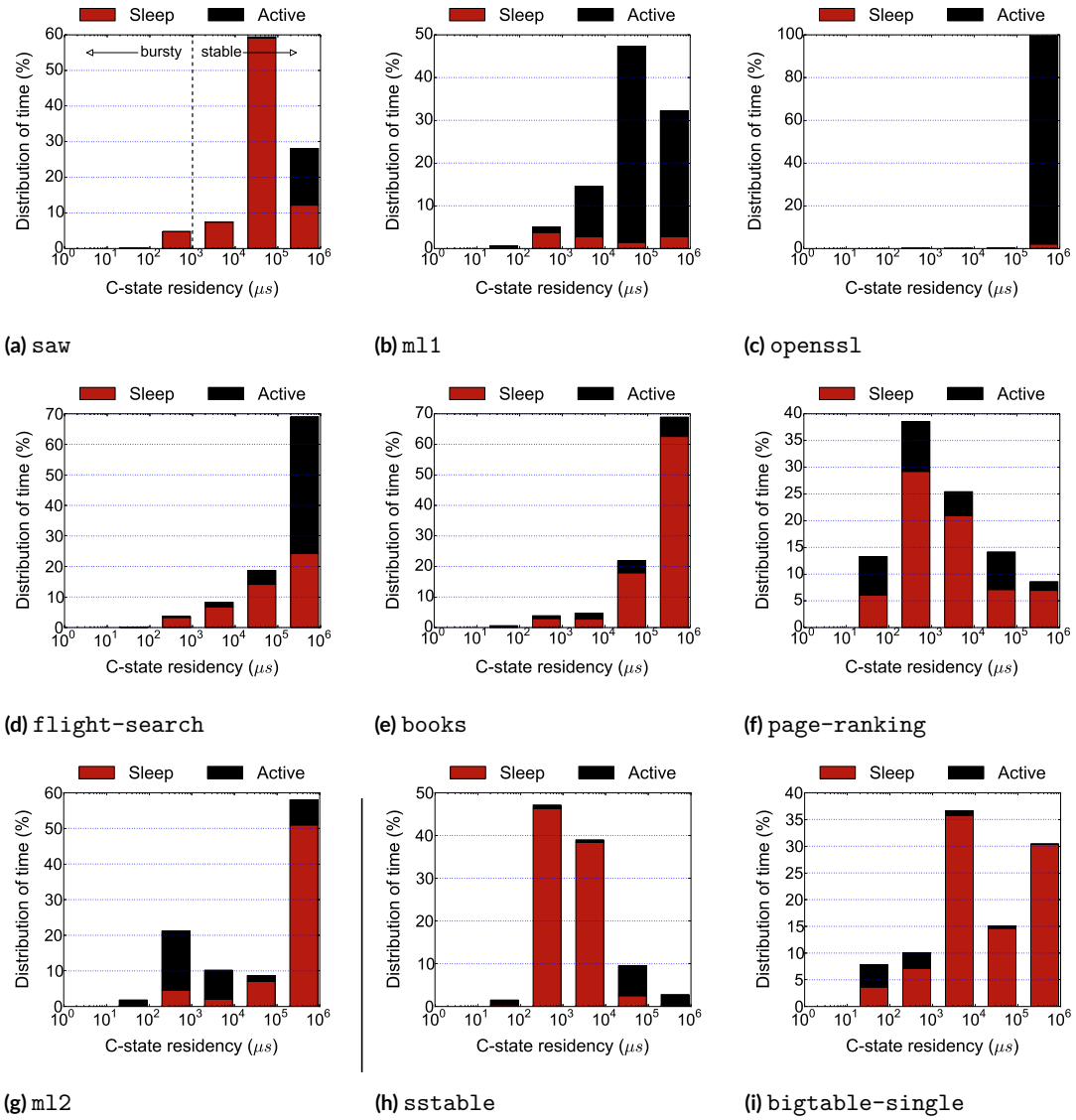
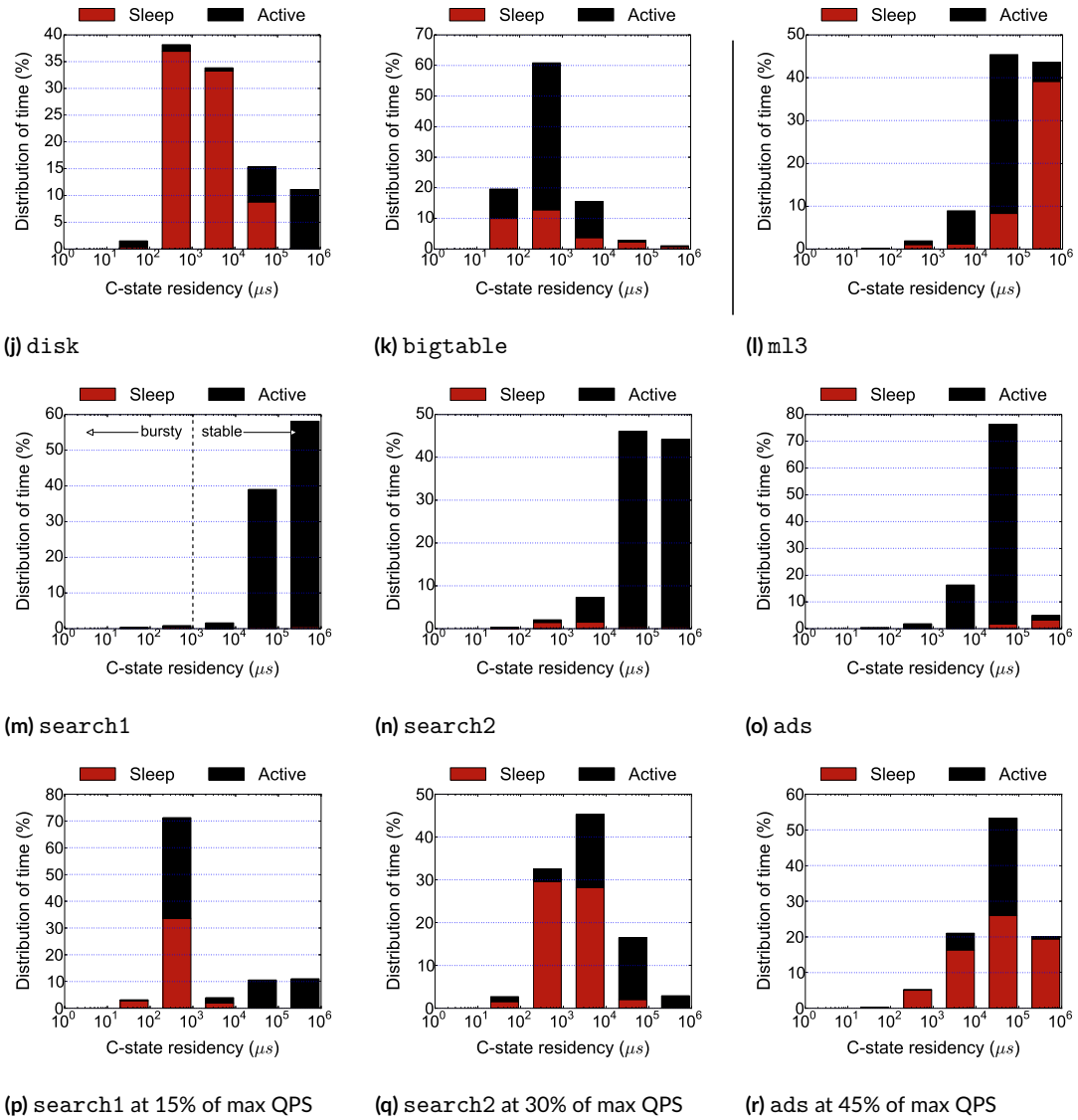


Figure 2.7: Idle state distribution of WSC benchmarks. (a)-(g) **Latency-insensitive** benchmarks: the majority of sleep activity is coarse-grained, with sleep/computation lengths, larger than 1ms. (h)-(k) **Latency-sensitive, IO-centric** benchmarks: a significant amount of execution is spent requesting sleep for short periods of time ($<$1ms).

multiple-machine BigTable test (`bigtable`, Figure 2.7k), short-length activity occurs for more than 80% of the execution time. This confirms the preconception of IO codes being bursty. It also implies that their latency might be significantly affected by C-state selection algorithms.



(j) `disk` **(k)** `bigtable` **(l)** `ml3`

(m) `search1` **(n)** `search2` **(o)** `ads`

(p) `search1` at 15% of max QPS **(q)** `search2` at 30% of max QPS **(r)** `ads` at 45% of max QPS

Figure 2.7: Idle state distribution of WSC benchmarks. (l)-(o) **Latency-sensitive, CPU-intensive** benchmarks: the programs completely occupy the processor (or a subset of cores) for long periods, leaving no room for fine-grained power management. (p)-(r) **Latency-sensitive, CPU-intensive (low QPS)** benchmarks: when emulating low-activity periods, short sleeps begin to emerge.

Latency-sensitive CPU-heavy applications Finally, Figures 2.7(l)-(o) show the group of CPU-bound benchmarks. They either occupy every core in the system (`search1`, `search2`, `ads`), or completely utilize just a subset of cores (the rest). Either case requires very minimal power management.

For the results so far, we assumed that services were fed with the maximum QPS sustainable by a single machine. If we restrict the rate of incoming requests, power management becomes relevant for these applications, too. In fact, this is a more realistic scenario since datacenter server CPU utilization is typically far below 100%, as seen in Section 5.1. Figures 2.7(p)-(r) show the sleep length distribution for the active logical cores of `search1`, `search2` and `ads` when incoming QPS is respectively 15, 30 and 45% of the maximum sustainable by a single server. These particular numbers are chosen to represent CPU utilization in the 30-60% range. Notice that in this case, a significant part of sleep and activity periods is short in length (<1ms). For example, `search1` has sub-millisecond sleep activity for more than 70% of the execution time.

Based on these workload observations, we can select a subset of benchmarks that are likely to be affected by sleep policies, and continue the analysis with them. We choose the major services that show bursty sleep behavior – `bigtable` and the low-QPS variants of `search1`, `search2` and `ads`.

2.3.3 Selecting the proper C-state matters

After identifying macro-level benchmarks that can be affected by C-state management, we can quantify the maximum effect that different governors could have on the benchmark execution. The metrics that we look at are tail request latency and average power. Tail latency here refers to 95-th percentile latency for `search1` and `ads`, and 99-th percentile for `bigtable` and `search2`[3].

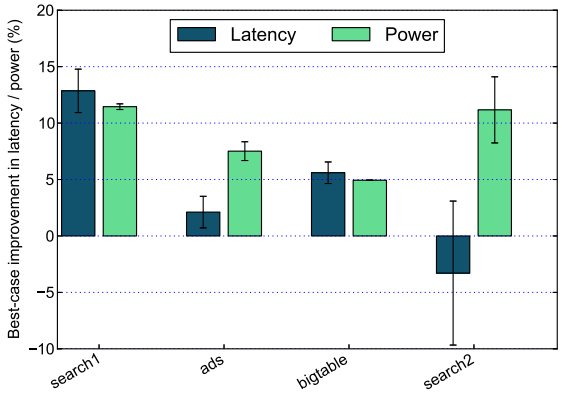
In order to find the maximum improvement in both metrics, we use two trivial C-state governors – `deep` and `shallow`. As their names imply, `deep` always selects the deepest sleep state (C7) whenever a thread is idle, saving a maximal amount of power, but at the same time having the worst effect on latency due to its long residency requirement; while `shallow` always selects the least aggressive state (C1), saving little power, but also increasing latency minimally due to its fast wake-up time.[4]

Thus, the room that a C-state governor has for power savings is at most the difference between `shallow` power and `deep` power. Similarly, the room for latency improvement is given by the difference in `deep` and `shallow` latency. Because of the trade-off between power savings, and wake-up latency, a specific C-state selection algorithm would not realize these maximum gains in request latency and power simultaneously.

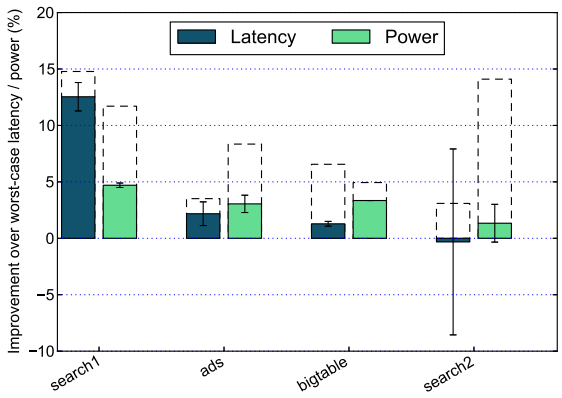
Figure 2.8a shows the amount of potential gains for the four applications. The bars labelled `Latency` measure latency reduction of `shallow` over `deep`; ones labelled `Power` measure power savings

[3]For purely practical reasons – different benchmarks report different percentiles.

[4]We found that keeping the core active in C0 by busy-spinning in the kernel has worse latency effects than `shallow`. This was counter-intuitive, but consistent. Therefore, we use `shallow` as the policy with the least power savings.



(a) `ideal-improvement`



(b) `menu`

Figure 2.8: Maximum improvement in average power and tail latency achievable by C-state selection (a). Achieving the maximum improvement for both power and latency with a single policy is unlikely, as evidenced by the realistic `menu` governor (b).

of `deep` over `shallow`. As expected, the potential power reduction from using deeper C-states are significant, averaging 8.6% across applications. Notice that this is aggregate power consumed by the server, not only by its processors. On the other hand, potential tail latency reduction varies more with the choice of workload.

For both `search1` and `bigtable` the relative improvement in tail latency is significant, and close in value to the potential for system power savings. This implies that realistic C-state selection algorithms can trade off optimizing for any of the two metrics. In the case of `ads`, the potential for improving tail latency is substantially lower. This is consistent with Figure 2.7r, which shows that `ads` has the smallest fraction of bursty sleeps among the four applications examined here.

Interestingly, in the case of `search2`, the average 99-th percentile latency increases when using `shallow` sleep, relative to `deep`, although the run-to-run variance is so high that it is hard to draw any major conclusions. `search2` is known to contain many application-specific optimizations for tail latencies (e.g. keeping busy-spin threads), and some of those customizations might be the reason for degraded performance when sleep is cheap (`shallow`). This effect illustrates the problem of trying to simultaneously optimize a single metric in multiple layers of the hardware/software stack.

Finally, Figure 2.8b shows that current prediction algorithms can realize a large fraction of the ideal gains demonstrated in Figure 2.8a. The solid bars show the real improvements in power/latency of the `menu` governor [92] in the Linux kernel, version 3.7, [5] compared to the best-case gains in Figure 2.8b (dashed lines).

While it is unlikely that the ideal gains in both latency and power can be realized simultaneously, there is still room for improvement in C-state selection. To show that, we tested a simple extension to the `menu` algorithm, wrapping it in a feedback loop that curbs too aggressive sleep. It measures the frequency of pre-mature wakeups, and adjusts the `menu` prediction towards a shallower state if that frequency is above a configurable threshold. That simple change alone provides 5 percentage points (pp) decrease in `bigtable` tail latency, while keeping power within ±1pp of `menu` results over all benchmarks.

While similar changes in C-state selection can help offset worst-case performance degradation (like the one seen in Figure 2.6), even the ideal additional power gains in Figure 2.8a are insufficient to bridge the energy proportionality gap identified in Section 2.1 One obvious direction for that is deeper C-states, which save more power, but do not take longer to resume from. However, such improvements are by no means WSC-specific, and hardware designers have likely already optimized their designs for such win-win opportunities. Thus, we turn our attention to another opportunity for power savings – active management through frequency scaling.

2.4 Frequency scaling

Exploiting periods of inactivity is not the only way to save power. While an application is not sleeping – not given up a core and with instructions in flight – there are still opportunities for power reduction. These come from exploiting memory and last-level cache (LLC) stalls. Dynamic voltage and frequency scaling (DVFS) has been widely studied as a mechanism for reducing power while a core is stalled.

[5] In short, the `menu` algorithm estimates the expected sleep time (by fitting a simple regression model) and selects the appropriate C-state for that time based on an estimated system latency tolerance.

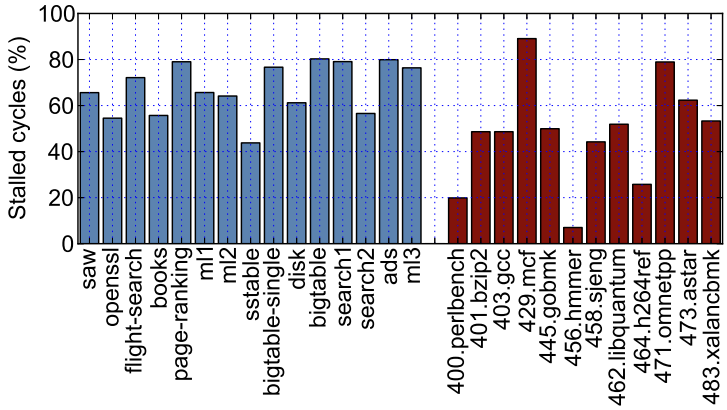


Figure 2.9: WSC benchmarks are more likely to keep cores stalled than traditional SPECint applications.

We will investigate the potential of DVFS which exploits workload phase behavior, ideally without performance overheads and without specific workload adaptation.

While there have been many (mostly academic) proposals to exploit stall periods with active power modes [51, 52, 61, 101, 123, 130], they have not made their way to conventional operating systems. For example, the Linux kernel's support for frequency scaling is based on a different premise. Power-saving frequency governors (`ondemand`, `powersave`) use OS-reported processor utilization as a proxy for the system's latency sensitivity, and scale frequency down when processor utilization is low. While such a heuristic might be useful for the desktop and mobile domains, WSC requirements differ – services can be very sensitive to latency regardless of how high processor utilization is (e.g. `disk` in Figure 2.7j).

Instead, our exploration focuses on memory-bound phases of execution, during which slowing down cores does not affect end performance. It is motivated by the fact that WSC applications appear highly memory-bound on aggregate. Figure 2.9 illustrates that – it measures average stall cycles, that is, cycles when the cores in the system are not sleeping, but also not committing instructions. The majority of WSC applications have stall ratios comparable to those of the most memory-bound SPECint applications (`429.mcf` and `471.omnetpp`), which have also shown largest benefits from DVFS [113]. Some open-source datacenter workloads also show comparable average memory-boundness [39].

This high degree of memory-boundness suggests that WSC applications may benefit significantly from DVFS. In the rest of this section, we postulate a wishlist of requirements that a practical datacenter-ready DVFS scheme would need. We then follow up with a simple prototype implementation, showing that large gains are not realistic without either workload-specific tuning, or fine-grain hardware support.

2.4.1 Ideal requirements

An ideal active power management scheme for a WSC environment would have the following characteristics:

Workload-agnostic It is well-known that different workloads have varying performance headrooms for power savings through DVFS [113]. Some of the factors influencing this headroom are easy to quantify with only global machine knowledge (e.g. memory-boundness through hardware performance counters). Others are more workload-specific. Consider the `search1` example in the bottom plot of Figure 2.10 (for now, focus on any set of circles, which represent average request latency). At low arrival rates, the average latency to handle a request is far from the targeted SLA and aggressive power management is desirable. However, when load is high latencies get dangerously close to the SLA and performance should be prioritized.

A DVFS policy that exploits such workload-specific knowledge can certainly lead to larger power savings than a generic one. But, in the WSC case, it also requires detailed understanding of (and a common software interface to) the performance properties of a potentially large number of workloads. Such complexity is significantly exacerbated in a shared cluster – with many jobs, often co-scheduled together, and having overprovisioned performance requirements [104]. A *workload-agnostic* policy, on the other hand, only uses global machine- and system-level information, without having to address such complexities. We investigate whether such a policy can also lead to power savings.

Zero-tolerant Once a large number of machines are involved in a tiered, high-fanout WSC service tree, individual machine performance variability is significantly amplified at the overall service level. For example, Dean and Barroso illustrate a case where the 99^{th}-percentile latency increases by more than 10× between one random leaf node in such a tree and its parent node (which has to wait for all children to return an answer)[34]. This conventional wisdom leads datacenter operators to the conclusion that performance variability is unacceptable if it affects latency metrics. Then, unless a DVFS policy is able to monitor such latency metrics (which would require workload specificity[6]), it has to be very conservative, and only cause a minimal performance degradation, if any. We call this property *zero-tolerance*.

Thread-granular An ideal DVFS policy for WSC applications is also able to adjust performance states on a per-thread basis. This requirement is obviously beneficial in the shared cluster case, where

[6] ... and possibly significant complexity. One example is the work by Raghavendra et al. [100], which uses a multi-layer coordinated control-theoretic framework that bounds the distribution of request latency.

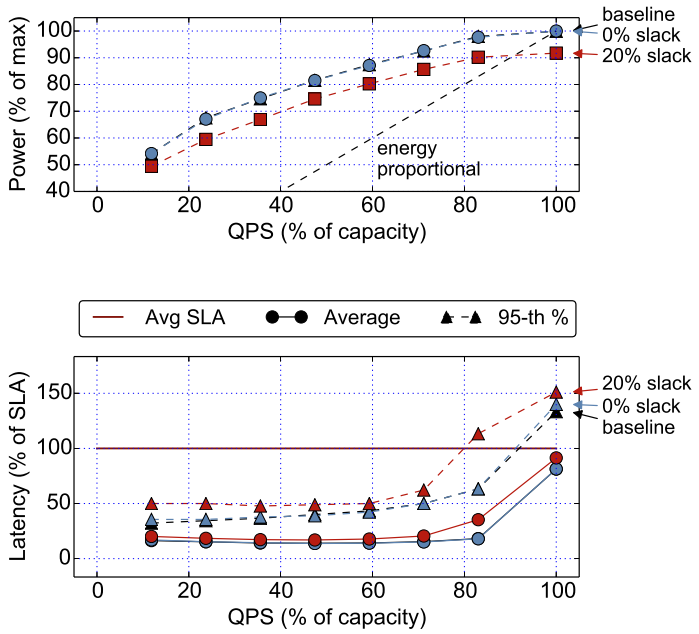


Figure 2.10: DFS on `search1` becomes effective only after exploiting SLA slack (20% slack lines), not in the zero-tolerance case (0% slack lines).

co-scheduling multiple tasks that do not allocate all cores in a machine is the norm [104]. Dedicated clusters would also benefit significantly from per-thread DVFS. They typically run query-driven workloads, where different threads execute different incoming queries, whose execution phases do not necessarily overlap.

Note that implementing thread-granular DVFS on contemporary x86 server processors is challenging. This is mostly due to the limited number of frequency and voltage domains, which only allow scaling voltage and frequency for all cores together [70]. This could change in the future when per-core voltage regulation [62] amplifies the potential power savings.

Fine-grained Finally, different incoming queries are uncorrelated not only among threads, but also within a single thread. This limits the duration of memory-bound phases exploited by DVFS to the latency of a single query. However, the majority of these queries are short – Meisner et al. [87] show an example of more than 90% of queries in a search leaf completing in 5ms.

Previous phase-based DVFS approaches looked into much coarser-grained intervals – e.g. 100M instructions [52] – which would contain many independent (i.e. uncorrelated) requests in a WSC application. Furthermore, in simulation, IPC variability (and, hence, potential DVFS gains) of SPEC

benchmarks has been shown to decrease by more than 100× between intervals of 300 and 10K instructions [102]. This high sensitivity to granularity implies that an efficient WSC policy would benefit significantly from fine-grained control.

2.4.2 Prototyping zero-tolerance DFS

While many DVFS proposals satisfy two or three of the above requirements [51, 52, 61, 101, 123, 130], we are unaware of a system that matches all of them. Thus, we implemented a simple DVFS prototype to (i) attempt to satisfy all four; and (ii) estimate the importance of each one in case it cannot. As expected, our prototype fails to satisfy all of them simultaneously on contemporary server hardware, suggesting there might be a fundamental tradeoff, and that some need to be relaxed in order to realize significant power savings.

The mechanism we used for active power management is clock duty cycle modulation [129]. It allows adjusting per-core effective clock frequencies. Note that it does not adjust voltage (current server processors only have a single core voltage plane), so at best it can achieve power savings linear with frequency. We adjust frequencies within the TurboBoost range (2.6-3.3GHz at ≈200MHz increments for the system described in Section 2.3.1).

The prototype implements a simple algorithm which empirically estimates the sensitivity of performance to frequency, similar to the one proposed by Hsu and Feng [51]. Sensitivity, or CPU-boundness, is defined by a simple linear model of normalized instructions per second (IPS) versus frequency: $Sensitivity = \frac{\Delta IPS}{IPS} \times \frac{f}{\Delta f}$, and used to predict performance at different frequencies. This is done on a per-core basis, with special attention to accounting for hyperthreads.[7]

The sensitivity model is used to predict the IPS decrease from lowering a core's frequency at every time step. If the predicted performance degradation is lower than a target "slack" parameter, the core asks for a lower frequency. A slack value of 0% represents the zero-tolerance policy from the previous section which only exploits highly memory-bound phases during which performance is insensitive to core frequency. Both more complex phase detection mechanisms [52] and frequency selection policies [123] are certainly possible (and have been summarized previously [61]), but are not the purpose of this work. Our prototype implementation is the minimum realistic case that matches the four requirements in Section 2.4.1.

Figure 2.10 shows the results of applying this implementation to `search1` at different incoming QPS rates. The top plot displays power consumption at different values of the slack parameter, while

[7] Furthermore, if there is not enough frequency variability as a result of the control algorithm, the prototype occasionally perturbs cores to the minimum and maximum frequencies, so the sensitivity model does not get stale.

the bottom one illustrates the effects of power savings on average and 95th-percentile latency. Most importantly, zero-tolerance DVFS does not find periods of complete memory-boundness at the 1ms granularity, and does not save power. This is evident from the virtually identical lines labelled "baseline" and "0% slack".

Workload-controlled DVFS Relaxing the zero-tolerance constraint has the expected effect (9% full-system power savings on average at 20% IPS slack), but at a significant increase in tail latency (the line labelled "95-th %"). Relaxing even further, a hypothetical system that includes per-core voltage control can achieve 20% full-system power savings (assuming $P \propto f^{2.4}$ [87]). For some loads, when latency is significantly below the SLA (for example, QPS <80% on Figure 2.10), the increase in latency is completely tolerable – that is, there are no gains from aggressively beating the SLA. Recently, Lo et al. proposed a system that exploits this property for a websearch benchmark, adjusting DVFS aggressiveness based on the difference between observed latency and the latency agreement [77]. As discussed earlier, while such per-workload systems achieve impressive power savings for ubiquitous applications, deploying them across a wide range of different workloads could be challenging.

Phase granularity Another factor that could be limiting the power savings of the tested prototype is the granularity of program phases. For our software implementation, the minimal DVFS period that causes non-negligible performance loss is 1ms. We also ran experiments with 100μs intervals, and, despite the performance penalty of triggering decisions too often, power results were virtually identical to the ones in Figure 2.10. This implies that memory-bound phases in applications like `search1` either do not exist, or manifest themselves on a finer granularity than ≈100K instructions. The latter case is more likely – `search1` is highly stalled on average (Figure 2.9), and simulation studies [62, 102] (albeit on different workloads) have shown that shorter phases have orders of magnitude higher variability in CPU-boundness. Directly confirming the existence of such ultra-fine-grained phases in WSC applications would require at least a separate simulation study (although in Chapter 3 we find strong evidence); exploiting them – hardware support which does not have to pay the overheads of switching to the kernel so often.

The analysis in this section suggests that, while initially appealing, a DVFS solution that is at the same time workload-independent, zero-performance-overhead, fine-grained and per-thread does not work on current server hardware. For significant power gains, one needs to either exploit workload characteristics or use additional hardware which can track extremely short-lived memory-bound phases, or simply tolerate performance variability.

2.5 Conclusion

With the increasing popularity of online services, intelligently managing power for warehouse-scale machines is becoming ever more relevant. We have characterized datacenter workloads, focusing on opportunities to save power at all ranges of processor utilization. We have shown that such workloads are neither completely CPU- nor IO-bound. Instead, they mix bursts of computation with short periods of sleep, emphasizing the need for comprehensive sleep state selection algorithms. We have shown that power savings are possible while not sleeping, too, but only after a careful and workload-specific frequency scaling policy.

3

Profiling a warehouse-scale computer

3.1 Why profile a live datacenter?

At datacenter 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 [39, 126], or system-level characterizations of WSCs [9, 65], do exist, little is known about detailed performance characteristics of at-scale deployments.

This chapter 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 3.2). Even though extracting maximal performance from a WSC requires a careful concert of many system components [8], we

choose to focus on server processors (which are among the main determinants of both system power and performance [59]) 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 3.3). While we find little hotspot behavior within applications, 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 3.4). 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 3.5), 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 3.6). The instruction footprints for many key workloads show significant growth rates ($\approx$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 3.7). 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 3.8), 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.

3.2 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) [45]. Requests and responses are serialized in a common format (at Google, protocol buffers [46]). Latency, especially at the tail end of distributions, is the defining performance metric, and a plethora of techniques aim to reduce it [34].

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 [43]. 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) [105]. 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 [4].

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` [33]), (iii) symbolize the collected sample's callstacks (such that they are

[1] Recently the term "microservices" [89] has been coined to describe such a system architecture. The concept itself predates the term [93].

Binary	Description
`ads`	Content ad targeting – matches ads with web pages based on page contents.
`bigtable`	Scalable, distributed, storage [23].
`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 [9].
`search1, search2, search3`	Search leaf nodes [87].
`video`	Video processing tasks: transcoding, feature extraction.

Table 3.1: Workload descriptions

tagged with corresponding code locations) and (iv) aggregate a large number of such samples from many machines in a Dremel [88] database for easy analysis. The GWP collection pipeline has been described in detail by Ren et al. [105].

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 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 sam-

ples 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.

Performance counter analysis We use a performance analysis methodology, called Top-Down, recently proposed by Yasin [125]. Top-Down allows for reconstructing approximate CPI stacks in modern out-of-order processors, a task considered difficult without specialized hardware support [38]. The exact performance counter expressions that we use are identical with the ones listed in the Top-Down work [125].

Similar to other cycle counting methodologies [16, 38], 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 3.1) 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

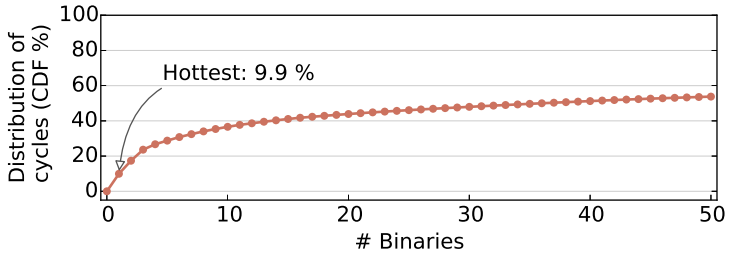


Figure 3.1: There is no "killer application" to optimize for. The top 50 hottest binaries only cover ≈60% of WSC cycles.

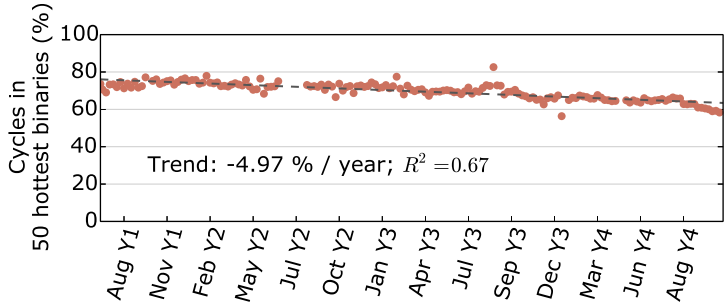


Figure 3.2: Workloads are getting more diverse. Fraction of cycles spent in top 50 hottest binaries is decreasing.

(`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 name (in other words, one unique build target, potentially with multiple deployed versions) and use the two terms interchangeably (Kambadur et al. [57] 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.

3.3 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 [9], 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

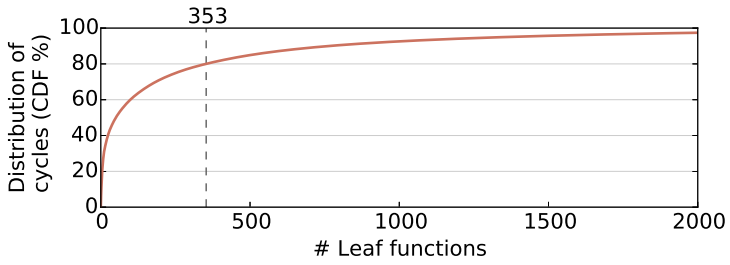


Figure 3.3: Individual binaries are already optimized. Example binary without hotspots, and with a very flat execution profile.

warehouse-scale computers over more than 3 years. Figure 3.1 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 3.1 is just a single slice in time of an ongoing diversification trend. We demonstrate that in Figure 3.2, 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 3.3 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 [126].

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 [27]. 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 this chapter, 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.

3.4 Datacenter tax

Despite the significant workload diversity shown in Section 3.3, 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 3.4 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 3.2), 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 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 [46] 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

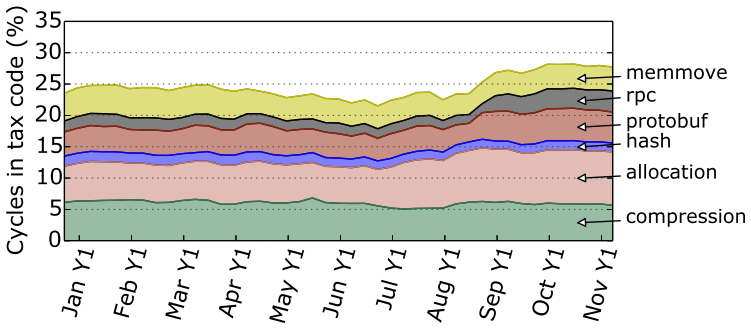


Figure 3.4: 22-27% of WSC cycles are spent in different components of "datacenter tax".

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 3.4.

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 [31, 118]. Like other data-intensive accelerators [66], 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 [35] 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 [108] could optimize away this piece of tax.

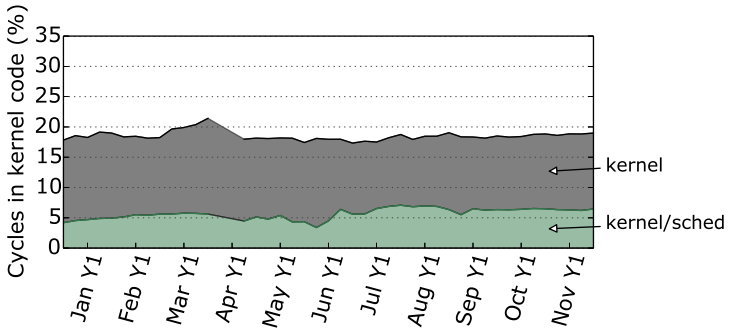


Figure 3.5: Kernel time, especially time spent in the scheduler, is a significant fraction of WSC cycles.

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 [72, 90].

Memory allocation Memory allocation and deallocation make up a substantial component of WSC computation (as seen by `allocation` in Figure 3.4), despite significant efforts in optimizing them in software [41, 69]. 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. Chapter 5 is one such investigation.

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 3.5). 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 3.5, which has to deal

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

with many diverse applications, each with even more concurrent threads (a 90-th percentile machine is running about 4500 threads concurrently [131]). Even after extensive tuning [119], the scheduler alone accounts for more than 5% of all datacenter cycles.

3.5 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 [125] 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 3.2.

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 3.6. 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`,

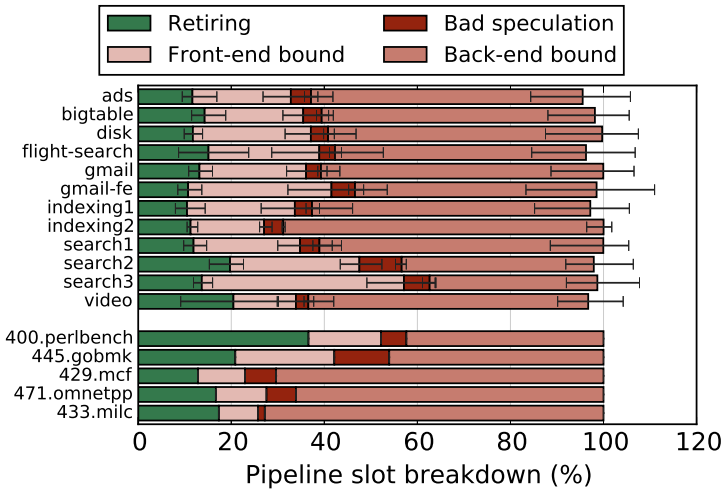


Figure 3.6: Top-level bottleneck breakdown. SPEC CPU2006 benchmarks do not exhibit the combination of low retirement rates and high front-end boundedness of WSC ones.

`471.omnetpp` – memory-bound, stressing memory latency; `433.milc` – memory-bound, stressing memory bandwidth.

The first immediate observation from Figure 3.6 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 3.7. `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.

3.6 Instruction cache bottlenecks

The Top-Down cycle breakdown shown in Figure 3.6 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

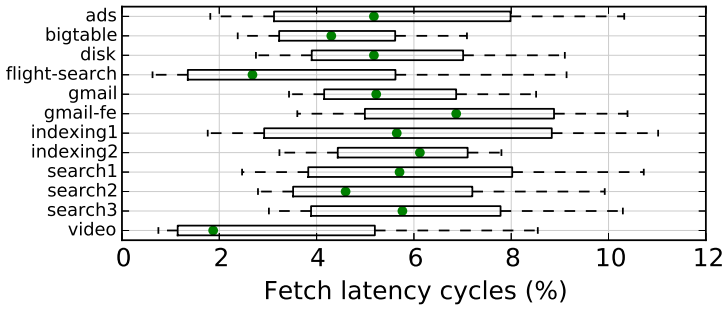


Figure 3.7: Cycles completely starved on front-end bottlenecks account for more than 5% of execution.

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 3.7 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.

Such a hypothesis is confirmed by the high exhibited L2 instruction miss rates from Figure 3.8. 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 [39].

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 3.3, 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 [5, 64]. 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

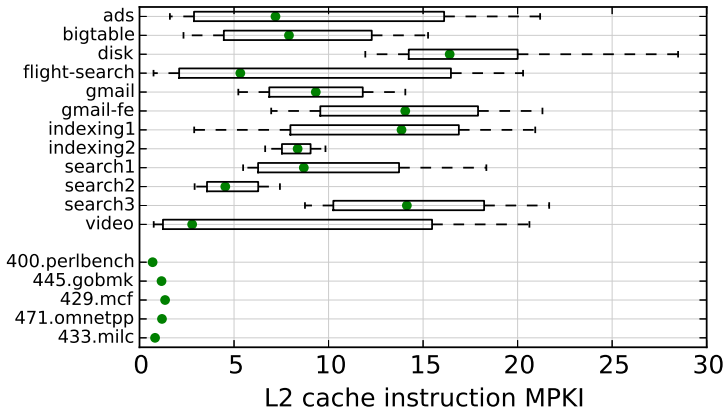


Figure 3.8: Instruction misses in the L2 cache are usually high.

(Qureshi and Patt [99], 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 [54], and the SPARC M7 design team opted for an architecture with completely separate L2 instruction and data caches [72].

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,

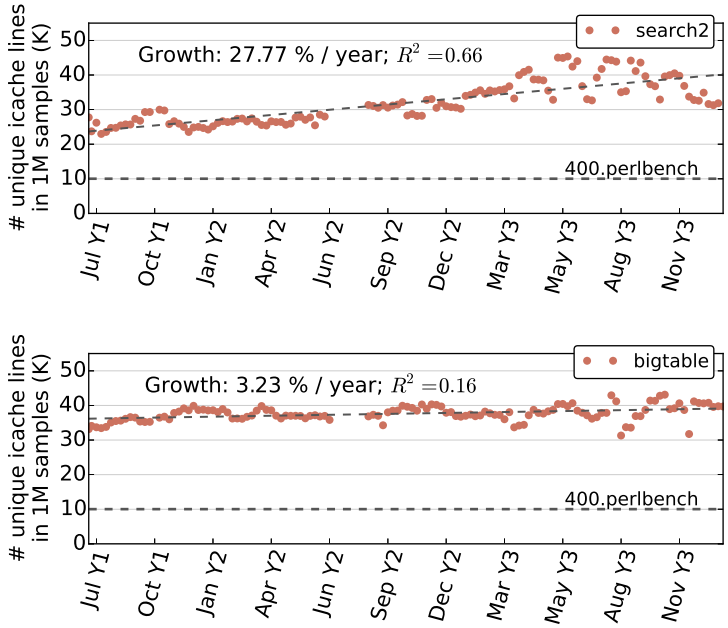


Figure 3.9: Large instruction cache footprints. Getting progressively larger for some applications.

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 3.9 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) [53].

First, compared to SPEC, all workloads demonstrated several fold larger i-cache working sets. Figure 3.9 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 alarming rates for some applications. Consider `search2` in Figure 3.9, 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.

3.7 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 3.6) 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 3.10) – 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 [55], and has led researchers to investigate the potential of small cores for warehouse-scale applications [3, 55, 76]. 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 3.11 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 3.6).

However, not all cycles are spent waiting on data caches. We demonstrate this in Figure 3.12, which measures the distribution of extracted ILP. By *extracted* ILP, we refer to the number 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%

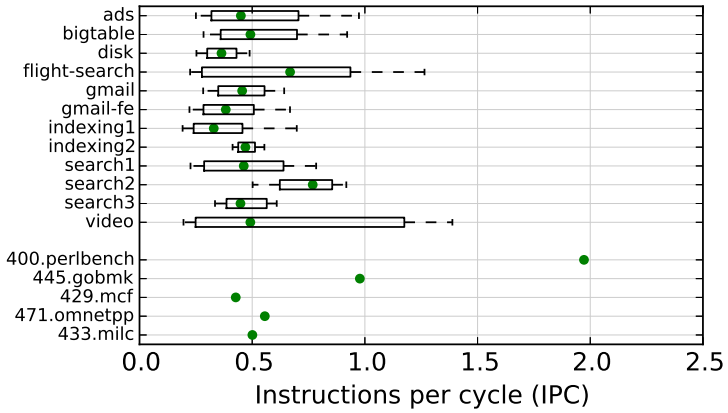


Figure 3.10: IPC is universally low.

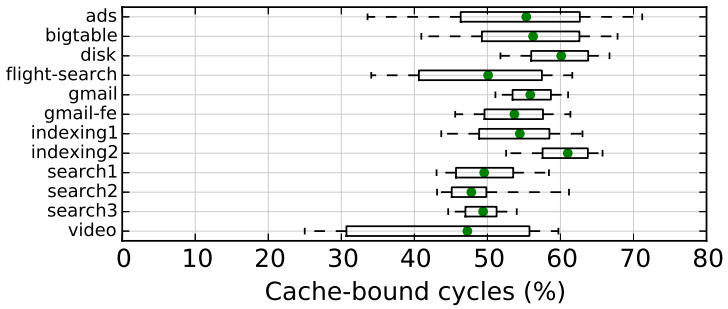


Figure 3.11: Half of cycles are spent stalled on caches.

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 3.13. The plot is a cumulative histogram of measured DRAM band-

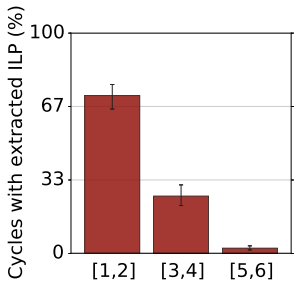


Figure 3.12: Extracted ILP. 28% of cycles utilize 3 or more execution ports on a 6-wide machine.

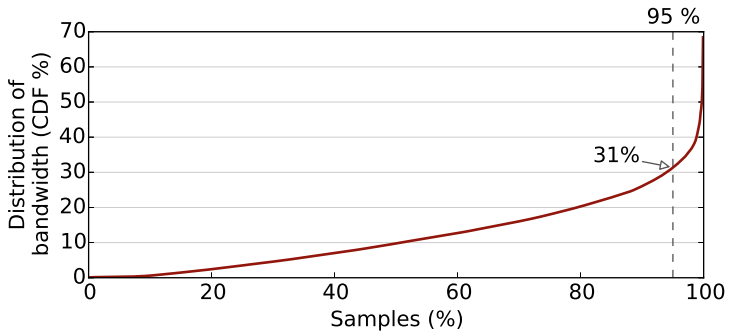


Figure 3.13: Memory bandwidth utilization is universally low.

width 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) [8], 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 [39] and other emerging datacenter workloads [82].

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

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

shift several benchmarks' memory bottlenecks from latency to bandwidth [125], causing architects to optimize for a less relevant target.

3.8 Simultaneous multi-threading

The microarchitectural results shown so far did not account for simultaneous multi-threading (SMT), even though it is enabled on the Ivy Bridge machines profiled. For example, the top-level cycle breakdown in Figure 3.6 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 3.14 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 3.12, 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 3.6). 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 3.14. Per-core `Front-end bound` cycles are significantly lower than when measured per-hyperthread – 16%

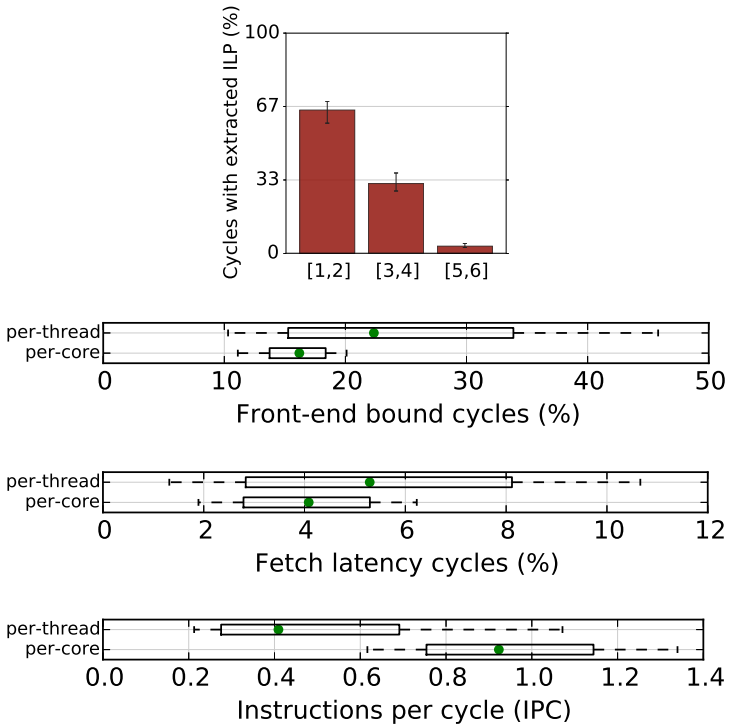


Figure 3.14: SMT effects on architectural behavior. From top to bottom: (i) more ILP extracted compared to Figure 3.12; (ii) front-end bound cycles decrease, but (iii) instruction starvation still exists; (iv) core throughput doubles with two hyperthreads.

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 3.14), 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 [72]. 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.

3.9 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 [3, 55, 76], and recent efforts have investigated specialized interconnects [79] and customized hardware accelerators [97]. 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 [39]. Recent efforts have provided in-depth microarchitectural characterization of portions of CloudSuite [126]. 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 [39]). Similarly, DCBench focuses in more depth on cloud data analytics [56]. 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 [65]. Another paper [9] 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 [57] and reducing [85, 131] contention between co-scheduled jobs, or of scheduling them in accordance with machine characteristics [84]. 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 stud-

ies [105]. We also leverage recent advances in Top-Down performance analysis [125] that allow us to estimate CPI stacks without specialized hardware support [38].

3.10 Conclusions

To better understand datacenter software performance properties, we profiled a warehouse-scale computer over a period of several years. 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.

4

XIOSim: a rich extensible user-level x86 simulator

While large-scale profiling studies allow us to get a broad picture of datacenter workloads, specific hardware changes stil need to be evaluated on a case-by-case basis. Because of the extremely high costs of chip design, virtually every architectural change is first prototyped in higher-level simulation models before silicon. These models are evaluated on three main dimensions – speed, accuracy and flexibility. Speed is both self-explanatory and easy to quantify. Usually, the more details that a model has, the slower it runs. Accuracy refers to the prediction error of the simulator versus the eventual chip it models. In practice, many new architecture proposals are incremental changes to an existing chip, so accuracy is evaluated in how well a baseline model matches with existing measured performance. Finally, flexibility refers to the (lack of) specificity of the performance model and its ability to predict performance of a wide range of designs. These three metrics are often in contention and architecture simulators tend to satisfy at most two.

We present XIOSim, a general-purpose x86 performance model that explicitly targets the accuracy and flexibility corners of this space. XIOSim is under a BSD license and is developed in the open at `http://xiosim.org`. XIOSim has very detailed cycle-by-cycle models of many architectural components that can be combined into many flavors of homogeneous and heterogeneous processors. It achieves simulation errors below 12%, for many workloads significantly so. It runs at 100-500 kilo-instructions per second (KIPS) for single-core simulations, and maintains similar per-core speeds up

to 8 simulated cores. In the rest of this chapter, we outline some of the main features of XIOSim that distinguish it from other publicly available models. We focus mostly on its co-simulation execution model and the efforts spent in validation, since validation has taken up the majority of development time. We finally showcase XIOSim with a case study of HELIX-RC – a co-designed system with a parallelizing compiler and a specialized core-to-core communication fabric – in order to demonstrate the types of studies enabled by a flexible cycle-by-cycle simulator.

4.1 Why another simulator?

The XIOSim project started in 2009, when the landscape of publicly available microarchitecture simulators was a lot different than at the time of writing of this document. SimpleScalar [15] had been the de facto model of choice for more than a decade. However, it lacked x86 support, and cross-compiling for the SimpleScalar instruction was forcing researchers to older and increasingly less supported toolchains. PTLSim [127] did support x86, with detailed core and uncore models. However, it was geared towards full system simulation, which is often too heavy and inflexible for CPU-intensive workloads that spend most of their execution time in user space. To fill the niche of user-level, highly detailed x86 models, Zesto [78] had just been released. However, support and further development of Zesto stopped almost immediately after the release.

The initial motivation behind XIOSim was to add aggressive in-order x86 core models to the existing rich out-of-order model in Zesto. With the rise of both mobile and datacenter computing in the late 2000s, simpler cores appeared to be making a comeback, especially when in large numbers, but there were no models for them. Available in-order simulators were targeted towards the extremely simplistic 1-way several-stage cores found in micro-controllers [30]. In contrast, XIOSim started as a model for the Intel Atom, an aggressive 2-way 15-stage in-order core, which shared many more characteristics with larger out-of-order x86 cores than with micro-controller-grade ones. Gradually, the focus in XIOSim development shifted away from just an Atom model towards a comprehensive framework that allows mixing many different types of x86 cores and uncores, with as rich and customizable component models as possible.

XIOSim maintains all models for microarchitectural structures described in the Zesto paper [78], though they are typically just some of the available options. In addition, there are new models for: Atom-like cores, 1-IPC cores, power consumption, cache controllers and coherency, cache structure re-implementations, load and store queue re-implementations, interconnects, vector units (SSE/AVX), synchronization operations (fences and atomics), a robust x86 decoder, a micro-op cracker, new micro-op fusion modes, macro-op fusion, a stack engine, move elimination, additional prefetchers, addi-

tional branch predictor components, voltage and clock domains, dynamic voltage and frequency scaling, among others.

In addition, XIOSim implemented a new co-simulation execution model with a user-space virtual machine acting as a functional simulator, which allows it to handle workloads significantly more complex than those handled by traditional user-level emulation-based simulators. A large number of features are enabled (or required) by this model. These include: a thread scheduler, multi-process simulation, a process core allocator (with several allocation algorithms), shadow virtual memory, host-speed fast-forwarding, along with several options for region-of-interest simulation (SimPoints, hook-based, instruction-count-based). The full implementation of this execution model is probably the most innovative feature of XIOSim, and we will discuss it in more detail in the following section.

Over several years of simulations, XIOSim has accrued several high-level APIs for typical simulation tasks. These are as simple to use as changing a configuration flag, and include: simulated time profiling, simulated sampling performance counters, instruction histograms, full microarchitectural event tracing, ignoring functions / specific instructions in simulated execution (for limit studies of performance gains), replacing functions / specific instructions either with fixed-latency "magic" instructions (for quickly evaluating the feasibility of acceleration), or with hand-encoded instruction sequences (for more detailed acceleration studies).

XIOSim's detailed and flexible models are very suitable for microarchitectural studies, where the cycle-by-cycle interactions between subcomponents are crucial. This is in contrast to newer simulators [20, 107] which raise the level of abstraction, achieving impressive gains in simulation speed, but sometimes at the cost of precision or flexibility in the types of components they can model. XIOSim keeps the level of detail of rich and heavy simulators, but its execution model allows rich simulation of workloads that were previously out of reach for them. In addition, a long list of performance optimizations have made sure that XIOSim performance is better than traditionally expected from rich simulation – typically around 5× slower than higher-level models like Sniper, but 10× faster than detailed models like Zesto.

4.2 Execution model

As many simulators before it, XIOSim implements a decoupled simulation model, with separate functional and timing simulators, responsible respectively for (i) the correct execution of the simulated programs, and (ii) their predicted performance on the simulated machine (this model is also know as *execute-at-fetch*). XIOSim extended this model by using a user-level virtual machine as the functional model. It is also the only simulator to do so in all cases, including speculative execution. The rest of

this section goes into detail about this execution model.

In early simulators, such as SimpleScalar and Zesto, the functional model was based on emulation, with an explicit representation of the simulated machine's architectural state in the simulator itself (especially registers, memory was usually at least partially shared with simulated processes because of the high cost of complete duplication). Such a simulator architecture makes many tasks relatively easy: for example, undoing mis-speculated instructions is done by simply running the emulator "in reverse" on the latest state representation. However, it puts enormous strain on the emulator. Because the program is effectively run on the emulated state, the emulator has to diligently and correctly support every single instruction in the instruction set simply for correctness. This is especially cumbersome for legacy instruction sets like x86, where whole classes of instructions (x87) and modes of operation (16-bit) are largely irrelevant for performance, but occur in initialization code just enough to corrupt the simulated machine's state, if not handled appropriately. In user-level simulators, this problem gets an additional dimension – the emulator can skip all system-level instructions (and many details like control registers), which simplifies implementation, but it is now responsible for emulating the effects of every single system call (and its parameters) on the simulated state, which erases many of the simplicity gains. Because of this pressure on emulation, classic simulators were often very picky about the workloads that can be simulated – they could often only target relatively simple benchmarks compiled with a specific compiler version against a specific system library on a specific distribution.

One of the very early observations in developing XIOSim was that emulation is in fact unnecessary – the simulation host itself is the best (correct by definition, already validated, faster than any emulator) functional model of an x86 machine. To that end, XIOSim uses the Pin binary instrumentation toolkit [81] to observe the effects of executing an instruction on the host and to pass them to a timing model. To the best of our knowledge, XIOSim is the first cycle-level simulator based on this premise (commit `a1f943` from 03/2010). It is inspired by PTLSim's co-simulation [127], which uses a modified version of the Xen hypervisor to transfer architectural state between the host and simulated (virtualized) machine. This has proven to be a popular model with other new simulators adopting it (McSimA+ [1], Sniper [20], ZSim [107]), independently of XIOSim.[1] Dealing away with emulators means that the simulator can now handle larger and significantly more complex workloads (e.g. databases, search engine nodes, workloads running under JITs, etc.) without any correctness issues due to esoteric instructions and their corner-case behavior. Of course, if such instructions are performance-relevant for a workload, they should be modelled accurately (which is why we focus heavily on XIOSim validation in the following sections).

[1] They were already under development when XIOSim implementation details were published and the code was open-sourced.

XIOSim takes this idea one step further – just as there is no need for emulation, performance estimation rarely needs the actual values in architectural state. So there is no need for an explicit representation of machine state for the majority of time. There are notable exceptions, of course, and these can be taken from the simulation host's architected state. Most importantly, memory operations need addresses for accurate simulation (which can be intercepted by Pin relatively cheaply). Similarly, some divider implementations can have variable latencies depending on the their operand values, but such cases are rare and can be treated separately. This significantly simplifies the implementation of a split simulator – the functional part is limited to a host state observer.

XIOSim implements this model by splitting the functional and timing simulators in two completely separate processes, which we call: instruction feeder and timing simulator. The instruction feeder uses Pin to instrument every executed instruction and to capture its current and next address, instruction bytes, and memory reference addresses, if any. These are passed to the timing simulator through interprocess queues, one per simulated core. In our implementation, the main parts of these queues are in-memory files on `tmpfs` (or on disk, if necessary). Both reads from (on the timing side) and writes to (on the feeder side) these queues are very frequent operations, so they are implemented in a lock-free manner, so they do not become bottlenecks. This multi-process split has several significant benefits. First, most obviously, it allows for multiprocess simulation (we launch multiple feeders and a harness process to orchestrate them). Second, keeping the feeder relatively light reduces the amount of interference that the simulator itself causes on the simulated application (for example, by polluting its address space). Finally, decoupling allows us to hide the costs of binary instrumentation and to effectively pipeline functional and timing simulation. This is at the expense of using extra cores on the simulation host.

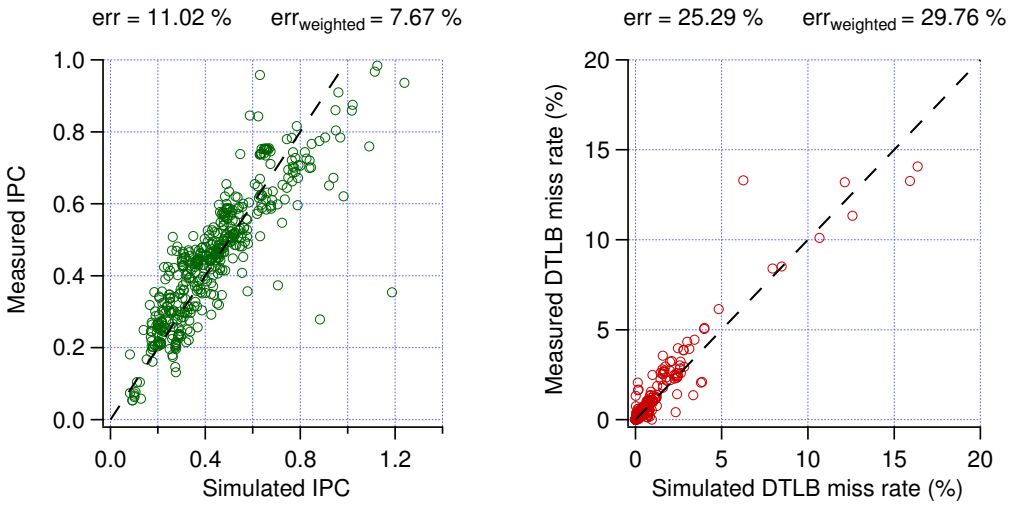
There are two major issues typically associated with instrumentation-based simulators. Both stem from the fact that the instrumentation engine can only see architectural machine state, and not implementation details. The first one is micro-op cracking, which most other Pin-based models also address. Modern x86 cores break down the potentially complex x86 instructions in simpler micro-ops, and most performance properties are based on micro-ops, not the full instructions. Any performance simulator with enough level of detail for microarchitecture studies needs a model of this cracking. The XIOSim micro-op cracker is somewhat non-traditional. The classic model is one of explicitly writing out micro-op tables by hand (the mapping from macro-op to micro-ops with their respective operands) and simply doing a lookup at cracking time. Instead, we use a macro-op-level instruction decoder (Intel XED [26]) to grab some properties of the macro-op and create the list of micro-ops based on heuristics on them. For example, a very popular pattern that we fall back to is load-op-store cracking, where a single macro-op with a read-write memory operand is broken down

into a load micro-op, the respective operation micro-op and a store micro-op. This shifts the burden of maintaining precise instruction tables to the XED developers (we do maintain an extensive unit test suite for our crackings, though) and allows for a simpler simulator implementation. The heuristics approach is also somewhat slower than a direct table lookup. So far, this has not been a significant performance problem in XIOSim, but we plan to hide the micro-op cracker behind a decode cache (as seen in some fast emulators [21]), should it become one.

The second, more fundamental issue, is modeling the effects of mis-speculated instructions. Instrumentation engines like Pin are oblivious to wrongly predicted execution paths, but the instructions on these paths can significantly impact end simulated performance. Other instrumentation-based models have very high-level models for wrong-path effects: either a constant cycle cost for branch mispredictions [107]; or a constant cost, but accounting for overlapping mispredictions [20]. Such models ignore many significant performance effects, such as wrong-path instructions disturbing instruction caches and prefetchers [40]. XIOSim is the only instrumentation-based simulator that we are aware of which models wrong-path instructions diligently and accurately, allowing one to capture such effects.

In XIOSim, each feeder thread has a branch predictor, identical to the one that will be executed in timing simulation. When that predictor makes an incorrect prediction, the feeder process is forked, and a speculative child is sent on the wrong execution path that was just predicted. The parent (correct-path) process then blocks until the child either: (i) produces enough instructions to fill a reorder buffer (the maximum speculation length) and exits cleanly, (ii) terminates because of a wrong-path exception (e.g. a segfault), or (iii) is about to cause side effects outside of its own process (e.g. write to a file or to a memory-mapped region). There are several important aspects for this model to be both correct and practical. First, limiting the side effects of the speculative child process is crucial. This includes finishing speculation on every system call, as well as capturing the `mmap` family of system calls in order to keep track of regions of virtual memory mapped to files. Second, it is important to have a fallback path, in case speculation stops because of side effects (which is an artefact of simulation, not something that would happen in real hardware) – in XIOSim, we send multi-byte NOPs to the timing model – which stress the core's front-end similarly to the instructions that would be executed. The performance of this scheme can be sufficient – instruction feeder latency can be hidden because of the buffering between itself and the timing simulator – and fork is relatively fast with copy-on-write memory. [2]

[2] One last performance bug for this scheme comes from the Pin VM sometimes capturing a segfault on a wrong path and taking more than 30ms for error reporting, which is enough to cause performance bottlenecks at very high branch misprediction rates.



(a) End performance – IPC

(b) DTLB miss rate

Figure 4.1: Macro-level validation of Intel Atom models over the SPEC CPU2006 suite.

4.3 Validation

Macro-benchmark validation Macro-level validation is useful for a single final number representing the "goodness" of a particular model and for catching accuracy regressions. However, it has limited value for improving the actual model – macro-benchmarks stress many of its components simultaneously – and can only guide the direction of improvement in very broad strokes.

Figure 4.1a shows macro-level validation for XIOSim's Intel Atom models against an Atom 330 for the SPEC CPU2006 suite. Each dot is a single 100M-instruction SimPoint [111] that is run both natively and in simulation. On average, XIOSim achieves 11% error. If we account for each points' weight in representing the full-length application, the error falls below 8%.

While macro-benchmark validation can guide the direction of model improvement – usually by comparing performance counters with simulation statistics – one has to be careful when interpreting the results. We illustrate this in Figure 4.1b. A cursory look at the very high levels of relative error in DTLB miss rates (>25%) would suggest spending significant development time to improve TLB model accuracy. However, upon closer inspection, one notices that for more than 80% of samples, TLB miss rates are in fact lower than 1%, and the hight relative error comes from comparing, say, 0.1% versus 0.05%. In this case, blindly chasing low relative per-component errors would hardly make a difference in overall performance accuracy.

Validating sampled execution Simulated execution is orders of magnitude slower than that on a native machine, so unless one is willing to tolerate multi-month-long validation experiments, some form of sampling must be employed when validating against macro-benchmarks. The two major general-purpose simulation sampling approaches – SimPoints and SMARTS – can have CPI errors reaching, respectively, 10% [94] and 8% [124]. Thus, when model errors reach 10% and lower, comparing sampled simulated execution with full-length native runs becomes impractical – there is no way to distinguish sampling error from model error. In an ideal validation experiment, the exact same instruction sequence should be compared between simulated and native execution.

Such an experiment is easy for workloads that have natural quanta of work with clearly defined boundaries – for example queries in database. For them, one can implement sampling on that quanta with minor source modifications, and either start/stop collecting performance counters or simulated execution at each sample [68]. That requires: (i) quanta of work; (ii) detailed knowledge of the workload in order to choose representative samples; and does not practically scale past 5 or 10 benchmarks.

For most XIOSim validation experiments, we choose a different approach. For the simulated runs, we use SimPoints as intended to choose representative simulation regions (in particular, the PinPoints implementation). For the native runs, we use lightweight binary instrumentation to approximate the start and stop markers for a particular SimPoint, and use these approximations to start or stop performance counter collection. For example, for natively-compiled SPEC workloads, we round SimPoints to the nearest function call and use hand-optimized lightweight instrumentation (through Pin probes; with <0.9% overhead) to match the number of calls to that function to the ones exercised by the SimPoint.[3] Similarly, in HELIX-RC experiments (described in the next section), we approximate SimPoints to the nearest loop boundary as identified by the ILDJIT compiler [18]. This methodology allows us to validate on virtually the same instruction sequence in both native and simulated execution, and not incur sampling error between the two.

Targeted validation: proof by optimization Probably the most common approach to validation is what Yasin calls "proof by optimization" [126]. A simulator has to make many assumptions about the implementation details of low-level components, and despite the large number of performance counters, the effects of these assumptions usually cannot be measured directly. In this case, the "optimization proof" that a new assumption fixes an accuracy regression is done in a few steps: (i) isolate the accuracy issue with a microbenchmark as targeted as possible, so that the discrepancy

[3]We extend this methodology to power trace validation in our XIOSim paper [60], where the instrumentation also inserts instruction sequences that cause a specific power consumption pattern picked up by an oscilloscope. Later, when post-processing the oscilloscope power trace, we look for these power markers to sync up simulated and measured traces.

shows up in a high-level measurable metric (cycles, branch/cache MPKI, etc.); (ii) implement a new version of the relevant component based on a new assumption (the "optimization" part); (iii) measure whether the high-level metric now matches up (the "proof" part); (iv) if not, repeat with an updated assumption.

The importance of the first step – a reproduction test that is targeted enough – cannot be overstated. Simulators can be thought of as performance models with thousands of dimensions, with various workload features serving as an input to the model. The interactions between different dimensions (especially when modelling an out-of-order CPU) are usually non-trivial – it is very easy to both mask a performance bottleneck with another one in an unrelated portion of the machine (e.g. high i-cache miss rates hiding an under-provisioned decode unit) and to exacerbate one bottleneck with another (e.g. high branch prediction miss rates leading to d-cache bandwidth saturation because of many speculative accesses). If a benchmark stresses many of these correlated dimensions, it is very easy to end up optimizing the wrong bottleneck, and still see an accuracy improvement in the high-level metric that is used to judge model goodness.

Example: One accuracy experiment from the early XIOSim models illustrates this approach well. We were investigating a substantial amount of inaccuracy in one of the SPEC CPU2006 workloads. Several performance counters were showing large differences with native execution – branch misprediction rates, i-cache miss rates, d-cache miss rates, among others. We started by following up on branch mispredictions. On the native execution, there were only several performance counters related to branches, so that could not drive the investigation to any specific part of the branch predictor. On simulated execution, we saw an unusually high fraction of return address stack (RAS) predictor misses. Our RAS predictor model at the time was a simple stack, with call instructions pushing their return addresses and return instructions popping them (as well as getting their next address predicted).

One of the (few) failure modes for a structure as simple as the RAS predictor is many speculative call / return pairs that end up being mismatched and corrupting the stack. The classic mechanism to account for that is checkpointing RAS state on a call, and reverting to the checkpoint on a related branch misprediction, which is what XIOSim modelled. We wrote a microbenchmark to stress this failure mode – stressing a call / return pair is easy with recursion, but we needed a bit more complexity, so that we have enough speculative execution, as well as multiple return sites to confuse the RAS. These can be all satisfied by a multiply recursive function, like a naive Fibonacci number implementation. This was a correct hypothesis, and the microbenchmark also had high RAS miss rates, as well as a large discrepancy in branch prediction rates with native execution.

The microbenchmark also suggested a possible solution – limit the amount the damage that spec-

ulative execution can cause to the RAS stack. We implemented a RAS with two stacks[4] – real and speculative – where pushes / pops to the real stack happen at commit time only by non-speculative calls and returns. Critically, returns before that (which can still be mispredicted) can only pop from the speculative stack, but can predict from the real stack. After implementing this multi-stack RAS, our microbenchmark's simulated performance very closely matched native hardware, strongly suggesting that Intel machines use a similar RAS predictor implementation. Similarly, the amount of branch predictor misses in the macro-benchmark that triggered this investigation was within an acceptable error margin, and the error on other counters also decreased noticeably – suggesting that RAS behavior was indeed one of the related bottlenecks in this instance.

Last resort: tracing If all other approaches fail, one can resort to tracing in order to find performance inaccuracies. In debug mode, XIOSim maintains a full trace of microarchitectural events that gives full visibility of the simulated machine. The trace can easily log 30-50 events per instruction, and, at simulation speeds of 100s of KIPS, can grow at 10s of MBs per second. This is why XIOSim keeps it in a in-memory circular buffer and only flushes it on assertion failures or at the end of simulation. Identifying the root cause of a problem in that trace is often quite laborious.

Example: This approach is best illustrated with a recent example from validating XIOSim on memory allocation workloads (described in detail in Chapter 5). A particular memory allocation microbenchmark exhibited random performance regressions in about 10% of simulations without any apparent change in conditions or simulation parameters. The microbenchmark was designed to stress best-case allocation, with very good cache behavior, so it achieved IPC of 3.0 on a 4-wide simulated Haswell core, more than most workloads we had ever simulated. In the 10% of faulty runs, simulated IPC dropped drastically down to 2.2. Typically, such large changes in performance between runs indicate problems in the cache hierarchy – dynamically-allocated addresses do change run-to-run – however, in this case, all cache statistics were the same between the faulty and accurate runs, and all showed the perfect caching behavior that was expected.

After several failed attempts to reproduce the issue with more targeted microbenchmarks, tracing was the last viable option. After comparing traces, we found many DTLB accesses not making it to the TLB and getting the corresponding load micro-ops rescheduled. The high IPC caused TLB bandwidth (provisioned at 3 for a Haswell machine) to become a bottleneck. Interestingly, we were already modelling a TLB with sufficient theoretical bandwidth. However, we assumed it was implemented as a simple interleaved banked cache, where a hash function sends requests to the appropriate bank. The faulty runs were due to dynamically allocated pages hashing to the same bank and reducing effective

[4] Conceptually, a hardware implementation can use a single array with two top-of-stack pointers

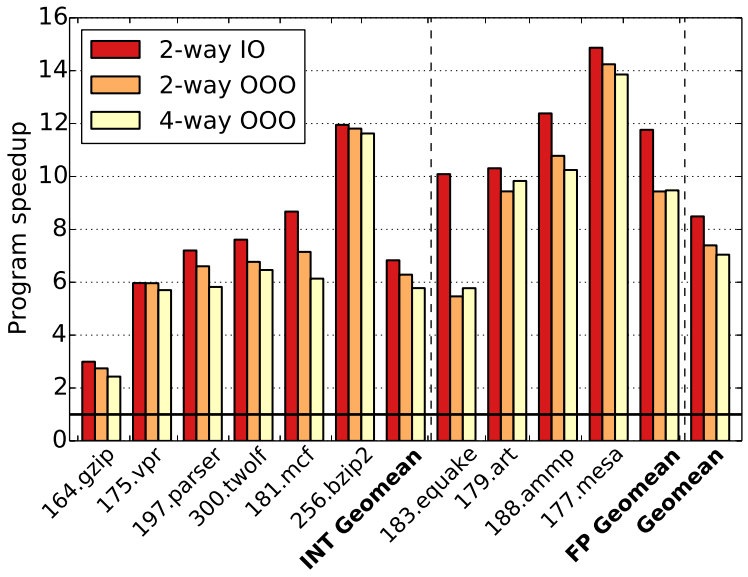


Figure 4.2: HELIX-RC speedup relative to each core type's baseline.

bandwidth. We changed our models to assume a more brute-force multi-ported cache implementation, but are also planning to implement more advanced TLB bandwidth optimizations [6], which can result in similar bandwidth without the large area cost of a multi-ported cache. We also added additional simulation statistics, so similar problems can be identified without resorting to tracing.

4.4 Case study: HELIX-RC

HELIX-RC [19] is a co-design between a parallelizing compiler and a small addition to a processor's cache hierarchy, *ring cache*, which enables the compiler to parallelize irregular programs with unpredictable control flow and many data dependences. Ring cache was modelled in XIOSim and the detailed cycle-by-cycle models proved invaluable in evaluating its various design choices. Moreover, HELIX-compiled code was complex enough, and included simulating a JIT compiler, which puts it beyond the workloads easily handled by traditional cycle-level simulators. While HELIX-RC is a large collaborative effort and all its details are beyond the scope of this dissertation, we will outline the requirements such a project poses on simulation infrastructure.

One of the main aspects of the proposed system is extreme latency sensitivity. The loops parallelized by the HELIX-RC compiler have extremely short iterations (the 50th percentile of loop iteration durations is only 25 cycles). However, in parallel execution, they also need to communicate to

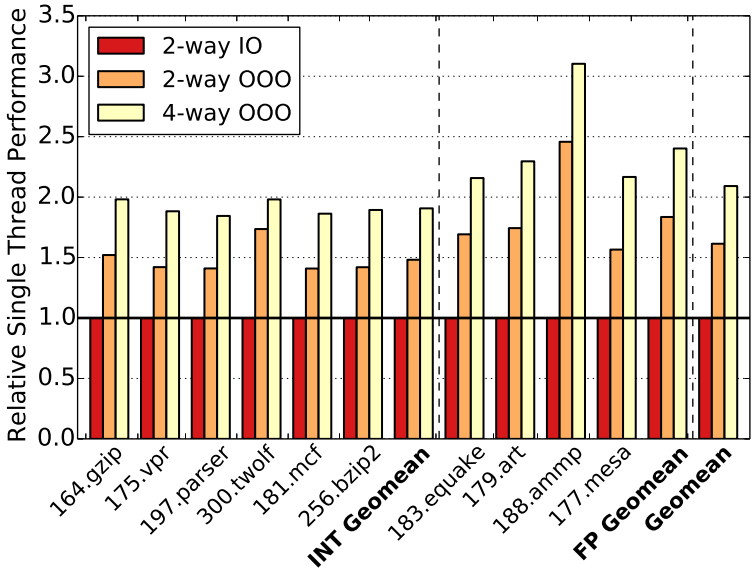


Figure 4.3: Baseline execution on various core types.

satisfy inter-iteration dependencies. With a conventional coherent cache hierarchy, the latency to send a single cache line to another core – at least 75 cycles on modern processors – is large enough to erase the gains from parallelizing these loops. Our sensitivity analysis has shown that even a very aggressive latency of 10 cycles is too high for meaningful speedup. Our proposed hardware addition, ring cache, cuts down core-to-core communication latency to effectively 0 by proactively sending data before it is needed and by relying on various compiler guarantees. This extreme latency sensitivity places significant importance on diligently modeling every single CPU cycle in simulation.

Our initial prototype assumed parallelizing for in-order, Atom-like cores, and achieved speedup of 6.85× on SPECint CPU2000 against these cores' sequential execution (Figure 4.2, 2-way IO). However, one interesting research question[5] was that integrating ring cache with a complex out-of-order core would erase the gains from parallelization. The rationale behind this hypothesis is that the HELIX compiler extracts very fine-grained thread-level parallelism (TLP), fine enough that it can compete with the instruction-level parallelism (ILP) that an out-of-order core is designed to extract.

Initial results from simulating aggressive out-of-order cores equipped with ring cache seemed to suggest this was the case – with no speedup. After investigating the cause of slowdown, we found it to be in the integration between the core's load-store queues and ring cache, and adjusted our designs accordingly. In more detail, one of the synchronization primitives we propose, `wait`, has semantics

[5] ... and a common cause of concern for paper reviewers

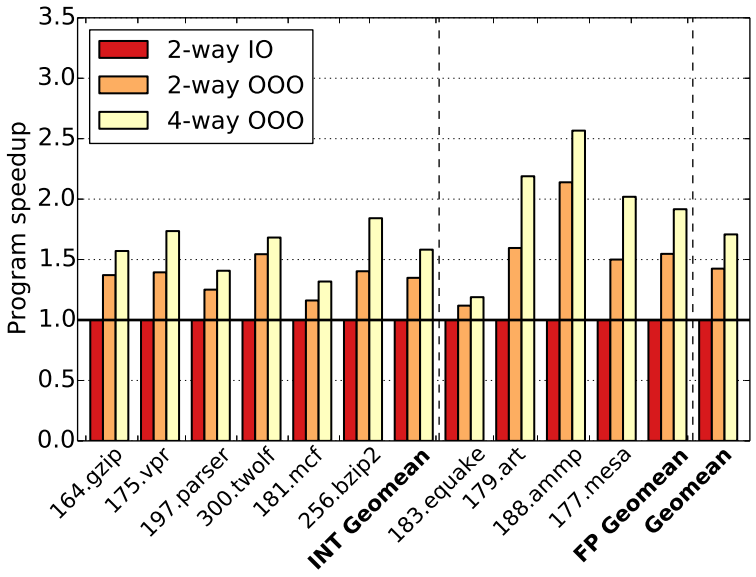


Figure 4.4: Overall parallel performance, normalized to simplest cores.

intuitively similar to a regular load, and some of our first models treated it like one – issuing it from the core's load queue, and surrounding it with lightweight fences to ensure ordering of `wait`s and that other regular loads are not reordered with respect to the `wait`. These fences also affect unrelated loads and greatly reduce the amount of memory-level parallelism of surrounding code, eventually resulting in slowdown.

Our later prototype treated `wait`s similarly to stores, and issued them from the naturally-ordered store queue. This significantly reduces both design complexity and the number of fences required. Speedup levels in this case are comparable with those of in-order cores (where none of these concerns are relevant), albeit still lower – as seen in Figure 4.2. Our experience with this particular design choice highlights the importance of cycle-by-cycle simulation when evaluating hardware-software co-design systems like HELIX-RC. Many higher-lever models would have likely missed the subtle differences in ordering and wrongly concluded that ILP is indeed very strongly cannibalizing TLP.

Our simulation results show this effect to a much smaller degree. Furthermore, while speedup indeed gets lower with more aggressive core implementations, each core's baseline (single-threaded execution) is also significantly faster (Figure 4.3), which results in higher overall performance for more aggressive cores – Figure 4.4. This higher performance certainly comes at high power cost, and one can imagine a heterogeneous system where the power-aware parallelizing compiler manages the parallel performance versus power tradeoffs. While we do not have validated power models for large out-of-

order cores yet to perform this study, it highlights the type of work that can be done with extensive and accurate simulation infrastructure.

Finally, after the initial HELIX-RC simulation study, a hardware implementation of ring cache in Verilog matched the performance predicted by XIOSim perfectly cycle-for-cycle [14].

5

Accelerating memory allocation

5.1 The need for broad acceleration

In the long term, the confluence of technology trends points steadily towards hardware specialization. Continued transistor density increases, coupled with the end of Dennard scaling, result in the inability to power a whole chip at maximum performance – the problem known as dark silicon. Hardware specialization has been widely adopted in processors to solve this problem.

Much existing effort in hardware specialization has focused on "deep" acceleration following the classic Amdahl's 90/10 rule. This involves identifying "killer applications" and optimizing their most costly kernels, be it ranking in websearch [97], convolutions in image processing [98], or matrix-vector products in neural network inference [103]. This strategy has seen especially great traction in mobile systems-on-chip, where the majority of silicon area in current designs is dedicated to specialized blocks [110]. However, the server chips powering cloud workloads remain predominantly general-purpose.

A major reason for this omission is that modern datacenter workloads are simply too diverse, without any opportunities for 90% optimization. Not only do they run thousands of different applications, but the individual workloads themselves have also been shown to not have significant hotspots that can be optimized with deep approaches [58]. This does not mean hardware acceleration in datacenters is infeasible. Characterization studies show that a large fraction of cycles is spent on the so-

called "datacenter tax" – low-level routines like remote procedure calls, data serialization and memory allocation. While each individual component of this tax is a relatively mild hotspot (in the 2-8% range), together they can comprise up to 30% of all cycles in Google datacenters [58].

The ubiquity of the datacenter tax suggests an alternative "broad" approach to acceleration: speeding up multiple shared low-level routines that appear in many applications. This approach may not provide the 10× application speedups typically associated with hardware specialization. But accumulating several instances of such several-percent optimizations can save significant amounts of CPU cycles, especially when deployed broadly across the hundreds of thousands of servers that cloud providers operate. Borkar refers to this approach as "10 × 10 optimization" [13] and argues that it is a necessity for continued performance increases in the era of dark silicon.

Of the components that comprise the datacenter tax, perhaps the most familiar one is `malloc`: dynamic memory allocation. `malloc` is such a popular programming paradigm that many collective developer-years have been spent researching and optimizing allocation strategies and techniques. For example, a typical `malloc` call takes only 20 CPU cycles on a current-generation general-purpose processor, setting the bar high for potential hardware implementations. `malloc` exemplifies the unique set of challenges facing broad acceleration: because calls to these routines tend to be very frequent, fast, and interspersed inside other application code, accelerators must be optimized for latency rather than throughput, and because each such accelerator brings a limited amount of overall application speedup, overheads must be kept to a bare minimum.

In this work, we present the design of *Mallacc*, a memory allocation accelerator that meets these constraints. Mallacc is a tiny in-core hardware block which accelerates the three primary operations of a typical memory allocation request: size class computation, retrieval of a free memory block, and sampling of memory usage. Mallacc is designed not for a specific allocator implementation, but for use by a number of high-performance memory allocators commonly found in datacenters today. Our goal is to make the already fast (20-30 cycle) `malloc` calls even faster, because they are so frequent, and Mallacc achieves that goal. It can reduce `malloc` latency by up to 50% while occupying less than 1500 μm^2 of silicon area. As we will show, Mallacc far exceeds the "1% speedup for 1% area" mantra that has informally guided processor development over the past decades.

5.2 Dynamic memory allocation trends

Dynamic memory allocation has been studied for decades. In this section, we place our work in the context of past literature. We discuss historical research on allocators, general techniques and structures that are still used in modern allocators, and factors that drove evolution of allocators over the

decades.

At a very high level, a dynamic memory allocator sits between an application and the operating system (often as a part of the platform's standard library). It requests continuous blocks of memory from the OS and distributes chunks of them, with different sizes, to call sites in the application that explicitly request them. Allocators are judged on both the speed with which they satisfy a request and their memory fragmentation, which measures how much memory is requested from the OS vs. how much memory the application actually uses.

In the very early days, main memory was expensive and scarce, so allocator design focused on minimizing memory fragmentation and overhead. Starting from the 1960s, researchers studied data structures for storing free objects, notably linked lists [29] and trees [115]. Various strategies for searching through free lists of memory blocks to identify the right object to return were examined: such as returning the first block large enough ("first fit"), the exact size ("best fit"), and many more [28]. Techniques for efficiently splitting and coalescing free memory objects were also studied; one notable example is the buddy system, in which a free object can split into two "buddy" objects for small allocations, but can only be merged with that same "buddy" when a large allocation is needed [63]. The notion of *size classes* – allocating memory from a set of specific sizes – was also conceived decades ago [117].[1] Many of these techniques and data structures are still used in today's allocators.

Over time, two trends motivated significant changes in allocator design. First, main memory costs dropped and densities increased exponentially thanks to Moore's Law. However, unlike CPU speeds, main memory access latencies stagnated. The increasing gap between CPU and memory speeds shifted the focus from memory fragmentation to speed. Second, the rise of multi-core processors and multi-threaded applications in the last decade motivated allocator designs that were fast and efficient in the face of problems like lock contention, false cache sharing, and memory blowup with large numbers of threads. Modern allocators like Google's `tcmalloc` [42], FreeBSD's `jemalloc` [36], Hoard [10], and others were all designed to support robust multithreaded performance.[2]

Modern multithreaded allocators like the ones listed above all share a common set of design principles. First, they logically organize available memory in a hierarchical fashion. The top level is a pool of memory that can only be accessed by a limited number of threads (often just one) to mitigate the cost of synchronization. These pools are highly optimized in software such that a hit in one is likely to be considered "fast enough". They are backed by lower-level pools, which are shared among threads. Memory is migrated back and forth as necessary. Second, they select a set of size classes and round

[1] Research in allocators has been especially prolific – for a significantly more complete survey of early approaches, refer to Wilson et al. [121].

[2] Similarly, Ferreira et al. [41] provide a succinct overview of the structure of modern allocators.

requested sizes to the next nearest size class, which simplifies splitting and coalescing of larger memory blocks and reduces the amount of metadata needed. Third, they use different methods to allocate "small" and "large" chunks of memory (though they differ on the exact thresholds of considering a chunk small). Finally, they ensure that memory can migrate from thread to thread to avoid memory blowup in scenarios where one thread allocates memory and another thread frees memory.

Within this framework of common design principles, modern allocators can differ significantly in their implementations. For example, size classes are selected based on different upper bounds of memory fragmentation. Heuristics for determining when to migrate memory from lower to upper levels, as well as how many blocks to move, vary greatly too. Lower level pools tend to store larger blocks of memory that are then sliced into smaller chunks for top level pools, which is a time-consuming process that requires synchronization. Similarly, at some point additional memory must be requested from the operating system, which requires a costly system call. Developers must balance the frequency of these requests with the overall memory usage and consider various allocation patterns from different applications. Therefore, the parameters of these procedures tend to change relatively frequently as developers seek out new optimizations and tradeoffs.

Compared to the broad popularity of software allocator research, creating custom hardware for allocators has received next to no attention. We are only aware of one feasibility study [75] and several variations of the buddy technique [17, 24, 25, 74], which show that it easily maps to purely combinational logic. While buddy allocation has been available for decades, modern allocators have converged to simpler techniques in their highest-level pools (most frequently, first-fit free lists), most likely due to buddy systems' reported high degrees of fragmentation [121] and relative complexity.

This presents an opportunity for hardware designers looking to accelerate allocation. Rather than design a whole new algorithm from scratch to simplify hardware implementation, they can speed up the common elements of modern allocators – the "fast enough" top-level pools – and allow different allocator algorithms to tune the details on the lower levels in software for their own workload assumptions. In the rest of the chapter, we demonstrate the feasibility of this approach by optimizing the top-level pools of TCMalloc [42]. While TCMalloc makes for a good anchor point to demonstrate gains – it is mature, robust and among the faster allocators [41] – the optimizations we propose can easily be used by other modern allocators.

5.3 Understanding TCMalloc

We start by describing how TCMalloc allocates and deallocates memory and compare and contrast it with other multithreaded allocators. We profile the costs of several allocator code paths and find that

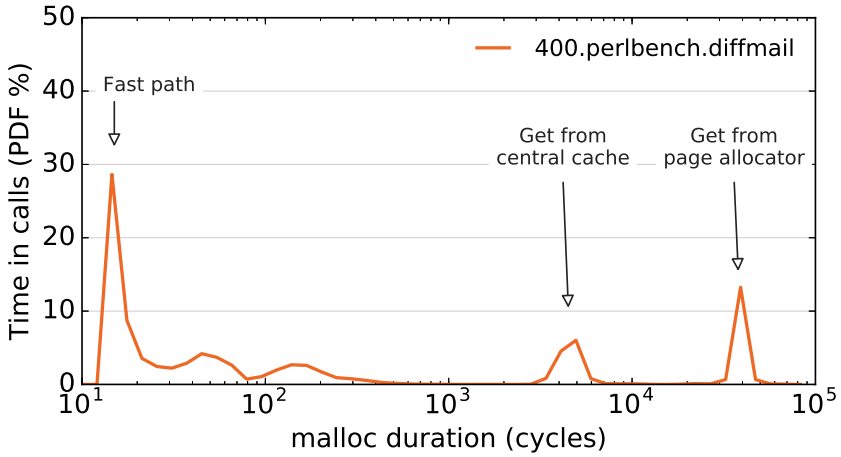


Figure 5.1: The costs of hits and misses in several allocation pools in TCMalloc vary by orders of magnitude.

the **fast path** is an overlooked area for potential optimization.

5.3.1 TCMalloc overview

Allocation pools Like many other allocators, TCMalloc allocates memory from a hierarchy of memory pools. At the top are *thread caches* assigned to each thread of a process, and meant to service small requests (< 256KB). Each cache contains many singly-linked *free lists* – lists with addresses to free chunks of memory of the same size. There is one free list per size class. TCMalloc currently has 88 size classes, a relatively large number picked to keep memory fragmentation low. When a free list is not empty, a small allocation can be satisfied by simply popping the head off the list. Since these caches are thread-local, no locks need to be acquired and a thread-cache hit is relatively fast. `jemalloc`'s thread caches were inspired by TCMalloc [37], and their size class organization is quite similar.

If a free list is empty, the allocator must first fetch blocks into a thread cache from a next-level pool. In TCMalloc, it either attempts to "steal" some memory from neighboring thread caches, or gets it from a *central free list*. Both approaches require locking, and are orders of magnitude slower than hitting in a thread cache. Should both of these sources be empty themselves, TCMalloc allocates a *span* (a contiguous run of pages) from a page allocator, breaks up the span into appropriately sized chunks, and places these chunks into the central free list and the thread-local cache. Large requests (> 256KB) go directly to spans and bypass the prior caches. Should the page allocator also be out of memory, TCMalloc then requests additional pages of memory from the operating system.

Figure 5.1 illustrates the cycles costs associated with hitting or missing in several of these pools for

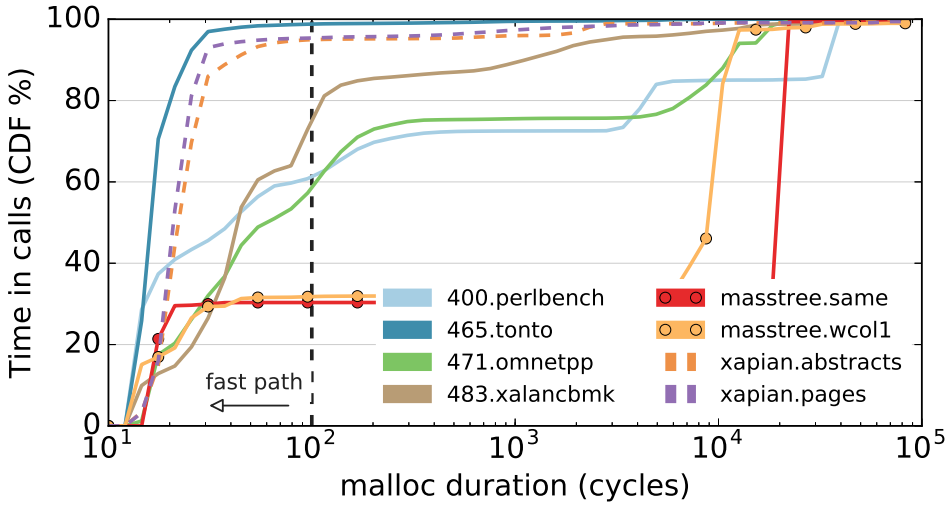


Figure 5.2: Majority of time in `malloc` in CPU2006 is spent on calls taking less than 100 clock cycles.

`400.perlbench` from SPEC CPU2006. It is a simulated distribution (details on our methodology follow in Section 5.5) of time spent in each `malloc()` call over the call's duration in cycles. The three major peaks correspond to hitting in a thread cache, missing in a thread cache and hitting in the central free list, and grabbing a span. Missing in a thread cache has a cost at least three orders of magnitude higher than that of a hit. Because of the high costs, too many misses in the highest-level pool can be detrimental to allocator (and application) performance. TCMalloc employs several heuristics to transfer chunks of memory between the various pools in an effort to maximize thread cache hit rates. These heuristics (and the particular implementation details of the lower-level pools) are what distinguishes different modern allocators from one another. Note, however, that despite their very low per-call cost, thread cache hits represent a significant chunk of allocator cycles overall for `400.perlbench`. We will come back to this observation in the following section.

Memory deallocation Deallocation follows a similar path. When memory is being freed, TCMalloc first determines the size class of the freed object. If that object is small, it gets pushed to the top of the appropriate thread cache free list, and if that free list now exceeds a certain size (2MB), TCMalloc returns unused objects back to the central free lists. If the freed object is large, pages of memory get returned back to the page allocator.

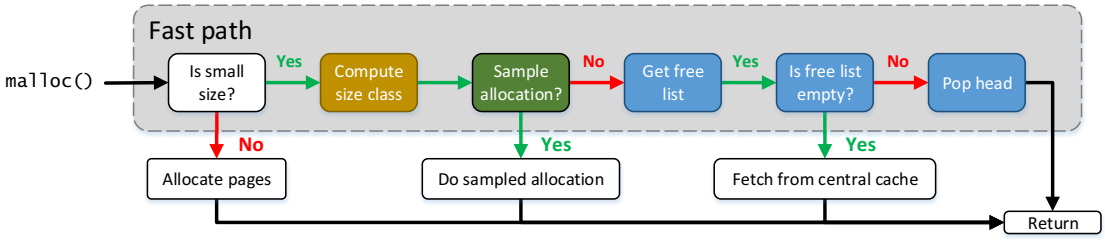


Figure 5.3: The steps that an allocation requests goes through. The colored boxes represent the major operations on the fast path, which we aim to optimize.

5.3.2 Time spent in the allocator

As discussed in the prior sections, research in allocator design has focused on the lower-level memory pools because of their potentially catastrophic effects on performance. This is also partially because the *fast paths* – those that hit in thread caches – are already considered sufficiently optimized. Microbenchmark experiments often support such a hypothesis. For example, our `tp_small` microbenchmark (described later) achieves an average `malloc()` latency of only 18 cycles.

However, we find that for a range of applications, time spent on the fast path is not only a significant, but also a major fraction of time spent in the memory allocator. Figure 5.2 shows this property for the four SPEC CPU2006 benchmarks that actually call the system allocator. In the cumulative distribution of `malloc()` time, more than 60% of time is spent on calls that take less than 100 cycles. For `xapian`, an open source search engine, we see an even higher fraction. This need not be the case for all workloads: for example, the performance tests of `masstree`, a key-value store, never free any memory and end up continuously getting more from the page allocator (which eventually goes to the operating system). A real deployment of `masstree` does free memory and has much better thread-cache hit rates, but even such corner-case behavior spends more than 30% of allocator time on the fast path.

There are two main reasons for the high fraction of fast-path time that we observed. First, while individually cheap, fast-path calls can be very frequent – a classic "death by a thousand cuts" scenario. This is especially true for applications that allocate and deallocate at similar rates so that their requests almost never have to reach the other memory pool levels. Second, thread caches are very cheap *in microbenchmarks*, but can get significantly more expensive when the requesting application itself is cache-heavy. In that case, the application's memory accesses evict the allocator's data structures from the CPU's caches, and a cheap 18-cycle fast-path call can turn into a hefty 100-cycle stall on main memory.

Thus, we believe the fast path of memory allocators presents an overlooked opportunity for opti-

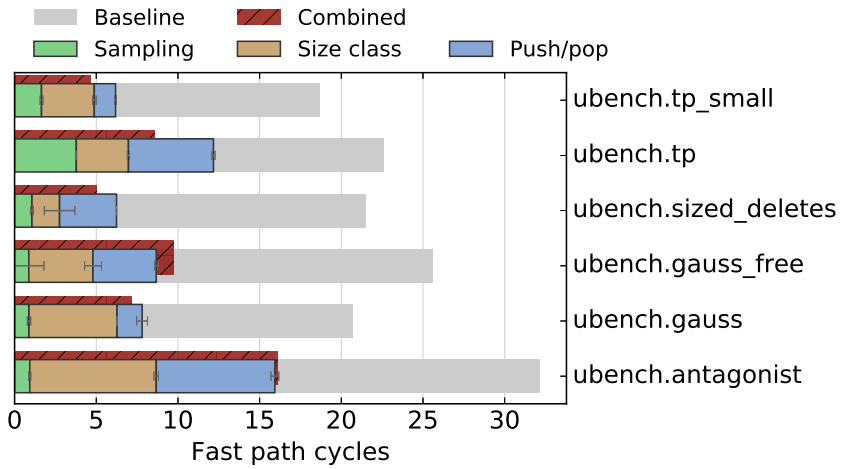


Figure 5.4: Time spent in the three main components of the fast path accounts for $\approx 50\%$ of cycles.

mization and focus the rest of the chapter on speeding it up with specialized hardware. For that, we need a detailed understanding of the work done during fast path calls, and the costs associated with it.

5.3.3 Analysis of the fast path

By definition, the fast path is a memory request satisfied by a thread cache free list. And by design, an access on the fast path has little work to do. For TCMalloc compiled with GCC 6.1, the fast path is only $\approx$ 40 static x86 instructions, and can take 18-20 cycles, assuming cache hits. It contains a few conditional branches that are easy to predict and no loops. Microbenchmarks with back-to-back allocations and deallocations can achieve an IPC of 3.0 on a 4-wide Intel Haswell core. In other words, it has been heavily optimized. Thus, speeding it up further is an exercise in performance microscopy and in reducing the latencies of the different steps of a fast allocation. Figure 5.3 illustrates these steps in the context of an incoming allocation request: 1) finding the appropriate size class for the requested size, 2) potentially sampling the request, and 3) satisfying it by popping the head of the corresponding free list. In the rest of the section, we go into more detail about the computation in each step and its cycle costs.

Note that we can (and do) estimate these costs, even if they are caused by only several instructions, because we rely on simulation. This is how we construct Figure 5.4, which contains cycle costs for several microbenchmarks designed to stress different fast path aspects. With simulation, we can simply remove these instructions from simulated execution and subtract the resulting cycle count from a baseline. These are estimates, and not strictly additive, since out-of-order cores explicitly overlap work.

```
size_t SizeClass(size_t size) {
  size_t class_index;
  if (size <= 1024)
      class_index = (size + 7) >> 3;
  else
      class_index = (size + 15487) >> 7;
  return size_class_table[class_index];
}

size_t class = SizeClass(requested size);
size_t alloc_size = size_table[class];
```

Figure 5.5: Size class lookup function.

When all removed together (the `Combined` bars in Figure 5.4), they make up for half the cycles of the fast path.

Size class calculation This operation rounds the requested allocation size to the nearest size class supported by the allocator. Because small size classes are more commonly observed, the spacing between small size classes is closer, and this spacing grows with the size class. Size classes are carefully tuned to balance fragmentation and allocator latency, so typically the mapping from size to size class does not have an easy closed form. In TCMalloc, this is implemented by first computing a *size class index* from the requested size and then indexing into two precomputed arrays, precomputed at initialization time for the size class and rounded size that it represents (Figure 5.5). The class index only requires an add and a shift, but the two array lookups can be comparatively costly, even if they hit in L1 because they are on the critical path of execution. The number of class indices (the size of the first array) is set by the threshold for a small allocation and by memory alignment requirements. This number was fixed at slightly above 2100 in 2007 when TCMalloc was open-sourced and has not changed. The second array is much smaller, currently at 88 (the number of size classes), and has seen two small increases since 2007.

Despite having 88 size classes available, we find that applications often use a relatively small subset. Figure 5.6 shows that, for the benchmarks we surveyed, all but one use less than 5 size classes on 90% of `malloc` calls. In fact, `masstree` almost exclusively uses a single size class.[3] `xalancbmk` has a much broader distribution, but even so, it uses one size class over half of the time. This observation motivates techniques to memorize the most common size classes.

[3]For allocations below 256KB only, which are handled by the fast path.

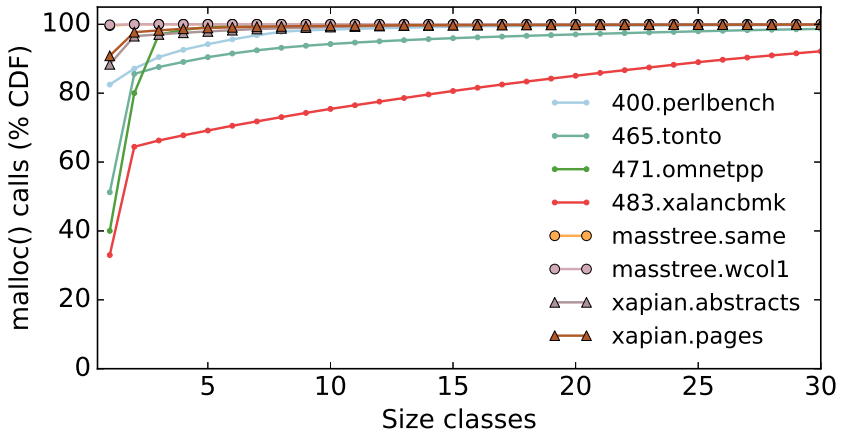


Figure 5.6: Many benchmarks use a very small number of size classes.

```
pop:
  load temp, MEM[head]          ; Get the head.
  load next_head, MEM[temp]     ; Get head->next.
  store MEM[head], next_head    ; head = head->next.
  return temp

push:
  load temp, MEM[head]          ; Get the head.
  store MEM[head], new_head     ; Set new_head as head.
  store MEM[new_head], temp     ; new_head->next = temp.
```

Figure 5.7: Critical memory accesses on a free list push/pop.

While usually `free` is perfectly complementary to `malloc` and we rarely mention it, there is a marked difference in size class computation. `free` does not take a size parameter, only the pointer to be deallocated, so it must perform extra work to determine the size class to return it to. In TCMalloc, this is implemented by a hash lookup from the address being freed to the size class. This hash tends to cache poorly, especially in the TLB, leading to expensive losses. C++11 ameliorates this problem because the compiler can choose to call `operator delete()` with an extra parameter equal to the size of the object being freed, as long as the object's size can be determined at compile time. With `-fsized-deallocation`, the compiler prefers calling that variant when it can. In our results, we assume sized delete when applicable.

Push/pop a free list head Once a size-class is identified, all that is left is to pop (or push) the head of its free list. Pushing to or popping from a free list generates a dependent chain of three memory

accesses, as shown in Figure 5.7. In these cases, the most critical operations are the two loads on the pop path, because long-latency load misses can stall execution and commit of younger instructions. Since calls to the allocator are interspersed among application code, the free lists are prone to eviction, making these loads likely to miss. Figure 5.4 demonstrates this clearly with the `antagonist` microbenchmark, which emulates such cache-trashing behavior, and sees a significant increase in Pop time. In contrast, stores misses are less likely to stall the execution or commit of younger instructions, making the deallocation path less performance-critical.

TCMalloc uses a trick to save memory taken up by the free lists: it stores the next pointer at the address of the block of memory it is about to return, instead of allocating a separate field in a struct for it. That is, `*head` is the value of the `next` pointer, rather than a more familiar list node with fields `node->head` and `node->next`. In addition to reducing allocator memory overhead, dereferencing `head` to get the next pointer has the side effect of prefetching the returned memory block itself, which can likely help the caller.

Sampling For monitoring and debugging purposes, TCMalloc can also sample allocation requests every N bytes. A sampled allocation dumps and stores a stack trace in addition to performing the allocation itself. Sampling is invaluable in a production setting for analyzing memory usage and debugging memory leaks without having to stop, let alone recompile, live jobs, but it adds a measurable overhead to each malloc request, since a counter must be decremented and checked against the threshold each time.

Remaining instructions The three main steps described above account for $\approx$ 50% of fast path cycles. The remainder are split roughly evenly between: function call overhead (pushing / popping registers), addressing calculations (for example, of a free list in a thread cache) and updates to metadata fields (such as free list lengths and total size). While it is tempting to consider hard-coding the latter two in hardware, this would result in a rather narrow and inflexible accelerator, and severely limit its applicability to either other allocators, or even future revisions of the same allocator.

5.4 Mallacc: a malloc accelerator

Based on the characterization in the previous sections, we propose Mallacc, a fast-path malloc accelerator to augment a general-purpose processor. Broadly, Mallacc consists of a tiny dedicated malloc cache and a sampling performance counter. Its design requirements are extremely stringent. Since each fast-path call is on average only a few tens of cycles long, proposed structures must be inside cores, or access

Valid	Size range (index range)	Size class	Size	Head	Next
1	0 - 1	1	8	0x8080	0x8088
1	63 - 64	25	512	0x9090	0x9290
1	5 - 6	4	48	0x0	0x0
0	-	-	-	-	-

Figure 5.8: A malloc cache with example values. The cache is searched by first an associative lookup over requested size and later by size class. It stores the corresponding size class, and the first two free-list elements for that size class.

latency will erase any gains, which implies very tight area constraints. In addition, we would like to hard-code as few allocator-dependent details as possible (ideally none), so that many current and future allocators can benefit from acceleration. Our proposed design demonstrates that it is possible to meet these constraints, and the rest of this section describes it in detail. Our descriptions assume the x86 architecture, but the general principles and mechanisms are not x86-specific.

5.4.1 The malloc cache

In Section 5.3.3, we identified size class computation and popping the head of a free list as the biggest contributors to fast-path cycles, especially with cache-heavy workloads invoking the allocator. We can optimize both with a tiny, two-part cache that we call *the malloc cache*. Conceptually, it learns the mappings from requested allocation sizes to both the size class they correspond to and the first two elements of the free list for that size class. In the case of a malloc cache hit, computation can almost immediately return to the caller. By only storing the most frequently-accessed size classes, the cache can be kept extremely small (several entries). Lookups, updates and prefetches in the cache are software-managed, so it is also not tied to a particular allocator implementation. A four-entry cache, populated with some example values, is shown in Figure 5.8. We will go over its main components.

Size class mappings By definition a single size class represents a range of allocation sizes that get rounded up and given the same amount of memory. The malloc cache learns and stores the mappings from a requested size range to the size class representing it.

When a requested size comes in, it is associatively checked with the upper and lower bounds of the ranges that currently make up all cache entries. If the request size is inside a range, the access is declared a hit, and the cache returns the size class and its corresponding rounded size. More interestingly, on a miss, execution goes to a fallback path, where software is left to compute the size class as it ordinarily

```
def mcszlookup(ReqSize):
  IsHit = Cache.FindRangeContaining(ReqSize)
  if IsHit:
    SizeClass = Cache[ReqSize].SizeClass
    AllocSize = Cache[ReqSize].AllocSize
    ZF = 1
  else:
    ZF = 0
  return SizeClass, AllocSize

def mcszupdate(ReqSize, AllocSize, SizeClass):
  IsHit = Cache.FindSizeClass(SizeClass)
  if IsHit:
    SizeRange = Cache[SizeClass]
    if not SizeRange.Contains(ReqSize):
      SizeRange.LowerBound = ReqSize
  else:
    if Cache.Full():
      Cache.Evict()
    SizeRange = (ReqSize, AllocSize)
    Cache.InsertRange(SizeRange, SizeClass)
```

Figure 5.9: Pseudocode for size class instructions.

would. Software is then responsible to update the cache with the new (requested size, allocated size, size class) entry. The cache either inserts a new size class entry with the new range, or it expands the range for an already existing size class. If the cache is full for an insertion, an old entry is evicted based on an LRU policy.

The cache is managed by two new instructions: `mcszlookup` and `mcszupdate` (malloc cache size lookup/update). `mcszlookup` takes the requested allocation size in an input register and returns the size class and allocation size in two output registers if the requested size is found in the cache. The zero flag (ZF) is set if found and cleared if not. `mcszupdate` takes the original requested size, the computed size class, and the allocation size in three input registers and either inserts a new entry into the cache or updates an existing one. No registers are modified. Pseudocode for the instruction mnemonics is shown in Figure 5.9. Figure 5.10 is an assembly snippet demonstrating how they integrate with the rest of allocator code.

Our actual implementation has one additional optimization – instead of keying the array on the actual requested size range, we use the range of size class indices, as defined in Figure 5.5, and add dedicated hardware to compute the index from the requested size. Because the space of indices is sig-

```
Start:
  ; rax = size class (dest)
  ; rbx = allocated size (dest)
  ; rcx = requested size (source)
  mcszlookup rax, rbx, rcx    ; Sets ZF
  je ComputeSizeClass         ; if ZF = 1, jump.
Resume:
  ; Continue with the rest of malloc.

ComputeSizeClass:
  ; The usual software calculation for the size class (rax)
  ; and allocated size (rbx). Then update the cache.
  mcszupdate rcx, rbx, rax
  jmp Resume
```

Figure 5.10: Integration of size class instructions in an allocator.

nificantly smaller than the space of requested sizes, the cache can learn mappings faster, with fewer cold misses. The hard-coded hardware adds an additional cycle of latency to the cache, which we do model. It is the only TCMalloc-specific optimization in Mallacc, and can be disabled with a configuration register. In this mode, the malloc cache will build ranges of known requested sizes, although with slightly higher miss rates.

Free list caching An allocation call requires popping an element off a free list. As mentioned in Section 5.3.3, this is the most performance-critical part of a fast-path call, because it caches poorly and accesses memory prone to eviction by the application's own cache accesses. The malloc cache tackles this bottleneck by storing copies of the head and the *next* head of the free list associated with a size class alongside the size class mappings. Figure 5.8 illustrates this with an example.

Storing the first *two* list items in the malloc cache allows a Mallacc-enabled allocator to immediately return a value to the application after a hit. It also allows the next instruction in a linked list pop, the one that sets the head of the linked list to the current next element, to commit immediately without waiting for an often-required L2 or L3 miss in order to first fetch that next element. We find that second effect especially important, because the long-latency miss often blocks other otherwise-ready instructions from committing.

We introduce two new instructions to enable such operation. Most importantly, `mchdpop` (Figure 5.11) takes in the requested size class as an input (which we have ideally gotten from the previous optimization), and returns the cached copies of the first two list elements on a hit. If either of them is not present (NULL) in the cache entry, the access is declared a miss, the other one is also invalidated,

```
def mchdpop(SizeClass):
  Found = Cache.FindSizeClass(SizeClass)
  if Found:
    Head = Cache.GetHead(SizeClass)
    Next = Cache.GetNext(SizeClass)
    if Head and Next:
        Cache.SetHead(SizeClass, Next)
        Cache.InvalidateNext(SizeClass)
        ZF = 1
        return Head, Next
  ZF = 0
  return NULL, NULL

def mchdpush(SizeClass, NewHead):
  FoundEntry = Cache.FindSizeClass(SizeClass)
  if FoundEntry:
    CurrHead = Cache.GetHead(SizeClass)
    Cache.SetNext(SizeClass, CurrHead)
    Cache.SetHead(SizeClass, NewHead)

def mcnxtprefetch(SizeClass, NewNext):
  CurrHead = Cache.GetHead(SizeClass)
  CurrNext = Cache.GetNext(SizeClass)
  if CurrHead and not CurrNext:
    Cache.SetNext(NewNext)
  elif not CurrHead:
    Cache.SetHead(NewNext)
```

Figure 5.11: Pseudocode for linked list instructions.

and execution falls back on software to perform the pop (Figure 5.12). Its dual operation, `mchdpush`, is invoked on the deallocation side and updates the cached `Head` with the pointer being freed, shifting the previous head to the `Next` slot.

Note that these instruction are merely performance optimizations meant to isolate free lists from cache antagonists. The real free list head pointer is always valid and updated in software on both a hit and a miss, as is any metadata (length, total size, etc.).

For a pop operation to consistently hit, we need two elements already cached. To maintain that for differently-balanced allocation patterns (i.e., with different rates of allocations and deallocations over time), we introduce yet another instruction, `mcnxtprefetch`. Conceptually, `mcnxtprefetch` prefetches a memory location into the malloc cache's `Next` entry, and is called after a pop hits and moves its `Next` element in the `Head` position. In this case, a subsequent pop request can immediately

```
malloc:
  ; rax = size class.
  ; rbx = location of the head of the free list.
  ; rcx = returned: head element.
  ; rdx = returned: next head element.
  ; rdi = temporary.
  mchdpop       rcx, rdx, rax          ; Search malloc cache.
  je cache_fallback                    ; If we missed, go to fallback.
  mov           QWORD PTR [rbx], rdx   ; Otherwise, update head.
  ; ...                                ; ... and metadata.
  jmp malloc_ret
cache_fallback:
  ; Execute the original software.
  mov rcx, QWORD PTR [rbx]             ; head = *freelist->head.
  mov rdx, QWORD PTR [rcx]             ; next = *head.
  mov QWORD PTR [rbx], rdx             ; freelist->head = next.
malloc_ret:
  mcnxtprefetch  rax, QWORD PTR [rdx]   ; Prefetch the next head.
  ; Clean up stack and return value.

free:
  ; rax = freed pointer.
  ; rcx = size class.
  mchdpush  rcx, rax                   ; Update malloc cache head.
  ; The rest of free
```

Figure 5.12: Integration of linked list instructions in an allocator.

hit as long as the prefetch has had enough time to return from the cache hierarchy. While not necessary for correctness, enabling a prefetch to update the `Head` field of an empty cache entry as well as the `Next` field allows for the prefetch instruction to be called on a miss, and leads to higher hit rates. We assume that in Figure 5.12. Finally, to ensure that the copies of the list elements stored in the malloc cache are always consistent (`Head` always points to `Next`), entries with an outstanding prefetch block and do not service pushes or pops until the prefetch comes back, or gets rolled back on a misprediction.

Core integration First, it is important to point out that the malloc cache only stores copies of list elements for fast access – the definitive version of free lists is always in regular memory. Thus, at interrupts or context switches, the whole cache can always be flushed without writebacks or correctness concerns. Similarly, at branch mispredictions, entries from the mispredicted epoch can be discarded, just as they would in any other long-latency unit.

Second, as part of the core, the malloc cache can potentially see instructions out-of-order. In order

to not break the invariant that a cached `Head`'s next pointer always points to the adjacent `Next` element, our three linked list instructions are ordered with each other. We implement that by an implicit read-write register dependency through an architecturally-invisible register, which out-of-order execution has to observe. As discussed earlier, blocking the cache when a prefetch is outstanding is also required to preserve the linked list invariant.

Finally, the prefetch instruction is slightly unconventional. Like a software prefetch in L1, it is allowed to commit, so that it does not block subsequent code, but a result still has to make its way from the cache hierarchy to the malloc cache. From the core's point of view, this is treated in a virtually identical manner to a *store*, which is also allowed to commit with an outstanding memory access, but reserves a slot in a structure (sometimes called a senior store queue), where it waits for an acknowledgment from coherency controllers.

5.4.2 Sampling

The sampling code in TCMalloc (and its equivalents in `jemalloc` [37] and others) presents an additional opportunity to remove several additional cycles from the allocation critical path. The operation performed by the sampler – accumulate a value and capture a stack trace at a threshold – is precisely what a performance counter does and what the `perf_events` subsystem performs when the performance monitoring unit (PMU) raises an interrupt on an event. We propose dedicating a hardware performance counter for sampling allocation sizes, which entirely removes a conditional branch on the fast path. Stack traces are only required when a user explicitly requests them, and this can be handled through the `perf_events` interface as it typically is currently.

The main difference between our proposal and current performance counters is that it will need to increment by the value of a register, which holds the requested allocation size. As a result, the PMU will need access to the actual register file (or just the ROB), instead of the more typical occupancy statistics. As the design of a performance counter is fairly straightforward, we will focus on the design of the malloc cache for the remainder of this text.

5.5 Methodology

To evaluate Mallacc, we ran simulations on two systems – a conventional aggressive out-of-order processor as a baseline, and the same processor equipped with Mallacc, as described in the previous section. We also performed limit studies on our optimizations for an optimistic upper bound of speedup. To do so, we ran simulations in which the instructions comprising the three steps from Section 5.3.3 are

simply ignored by performance simulation.

Microbenchmarks To better understand allocator performance and the effect from our optimizations, we first constructed a suite of microbenchmarks to stress certain aspects of the fast path code. They are divided into two categories based on their allocation patterns: strided and Gaussian. Strided benchmarks allocate in increments of N bytes, up to some value, while Gaussian benchmarks issue allocation requests by drawing from normal distributions. All strided benchmarks fit perfectly in L1 caches and represent the very best baseline cases. Gaussian benchmarks allocate more, have subsequently larger working sets and more interesting caching behavior.

- `tp`: A throughput-oriented microbenchmark. It performs a series of back-to-back `malloc` and `free` calls, which always hit in thread caches. Each iteration strides through request sizes in increments of 16 bytes from 32 to 512 bytes.
- `tp_small`: Same as above, but we only stride up to 128 bytes. This ensures that: (i) each iteration accesses a different free list; and (ii) we only use four size classes. This microbenchmark captures the fastest possible fast path on the allocation side.
- `sized_deletes`: A variant of `tp_small` that uses eight size classes and sized deletes to speed up deallocation.
- `gauss`: A more realistic allocation pattern. `gauss` chooses randomly between small (16-64 byte) and relatively large (256-512 byte) allocations. 90% of allocations are chosen from the small set to represent typical behaviors for strings and small lists. Within each range, the size is selected from a Gaussian distribution. However, no objects are ever freed, which renders free lists virtually useless. This is a lower bound on the gains from any free-list centric optimizations.
- `gauss_free`: Same allocation behavior as `gauss`, but it randomly frees allocated memory with 50% probability.
- `antagonist`: Same allocation behavior as `gauss_free`, but after every allocation, invokes a simulator callback which evicts the less used half of each set of the L1 and L2 data caches. This mimics the behavior of an application that strides through a large working set, without simulating the millions of instructions required for the stride.

All microbenchmarks explicitly minimize the number of instructions between allocator calls (which is important when each call is only 40 instructions) and are run with sufficient warmup time.

	cycle error (%)
gauss	5.32
gauss_free	3.67
tp	12.3
tp_small	5.92
sized_delete	4.21
Average	6.28

Table 5.1: Simulator validation on `malloc` microbenchmarks.

Macrobenchmarks We evaluate our optimizations on the four benchmarks from SPEC CPU2006 that use the system allocator and two workloads that are more likely to be found in datacenters. For datacenter-like workloads, we use the `xapian` open-source search engine and the `masstree` [83] key-value store. `xapian` is configured as a leaf node and performs searches on an index of 1.6 million English Wikipedia pages, as well as on a smaller index of the same number of page abstracts. The set of queries focus on popular Wikipedia pages, obtained from Wikipedia's weekly top 25 article digests. For `masstree`, we run its `wcol1` and `same` performance tests. For SPEC benchmarks, we simulate several SimPoints [112] of 1B instructions each per benchmark, for `xapian` we skip query parsing and only simulate query execution, and run `masstree` from start until completion.

Allocator We use TCMalloc at revision `050f2d`. To model our proposed instructions, we annotate potential optimization sites in TCMalloc code by inserting special x86 marker instructions. Later, in simulation we replace these instructions with our proposals. These marker instructions were carefully chosen to a) not already appear in TCMalloc and b) have the same number and type of operands as our proposed instructions. This is done so the compiler does not optimize surrounding code sub-optimally.

Compiler All benchmarks and allocators are built with GCC 6.1 at -O3 with `-fsized-deallocation`.

Simulator All experiments are run using the XIOSim simulator [60], configured for an aggressive out-of-order core modeled after an Intel Haswell microarchitecture. Since we are evaluating `malloc` fast path code, we validated our performance model on microbenchmarks against a Haswell desktop processor and achieved a mean error of 6.3% (Table 5.1). We omitted `antagonist` because it uses a simulator callback to emulate cache trashing and does not run natively.

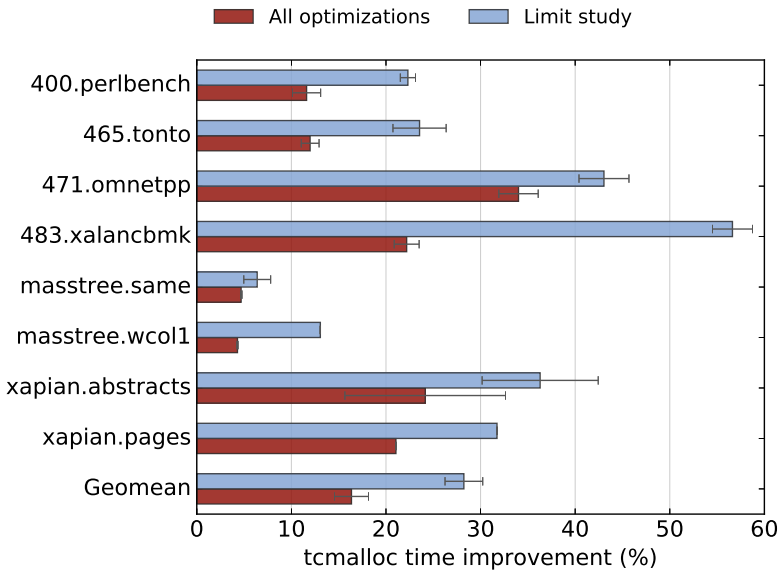


Figure 5.13: Improvement of time spent in the allocator.

5.6 Results

5.6.1 Allocator time speedup

Figure 5.13 shows the reduction of time spent in allocator code for our SPEC and cloud workloads. These results use a 32-entry malloc cache to demonstrate the potential of our accelerator; we will later present a cache size sensitivity study. On the total time spent in the allocator (including both `malloc` and `free`), Mallacc is able to achieve an average of 18% speedup, out of 28% projected by the limit study. Most of that is due to improvements on `malloc` calls, which see an average of nearly 30% speedup (Figure 5.14). The amount of speedup achieved is highly correlated with the fraction of time on the fast path shown in prior sections. We call out three particular benchmarks to get a deeper understanding of the causes for improvement, or lack thereof.

Xapian `xapian` uses a very small set of size classes, and its `malloc` calls almost exclusively take the fast path. As shown in Section 5.3, this is true whether it is searching over an index of small documents (abstracts) or an index of large documents (full articles). This makes `xapian` a great candidate for fast path acceleration and Figure 5.14 confirms that – the malloc cache provides over 40% speedup on `malloc` calls.

Figure 5.15 implies that the causes for this improvement are the latency-reducing portions of Mallacc–

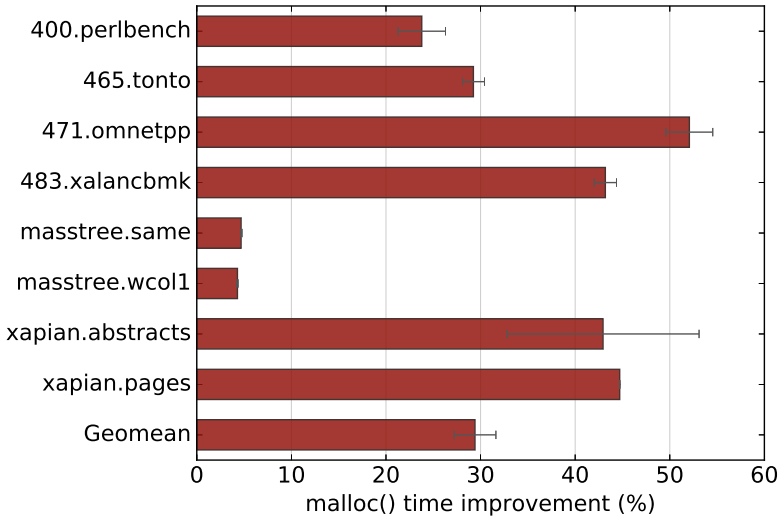


Figure 5.14: Improvement in time spent on `malloc()` calls (both fast and slow paths).

size class lookups, sampling, and, to a much smaller degree, linked list caching. It is a distribution of time im `malloc` calls over the call duration for three cases: the baseline implementation, our limit study, and Mallacc. The baseline case is already very fast – with virtually all calls between 20 and 40 cycles, not unlike our striding microbenchmarks, which implies very small effects from cache antagonism. Our best-case latency optimizations manage to reduce the average call length almost twofold, with median calls now at 13 cycles, and a distribution very close to that of the limit study. The size class cache in particular is very effective because of the small number of size classes used by `xapian`.

Xalancbmk As demonstrated by Figure 5.2, `xalancbmk` uses the most number of size classes, requiring 30 size classes for 90% coverage. Nevertheless, it has enough size class locality to also benefit from Mallacc, achieving over 40% speedup on `malloc` calls. Figure 5.16 shows the `malloc` call duration distribution for this benchmark. The first spike corresponds to the fastest of fast path calls, where the effects are similar to those seen in `xapian`. The next large spike, between 20 and 70 cycles includes fast path calls that missed in L1 and L2 caches and had to go to L3 (34 cycles latency on Haswell). The malloc cache is particularly beneficial in this region because of its cache isolation properties. Finally, note that Mallacc only improves fastpath behavior without affecting slower calls.

Masstree `masstree` has the lowest overall malloc speedup of all the workloads we tested. As we pointed out in Section 5.3.2, this is because the `masstree` performance tests never free any memory, so many `malloc` calls must request large amounts from the page allocator. The little time spent on

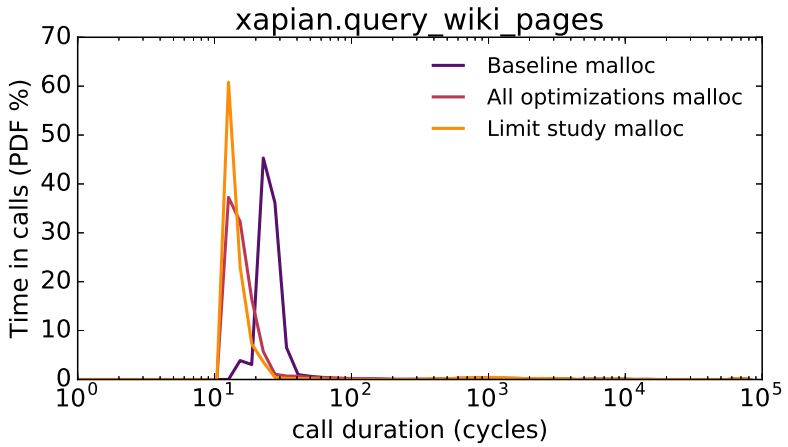


Figure 5.15: Xapian sees a significant improvement on already-fast calls.

the fast path results in an allocator time improvement of just 5%. However, a real deployment of `masstree` would inevitably free more memory, and likely have significantly higher thread-cache use, so we would expect different results.

5.6.2 Sensitivity to malloc cache size

The malloc cache is a part of the core, where silicon real estate is expensive, so we must maximize our performance gains with the least number of entries. To understand the effects of malloc cache sizing, we sweep malloc cache sizes from 2 to 32 on our suite of microbenchmarks. The results of this sweep are shown in Figure 5.17. We non-surprisingly find that too small of a cache will result in slowdown rather than speedup. At a high enough miss rate, not only is execution going through the fallback paths (the same instructions that we started optimizing away), but also with the additional cache lookups to determine that. However, once the cache is large enough to capture the majority of allocation requests, we quickly achieve speedup. One example are the strided benchmarks, which have *no* size class locality until we can capture *all* of their requests, resulting in very sharp jumps. `sized_deletes`, `tp`, and `tp_small` use 8, 25, and 4 size classes, respectively, and we see that the speedup inflection points occur precisely at those malloc cache sizes. The Gaussian benchmarks have more size class locality because they are more likely to allocate small size classes, which results in a more gradual increase in speedup until cache size 12, because Gaussian benchmarks allocate from 13 possible size classes.

Once the malloc cache is sufficiently sized, Mallacc can achieve within 10-20% of ideal speedup. The lone exception here is `tp`. For certain points of execution, this microbenchmark starts allocating and deallocating from the same size class in a very tight loop ($\approx$ 30 cycles for a malloc-free pair). In

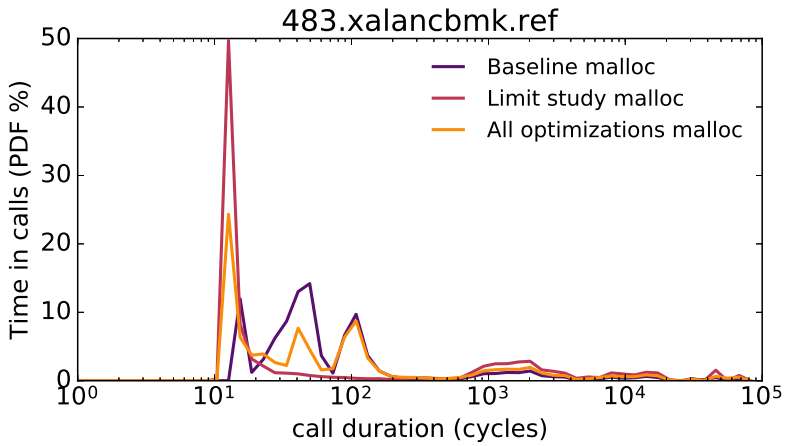


Figure 5.16: Xalan benefits both from latency reduction and cache isolation.

this case, the malloc cache blocks until each of the malloc prefetches returns with a value, causing the slowdown. The prefetch instruction is based on exactly the opposite assumption – that there is enough time between requests to prefetch for the next one and this slowdown is expected. None of our macro workloads exhibit slowdown due to prefetch blocking.

It is important to remember that these microbenchmarks are designed to stress the fast path of malloc, not to exhibit realistic allocation behavior. As we showed in Figure 5.6, most benchmarks use a very small number of size classes. We swept malloc cache sizes and only `xalancbmk` is meaningfully affected by a smaller size – it loses 6 percentage points of allocator time improvement between 32 and 16. Because of that, we consider a size of 16 sufficient for most workloads.

5.6.3 Full program speedup

Finally, we present improvements on full benchmark execution time, not only allocator time. This speedup is obviously bounded by the total time each benchmark spends in the allocator itself. Figure 5.18 shows these fractions for our workloads, compared to published data from Google's datacenters [58]. Most of our workloads spend a much lower fraction of time in allocator code, so we can expect small gains. As mentioned before, the `masstree` performance tests have very high `malloc` time because they exclusively allocate memory and never free any, resulting in many slow path calls.

Table 5.2 shows full program speedup for workloads where the measured speedup through simulation is statistically significant. For them, the mean program speedup is 0.49%, with a maximum of 0.78% for `perlbench`.

Because total speedup tends to be small, run-to-run variance on some of the workloads is enough

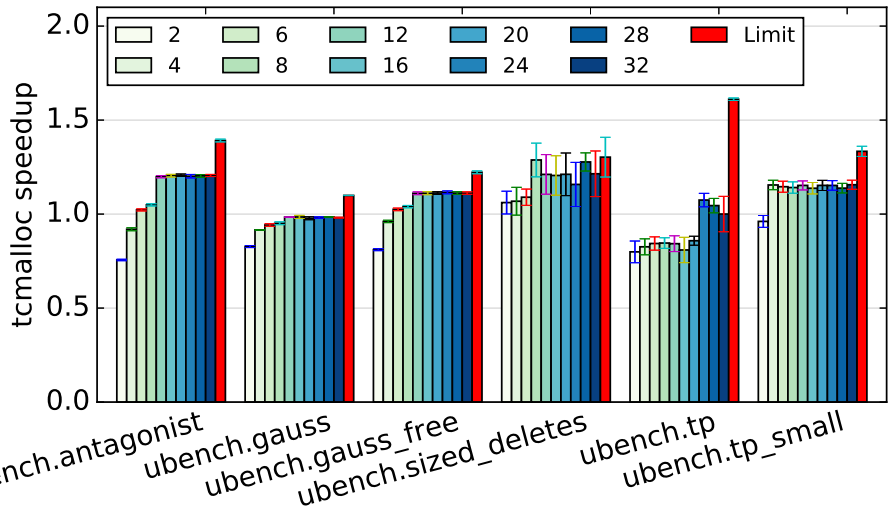


Figure 5.17: Effect of cache size on overall tcmalloc speedup.

to mask out any improvements we achieved with the `malloc` accelerator. More precisely, we do not include the workloads for which a single-sided Student's T-test fails to reject a hypothesis of slowdown with 95+% probability. Note that for these workloads, the speedup *in allocator code* is still statistically significant, as shown by the low error bars in Figure 5.13. It is simply more difficult to reliably estimate its effects on full program execution time due to the high variance.

5.6.4 Area cost of Mallacc

Mallacc consists of the malloc cache and a performance counter. The malloc cache requires 152 bits of storage per entry. Because the malloc cache is fully associative, it must be implemented using content addressable memories (CAMs) and standard SRAM. CAM cells can be significantly larger than SRAM cells, but the malloc cache requires so few bits that this difference is negligible. We do not lay out the malloc cache to provide precise area estimates, but it is so small that a reasonable upper bound will suffice. Also, we will ignore the area overhead of the performance counter, since it is just one 64-bit register per hardware thread.

The malloc cache requires three CAM arrays to implement the index and size class search and LRU functions, while the rest of the data – allocated size, list pointers, and valid bits – can be stored in an SRAM array. The index CAM requires 24 bits per entry to store two 12-bit indices, while the size class CAM requires 8 bits per entry to store size classes, and the LRU CAM stores $\log_2 n$ bits per entry, where n is the number of entries. The head pointer SRAM array requires 117 bits per entry to store two

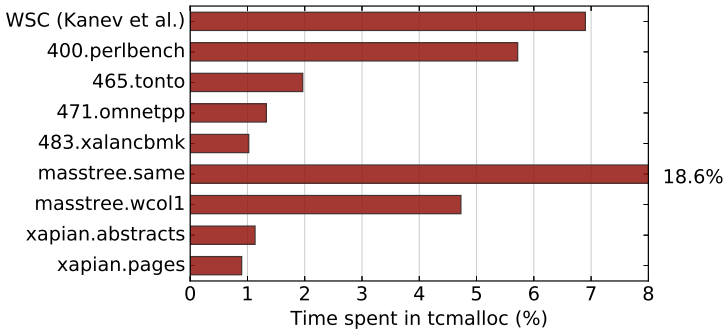


Figure 5.18: Fraction of time spent in the allocator.

48-bit pointers (currently, x86 only uses the lower 48-bits of 64-bit addresses), 20 bits for the allocated size, plus a valid bit. Our analysis has shown 16 entries to be sufficient for the workloads analyzed; this means the CAMs and SRAM are 72 bytes and 234 bytes, respectively. We also need a shifter and adder for the index computation.

We used CACTI 6.5+ [73] to estimate the sizes of these four arrays in 28nm. The CAMs collectively occupy 873 μm^2 and the SRAM occupies 346 μm^2 for a total of 1219 μm^2. This is certainly a pessimistic upper bound; Jeloka et al. recently demonstrated a 512 byte configurable CAM array occupying merely 1208 in 28nm μm^2 [116]. We scale published area numbers of shifters and adders from accelerator models [110] by ITRS technology scaling factors and estimate a total area of 265 μm^2, bringing our upper bound to about 1500 μm^2.

Consider this area in the context of a typical high-performance CPU. An Intel Haswell core measures 26.5 mm^2 (including private L1 and L2 caches). If integrated into a Haswell CPU, Mallacc is merely 0.006% of the core area. Pollack's Rule states that historically, the performance increase of a chip is approximately proportional to the square root of the increase in complexity, where complexity refers to area [13]. By this rule, an area increase of 0.006% would only produce 0.003% speedup. In contrast, Mallacc demonstrates average speedup of 0.49%, which is over 150× greater. It is clear that Mallacc far surpasses the "1% performance for 1% area" rule of thumb that has informally guided processor development over the last few decades.

5.7 Conclusion

Dynamic memory allocation is a widely used programming paradigm that has seen decades of software research and optimization. Recent work has discovered that despite being well-optimized, memory allocation can consume a significant percentage of datacenter cycles. In this work, we present Mallacc,

	Speedup	Stddev	p-value
400.perlbench	0.78%	0.05%	<0.001
465.tonto	0.35%	0.08%	0.025
483.xalancbmk	0.27%	0.06%	0.043
masstree.same	0.49%	0.05%	0.002
xapian.abstracts	0.55%	0.05%	0.002

Table 5.2: Full program speedup.

a tiny in-core hardware block for accelerating dynamic memory allocation. Mallacc does not implement a new allocator; rather, it is designed to accelerate various operations that are common to many existing high-performance allocators. Unlike many hardware accelerators that design for maximum throughput, Mallacc is designed to minimize latency. We show that Mallacc can accelerate the most commonly observed `malloc` behavior – fast allocation requests that only take 20-30 cycles on modern processors – by up to 50%, while consuming less than 1500μm^2 of silicon area, and that integrating Mallacc into a CPU provides speedups that greatly outstrip "1% performance for 1% area", a rule of thumb that has for decades informally guided high-performance processor design.

6
Conclusion

With the increasing popularity of online and cloud services, designing and managing the architectures for warehouse-scale machines is becoming ever more relevant. The vast scale of WSCs has enabled significant improvements in the total cost per computation, but has also increased the startup cost of performing datacenter research. The first portions of this dissertation address this issue by characterizing real, live, production-grade WSCs over several years by and exposing several opportunities for improvement in both performance and power efficiency.

We have shown that WSC workloads are neither completely CPU- nor IO-bound. Instead, they mix bursts of computation with short periods of sleep, emphasizing the need for comprehensive sleep state selection algorithms. For them, additional power savings are possible while not sleeping, too, but only after a careful and workload-specific frequency scaling policy.

We have also demonstrated the large diversity in WSCs, both in terms of the applications themselves, and within each individual one. Despite the large variance, we can identify a common microarchitectural 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.

In the long term, as technology scaling slows down, specialized architectures are a clear path to efficiency gains in the datacenter. However, their adoption in WSCs so far has been limited. Our profiling results attribute this to the very wide diversity in applications in a typical WSC. By profiling across binary boundaries, we have identified common low-level functions ("datacenter tax"), which

show potential for specialized hardware in future server SoCs. Such "broad" accelerators have very different design constraints from typical specialized blocks. The benefits from each single one is limited, but so is the overhead it incurs, so a large collection of broad accelerators can still be a significant net efficiency win.

We demonstrated the feasibility of the broad acceleration approach by designing a specialized block for dynamic memory allocation – one of the most popular (and perhaps most heavily optimized) components of the datacenter tax. Our prototype implementation improves `malloc` latency by up to 50%, while incurring a negligible 0.006% silicon area overhead. We believe this will spur additional research interest in designing collections of broad accelerators, and their eventual adoption in warehouse-scale systems-on-chip.

References

[1] J. H. Ahn, S. Li, O. Seongil, and N. P. Jouppi, "McSimA+: A manycore simulator with application-level+ simulation and detailed microarchitecture modeling," in *Performance Analysis of Systems and Software (ISPASS)*, 2013.

[2] H. Amur *et al.*, "Idlepower: Application-aware management of processor idle states," *MMCS, in conjunction with HPDC*, 2008.

[3] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: A fast array of wimpy nodes," in *Operating systems principles (SOSP)*, 2009.

[4] J. Anderson, L. Berc, G. Chrysos, J. Dean, S. Ghemawat, J. Hicks, S.-T. Leung, M. Lichtenberg, M. Vandevoorde, C. A. Waldspurger *et al.*, "Transparent, low-overhead profiling on modern processors," in *Workshop on Profile and Feedback-Directed Compilation*, 1998.

[5] M. Annavaram, J. M. Patel, and E. S. Davidson, "Call graph prefetching for database applications," *Transactions of Computer Systems*, 2003.

[6] T. M. Austin and G. S. Sohi, "High-bandwidth address translation for multiple-issue processors," in *Computer Architecture (ISCA)*, 1996.

[7] L. Barroso and U. Hölzle, "The case for energy-proportional computing," *IEEE Computer*, 2007.

[8] L. A. Barroso, J. Clidaras, and U. Hölzle, "The datacenter as a computer: an introduction to the design of warehouse-scale machines," *Synthesis Lectures on Computer Architecture*, 2013.

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

[10] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A Scalable Memory Allocator for Multithreaded Applications," in *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, 2000.

[11] M. Bligh *et al.*, "Linux kernel debugging on google-sized clusters," in *Linux Symposium*, 2007.

[12] P. Bohrer *et al.*, "The case for power management in web servers," *Power aware computing*, 2002.

[13] S. Borkar and A. A. Chien, "The future of microprocessors," *Communications of the ACM*, 2011.

[14] K. Brownell, "Architectural Implications of Automatic Parallelization With HELIX-RC," Ph.D. dissertation, Harvard University, 2015.

[15] D. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," 1997.

[16] P. Calafiura, S. Eranian, D. Levinthal, S. Kama, and R. A. Vitillo, "GOoDA: The generic optimization data analyzer," in *Journal of Physics: Conference Series*, 2012.

[17] H. Cam, M. Abd-El-Barr, and S. M. Sait, "A high-performance hardware-efficient memory allocation technique and design," in *Computer Design (ICCD)*, 1999.

[18] S. Campanoni, G. Agosta, S. Crespi Reghizzi, and A. Di Biagio, "A highly flexible, parallel virtual machine: design and experience of ILDJIT," *Software: Practice and Experience*, 2010.

[19] S. Campanoni, K. Brownell, S. Kanev, T. M. Jones, D. Brooks, and G.-Y. Wei, "HELIX-RC: An Architecture-Compiler Co-Design for Automatic Parallelization of Irregular Programs," in *Computer Architecture (ISCA)*, 2014.

[20] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: Exploring the Level of Abstraction for Scalable and Accurate Parallel Multi-Core Simulations," in *High Performance Computing, Networking, Storage and Analysis (SC)*, 2011.

[21] N. Chachmon, D. Richins, R. Cohn, M. Christensson, W. Cui, and V. J. Reddi, "Simulation and analysis engine for scale-out workloads," in *International Conference on Supercomputing (ICS)*, 2016.

[22] F. Chang *et al.*, "Bigtable: A distributed storage system for structured data," *ACM Transactions on Computer Systems (TOCS)*, 2008.

[23] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Operating Systems Design and Implementation (OSDI)*, 2006.

[24] J. M. Chang and E. F. Gehringer, "A high performance memory allocator for object-oriented systems," *Transactions on Computers*, 1996.

[25] J. M. Chang, W. Srisa-An, and C.-T. Lo, "Architectural support for dynamic memory management," in *Computer Design (ICCD)*, 2000.

[26] M. Charney, "XED: An x86 encoder/decoder," http://goo.gl/ORn5JH.

[27] D. Chen, N. Vachharajani, R. Hundt, S.-w. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng, "Taming hardware event samples for FDO compilation," in *Code generation and optimization (CGO)*, 2010.

[28] G. O. Collins Jr, "Experience in automatic storage allocation," *Communications of the ACM*, 1961.

[29] W. Comfort, "Multiword list items," *Communications of the ACM*, 1964.

[30] G. Contreras, M. Martonosi, J. Peng, R. Ju, and G.-Y. Lueh, "XTREM: a power simulator for the Intel XScale core," in *ACM Sigplan Notices*, 2004.

[31] Z. Dai, N. Ni, and J. Zhu, "A 1 cycle-per-byte XML parsing accelerator," in *Field Programmable Gate Arrays*, 2010.

[32] A. de Melo, "The new linux "perf" tools," in *Linux Kongress*, 2010.

[33] A. C. de Melo, "The new linux 'perf' tools," in *Slides from Linux Kongress*, 2010.

[34] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, 2013.

[35] F. Duarte and S. Wong, "Cache-based memory copy hardware accelerator for multicore systems," *IEEE Transactions on Computers*, 2010.

[36] J. Evans, "A Scalable Concurrent malloc Implementation for FreeBSD," in *Proceedings of the Technical BSD Conference*, 2006.

[37] ——, "Scalable memory allocation using jemalloc," https://goo.gl/rvl20K, 2011.

[38] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith, "A top-down approach to architecting cpi component performance counters," *IEEE Micro*, 2007.

[39] M. Ferdman, B. Falsafi, A. Adileh, O. Kocberber, S. Volos, M. Alisafaee, D. Jevdjic, C. Kaynak, A. D. Popescu, and A. Ailamaki, "Clearing the clouds," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[40] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive instruction fetch," in *Microarchitecture (MICRO)*, 2011.

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

[42] S. Ghemawat and P. Menage, "TCMalloc: Thread-caching malloc," http://goog-perftools.sourceforge.net/doc/tcmalloc.html, 2007.

[43] Google, "Bazel," http://bazel.io/.

[44] ——, "Efficiency: How we do it," https://www.google.com/about/datacenters/efficiency/internal/.

[45] ——, "gRPC," http://grpc.io/.

[46] ——, "Protocol buffers," https://developers.google.com/protocol-buffers/.

[47] J. Hamilton, "Cooperative expendable micro-slice servers (CEMS): low cost, low power servers for internet-scale services," in *Conference on Innovative Data Systems Research (CIDR)*, 2009.

[48] ——, "Counting servers is hard," http://perspectives.mvdirona.com/2013/07/counting-servers-is-hard/, 2013.

[49] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*, 2012.

[50] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in *High Performance Computer Architecture (HPCA)*, 2015.

[51] C.-H. Hsu and W.-c. Feng, "Effective dynamic voltage scaling through CPU-boundedness detection," in *Workshop on Power-Aware Computer Systems*, 2004.

[52] C. Isci *et al.*, "Live, Runtime Phase Monitoring and Prediction on Real Systems with Application to Dynamic Power Management," in *Microarchitecture (MICRO)*, 2006.

[53] A. 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.

[54] A. Jaleel, J. Nuzman, A. Moga, S. C. Steely Jr, and J. Emer, "High Performing Cache Hierarchies for Server Workloads," in *High-Performance Computer Architecture (HPCA)*, 2015.

[55] V. Janapa Reddi, B. C. Lee, T. Chilimbi, and K. Vaid, "Web search using mobile cores: quantifying and mitigating the price of efficiency," *Computer Architecture (ISCA)*, 2010.

[56] Z. Jia, L. Wang, J. Zhan, L. Zhang, and C. Luo, "Characterizing data analysis workloads in data centers." in *Workload characterization (IIWSC)*, 2013.

[57] M. Kambadur, T. Moseley, R. Hank, and M. A. Kim, "Measuring interference between live datacenter applications," in *High Performance Computing, Networking, Storage and Analysis (SC)*, 2012.

[58] S. Kanev, J. P. Darago, K. Hazelwood, T. Parthasarathy, Ranganathan amd Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Computer Architecture (ISCA)*, 2015.

[59] S. Kanev, K. Hazelwood, G.-Y. Wei, and D. Brooks, "Tradeoffs between Power Management and Tail Latency in Warehouse-Scale Applications," in *Workload Characterization (IISWC)*, 2014.

[60] S. Kanev, G.-Y. Wei, and D. Brooks, "XIOSim: power-performance modeling of mobile x86 cores," in *Low-power electronics and design (ISLPED)*, 2012.

[61] S. Kaxiras and M. Martonosi, "Computer Architecture Techniques for Power-Efficiency," *Synthesis Lectures on Computer Architecture*, 2008.

[62] W. Kim *et al.*, "System level analysis of fast, per-core DVFS using on-chip switching regulators," in *High Performance Computer Architecture (HPCA)*, 2008.

[63] K. C. Knowlton, "A fast storage allocator," *Communications of the ACM*, vol. 8, no. 10, 1965.

[64] A. Kolli, A. Saidi, and T. F. Wenisch, "RDIP: Return-address-stack Directed Instruction Prefetching," in *Microarchitecture (MICRO)*, 2013.

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

[66] S. Kumar, A. Shriraman, V. Srinivasan, D. Lin, and J. Phillips, "SQRL: Hardware Accelerator for Collecting Software Data Structures," in *Parallel architectures and compilation (PACT)*, 2014.

[67] E. Le Sueur and G. Heiser, "Slow down or sleep, that is the question," in *USENIX Annual Technical Conference (USENIX ATC)*, 2011.

[68] B. C. Lee, "Datacenter Design and Management: A Computer Architect's Perspective," *Synthesis Lectures on Computer Architecture*, 2016.

[69] S. Lee, T. Johnson, and E. Raman, "Feedback directed optimization of tcmalloc," in *Proceedings of the workshop on Memory Systems Performance and Correctness*, 2014.

[70] O. Lempel, "2nd generation intel core processor family: Intel core i7, i5 and i3," in *Hot Chips*, 2011.

[71] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in *Symposium on Cloud Computing (SoCC)*, 2014.

[72] P. Li, J. L. Shin, G. Konstadinidis, F. Schumacher, V. Krishnaswamy, H. Cho, S. Dash, R. Masleid, C. Zheng, Y. David Lin *et al.*, "A 20nm 32-Core 64MB L3 cache SPARC M7 processor," in *Solid-State Circuits Conference (ISSCC)*, 2015.

[73] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009.

[74] W. Li, S. P. Mohanty, and K. Kavi, "A page-based hybrid (software-hardware) dynamic memory allocator," *Computer Architecture Letters (CAL)*, 2006.

[75] W. Li, M. Rezaei, K. Kavi, A. Naz, and P. Sweany, "Feasibility of decoupling memory management from the execution pipeline," *Journal of Systems Architecture*, 2007.

[76] 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 *Computer Architecture (ISCA)*, 2008.

[77] D. Lo *et al.*, "Towards energy proportionality for large-scale latency-critical workloads," in *Computer Architecture (ISCA)*, 2014.

[78] G. H. Loh, S. Subramaniam, and Y. Xie, "Zesto: A cycle-level simulator for highly detailed microarchitecture exploration," in *Performance Analysis of Systems and Software (ISPASS)*, 2009.

[79] P. Lotfi-Kamran, B. Grot, M. Ferdman, S. Volos, O. Kocberber, J. Picorel, A. Adileh, D. Jevdjic, S. Idgunji, E. Ozer *et al.*, "Scale-out processors," in *Computer Architecture (ISCA)*, 2012.

[80] Y. Lu *et al.*, "Quantitative comparison of power management algorithms," in *Design, Automation and Test in Europe (DATE)*, 2000.

[81] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," in *Program Language Design and Implementation (PLDI)*, 2005.

[82] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz, "Towards energy-proportional datacenter memory with mobile DRAM," *Computer Architecture (ISCA)*, 2012.

[83] Y. Mao, E. Kohler, and R. Morris, "Cache craftiness for fast multicore key-value storage," in *EuroSys*, 2012.

[84] J. Mars and L. Tang, "Whare-map: Heterogeneity in "homogeneous" warehouse-scale computers," in *Computer Architecture (ISCA)*, 2013.

[85] 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 *Microarchitecture (MICRO)*, 2011.

[86] D. Meisner *et al.*, "Bighouse: A simulation infrastructure for data center systems," in *Performance Analysis of Systems and Software (ISPASS)*, 2012.

[87] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in *Computer Architecture (ISCA)*, 2011.

[88] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis, "Dremel: interactive analysis of web-scale datasets," *Very Large Data Bases (VLDB)*, 2010.

[89] D. Namiot and M. Sneps-Sneppe, "On micro-services architecture," *Open Information Technologies*, 2014.

[90] J. Ouyang, H. Luo, Z. Wang, J. Tian, C. Liu, and K. Sheng, "FPGA implementation of GZIP compression and decompression for IDC services," in *Field-Programmable Technology (FPT)*, 2010.

[91] L. Page *et al.*, "The PageRank citation ranking: bringing order to the web." 1999.

[92] V. Pallipadi *et al.*, "cpuidle: Do nothing, efficiently," in *Linux Symposium*, 2007.

[93] M. P. Papazoglou and W.-J. Van Den Heuvel, "Service oriented architectures: approaches, technologies and research issues," *The VLDB journal*, 2007.

[94] H. Patil, R. Cohn, M. Charney, R. Kapoor, A. Sun, and A. Karunanidhi, "Pinpointing representative portions of large intel® itanium® programs with dynamic instrumentation," in *Microarchitecture (MICRO)*, 2004.

[95] D. A. Patterson, "The data center is the computer," *Communications of the ACM*, 2008.

[96] R. Pike *et al.*, "Interpreting the data: Parallel analysis with Sawzall," *Scientific Programming*, 2005.

[97] A. Putnam, A. M. Caulfield, E. S. Chung, D. Chiou, K. Constantinides, J. Demme, H. Esmaeilzadeh, J. Fowers, G. P. Gopal, J. Gray *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *Computer Architecture (ISCA)*, 2014.

[98] W. Qadeer, R. Hameed, O. Shacham, P. Venkatesan, C. Kozyrakis, and M. A. Horowitz, "Convolution engine: balancing efficiency & flexibility in specialized computing," in *Computer Architecture (ISCA)*, 2013.

[99] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Microarchitecture (MICRO)*, 2006.

[100] R. Raghavendra *et al.*, "No power struggles: Coordinated multi-level power management for the data center," in *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[101] K. Rajamani *et al.*, "Power management solutions for computer systems and datacenters," in *Low Power Electronics and Design (ISLPED)*, 2008.

[102] K. K. Rangan *et al.*, "Thread motion: fine-grained power management for multi-core systems," in *Computer Architecture (ISCA)*, 2009.

[103] B. Reagen, P. Whatmough, R. Adolf, S. Rama, H. Lee, S. K. Lee, J. M. Hernández-Lobato, G.-Y. Wei, and D. Brooks, "Minerva: Enabling low-power, highly-accurate deep neural network accelerators," in *Computer Architecture (ISCA)*, 2016.

[104] C. Reiss *et al.*, "Heterogeneity and dynamicity of clouds at scale," in *Symposium on Cloud Computing (SoCC)*, 2012.

[105] 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.

[106] E. Rotem *et al.*, "Power-management architecture of the Intel microarchitecture code-named Sandy Bridge," *IEEE Micro*, 2012.

[107] D. Sanchez and C. Kozyrakis, "ZSim: fast and accurate microarchitectural simulation of thousand-core systems," *Computer Architecture (ISCA)*, 2013.

[108] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry, "RowClone: Fast and Energy-efficient in-DRAM Bulk Data Copy and Initialization," in *Microarchitecture (MICRO)*, 2013.

[109] Y. S. Shao, "Design and modeling of specialized architectures," Ph.D. dissertation, Harvard University, 2016.

[110] Y. S. Shao, B. Reagen, G.-Y. Wei, and D. Brooks, "The Aladdin Approach to Accelerator Design and Modeling," *IEEE Micro*, 2015.

[111] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically characterizing large scale program behavior," in *Computer Architecture (ISCA)*, 2002.

[112] ——, "Automatically characterizing large scale program behavior," in *Computer architecture (ISCA)*, 2002.

[113] V. Spiliopoulos *et al.*, "Green governors: A framework for Continuously Adaptive DVFS," *International Green Computing Conference and Workshops*, 2011.

[114] M. Stansberry and J. Kudritzki, "Uptime institute 2014 data center industry survey," *Uptime Institute Survey*, 2014.

[115] C. Stephenson, "New methods for dynamic storage allocation (fast fits)," in *Operating systems principles (SOSP)*, 1983.

[116] Supreet Jeloka and Naveen Bharathwaj Akesh and Dennis Sylvester and David Blaauw, "A 28nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory," *IEEE Journal of Solid-State Circuits*, vol. 51, no. 4, 2016.

[117] M. Tadman, "Fast-fit: A new hierarchical dynamic storage allocation technique," Master's thesis, 1978.

[118] J. Van Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson, "XML accelerator engine," in *Workshop on High Performance XML Processing*, 2004.

[119] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at Google with Borg," in *European Conference on Computer Systems (EuroSys)*, 2015.

[120] D. Wakabayashi, "Apple to build data command center in Arizona," *The Wall Street Journal*, 2015.

[121] P. R. Wilson, M. S. Johnston, M. Neely, and D. Boles, "Dynamic Storage Allocation: A Survey and Critical Review," in *International Workshop on Memory Management*, 1995.

[122] D. Wong and M. Annavaram, "Knightshift: Scaling the Energy Proportionality Wall through Server-level Heterogeneity," in *Microarchitecture (MICRO)*, 2012.

[123] Q. Wu *et al.*, "A Dynamic Compilation Framework for Controlling Microprocessor Energy and Performance," in *Microarchitecture (MICRO)*, 2005.

[124] R. E. Wunderlich, T. F. Wenisch, B. Falsafi, and J. C. Hoe, "SMARTS: Accelerating microarchitecture simulation via rigorous statistical sampling," in *Computer Architecture (ISCA)*, 2003.

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

[126] A. Yasin, Y. Ben-Asher, and A. Mendelson, "Deep-dive Analysis of the Data Analytics Workload in CloudSuite," in *Workload characterization (IISWC)*, 2014.

[127] M. T. Yourst, "PTLsim: A cycle accurate full system x86-64 microarchitectural simulator," in *Performance Analysis of Systems and Software (ISPASS)*, 2007.

[128] X. Zhan, R. Azimi, S. Kanev, D. Brooks, and S. Reda, "Carb: A c-state power management arbiter for latency-critical workloads," 2016.

[129] X. Zhang *et al.*, "Hardware Execution Throttling for Multi-core Resource Management," in *USENIX Annual Technical Conference (USENIX ATC)*, 2009.

[130] ——, "An evaluation of per-chip nonuniform frequency scaling on multicores," in *USENIX Annual Technical Conference (USENIX ATC)*, 2010.

[131] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "CPI²: CPU performance isolation for shared compute clusters," in *European Conference on Computer Systems (EuroSys)*, 2013.