AsmDB: Understanding and Mitigating Front-End Stalls in Warehouse-Scale Computers

Nayana Prasad Nagendra
Princeton University

Grant Ayers
Google

David I. August
Princeton University

Hyoun Kyu Cho and Svilen Kanev
Google

Christos Kozyrakis
Stanford University

Trivikram Krishnamurthy
Nvidia

Heiner Litz
University of California, Santa Cruz

Tipp Moseley and Parthasarathy Ranganathan
Google

***Abstract*—It is well known that the datacenters hosting today's cloud services waste a significant number of cycles on front-end stalls. However, prior work has provided little insights about the source of these front-end stalls and how to address them. This work analyzes the cause of instruction cache misses at a fleet-wide scale and proposes a new compiler-driven software code prefetching strategy to reduce instruction caches misses by 90%.**

Due to the continued growth of cloud-based digital services, warehouse-scale computers (WSC) are now serving billions of devices across the world. This massive growth necessitates improving the cost and efficiency of WSCs through microarchitectural and system software based optimizations.

WSC workloads are characterized by deep software stacks in which individual requests can traverse many layers of data retrieval, data

Digital Object Identifier 10.1109/MM.2020.2986212
Date of publication 16 April 2020; date of current version 22 May 2020.

processing, communication, logging, and monitoring. As a result, the instruction working set sizes of WSC workloads today are often 100× larger than server-class L1 instruction caches (i-cache)[1] and are currently expanding at rates of over 20% per year.[2] As cache sizes have not improved significantly over the last many years, WSC workloads are becoming increasingly front-end bound. Thus, processors are no longer able to sustain a high instruction fetch rate, manifesting itself in large unrealized performance gains due to front-end stalls, which are dominated by increased i-cache misses. While prior work has identified the growing importance of this problem, to date, there has been little analysis of the sources of these misses and of available opportunities to address them.

We corroborate this challenge for our WSCs on Google web search leaf servers, in which 13.8% of the total performance potential is wasted due to "front-end latency," principally caused by i-cache misses. We also measured L1 i-cache miss rates of 11 misses per kilo-instruction, and a hot steady-state instruction working set of approximately 4 MiB. This is significantly larger than the sizes of the L1 and L2 caches on today's server CPUs, but small and hot enough to easily fit and remain in the shared L3 cache (typically 10 s of MiB).[1]

To understand and improve the i-cache behavior of WSC applications, we focus on tools and techniques for *"broad" acceleration*[*] of thousands of WSC workloads. At the scale of a typical WSC server fleet, performance improvements of a few percentage points (and even sub-1% improvements) lead to millions of dollars in cost and energy savings, as long as they are widely applicable across workloads. To that end, our work provides three primary contributions: i) A methodology for analyzing instruction profiles at a fleet-wide scale; ii) detailed insights about code fragmentation and the perils of micro-optimization; and iii) a novel software-based code prefetch algorithm for reducing i-cache misses at fleet-wide scales.

[*] *"Deep" acceleration* would involve focusing on a handful of workloads and trying to recover most of the ≈ 15% performance opportunity.

AsmDB: A WSC ASSEMBLY DATABASE

To enable the necessary horizontal analysis and optimization across the server fleet, we built a continuously updated assembly database (AsmDB) to collect instruction- and basic-block-level information for most observed CPU cycles across the thousands of real production services executing across the Google fleet. AsmDB aggregates instruction and control-flow data collected from hundreds of thousands of machines each day and grows by multiple TiB each week. We have been continuously populating AsmDB over several years with the goal of providing easy-to-query assembly-level information for nearly every unique instruction executed in our WSCs. We demonstrate several cases where AsmDB proves invaluable for front-end optimization, including spotting opportunities for manual optimizations, finding areas for improvement in existing compiler passes, as well as for serving as a data source for a novel compiler-driven technique to improve i-cache hit rates.

To enable the necessary horizontal analysis and optimization across the server fleet, we built a continuously updated assembly database (AsmDB) to collect instruction- and basic-block-level information for most observed CPU cycles across the thousands of real production services executing across the Google fleet.

AsmDB is an always-on, massive-scale fleet-wide performance monitoring system. It uses hardware support to collect bursty execution traces, performs fleet-wide temporal and spatial sampling, and leverages sophisticated offline post-processing to construct full-program dynamic control-flow graphs. Collecting and processing profiling data from hundreds of thousands of machines is a daunting task by itself. However, we have carefully designed the system architecture such that it can capture and process profiling data in a cost-efficient way while still processing terabytes of data each week.

A fleet-wide assembly database, such as AsmDB, provides a scalable solution to search for performance antipatterns and opens up new

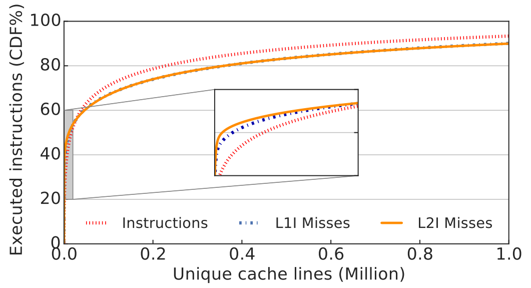

Figure 1. Fleet-wide distribution of executed instructions, and L1- and L2-instruction misses over unique cache lines. Like instructions, misses also follow a long tail.

opportunities for performance and total-cost-of-ownership optimizations. WSC servers typically execute thousands of unique applications, so the kernels that matter most across the fleet (the "datacenter tax"[2]) may not be significant for a single workload and are easy to overlook in application-by-application investigations. We leverage AsmDB's fleet-wide data in several case studies to understand and improve the i-cache utilization and IPC of WSC applications. We further correlate AsmDB with hardware performance counter profiles collected by a datacenter-wide profiling system—Google-wide profiling (GWP)[4]—to reason about specific patterns that affect front-end performance.

WSC APPLICATION ANALYSIS WITH AsmDB

WSC applications are well-known for their long instruction tails and flat execution profiles.[2] Figure 1 shows that i-cache misses in WSCs have a similar long tail. It plots the cumulative distribution of dynamic instructions, and L1-I and L2-I misses over unique i-cache lines over a week of execution, fleet wide. The zoomed-in view of the graph shows that the miss cumulative distribution function (CDF) initially has a more significant slope than the instruction CDF, suggesting that there exist some pointwise manual optimizations with high potential performance gains. However, the distribution of misses quickly tapers off. In particular, addressing just two-thirds of dynamic misses requires optimizations in $\approx 1M$ code locations, which is only conceivable leveraging automation. This points us toward exploring scalable, automated solutions—with compiler and/or hardware support and no developer intervention—to exploit these behaviors.

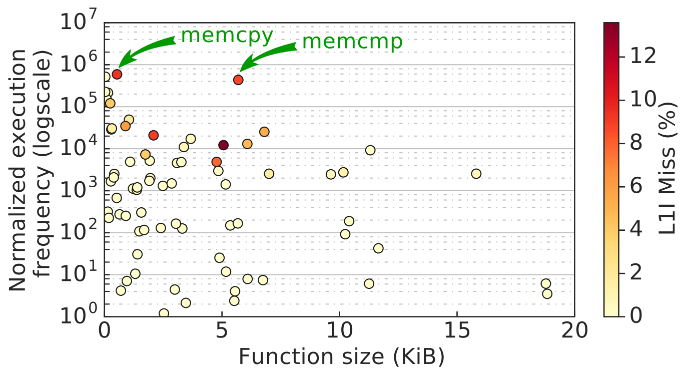

Figure 2. Normalized execution frequency versus function size for the top 100 hottest fleet-wide functions. memcmp is a clear outlier.

EFFECTS OF CODE FRAGMENTATION ON CACHES

Code bloat and unnecessary instruction complexity, especially in frequently-executed code, can lead to excessive i-cache pressure. We analyze code bloat in Figure 2, leveraging AsmDB-data—it plots the normalized function hotness (how often a particular function is called over a fixed period) versus the function's size in bytes for the 100 hottest functions in our WSCs. Perhaps unsurprisingly, it shows a loose negative correlation: Smaller functions are called more frequently. It also corroborates prior findings that low-level library functions ("datacenter tax"[2]), and specifically memcpy and memcmp, are among the hottest in our examined workloads.

However, despite smaller functions being significantly more frequent, they are not the major source of i-cache misses. Overlaying miss profiles from GWP onto Figure 2 (shading), we notice that most observed cache misses lie in functions larger than 1 KiB in code size, with over half in functions larger than 5 KiB. Most functions of 5 KiB or larger exhibit inlined call stacks of ten or more layers in depth.

While deep inlining is crucial for performance in workloads with flat callgraphs, it exponentially increases the amount of code loaded into the i-cache at each inline level, of which often only a small fraction is hot. Cold code brought

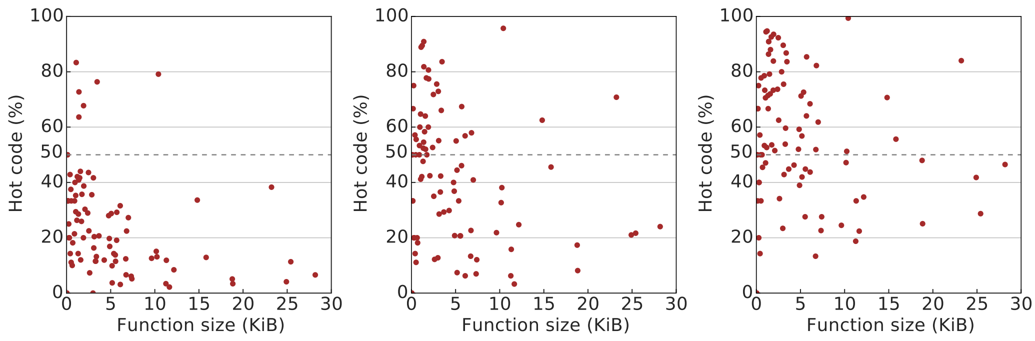

Figure 3. Fraction of hot code within a function among the 100 hottest fleet-wide functions. From the left-hand side to right-hand side, "hot code" defined as covering 90%, 99%, and 99.9% of execution.

into the cache, in addition to the necessary hot instructions leading to hot/cold fragmentation and thus suboptimal utilization of the limited cache resources.

We more formally define fragmentation to be the fraction of code (in bytes) that is necessary to cover the last 10%, 1%, or 0.1% of executions of a function. Because functions are sequentially laid out in memory, these cold bytes are very likely to be brought into the cache by next-line prefetchers. Intuitively, this definition measures the fraction of i-cache capacity potentially wasted by loading cold cache lines.

We find that intrafunction fragmentation is especially prevalent. Even after compiling with feedback-directed optimization, 50% of the codes in all functions are cold, frequently interleaved with hot code sections, and thus practically never executed despite being likely to be in the cache. This is true even among the hottest and most well-optimized functions in our server fleet.

Using AsmDB data, we calculate the measure of fragmentation for the top 100 functions by execution count in our server fleet. Figure 3 plots it against the containing function size. If we consider code covering the last 1% of execution as "cold," 66 functions out of the 100 are comprised of more than 50% cold code. Even with a stricter definition of cold (<0.1%), 46 functions have more than 50% cold code. Perhaps not surprisingly, there is a loose correlation with function size—larger (more complex) functions tend to have a larger fraction of cold code.

We attribute the intrafunction fragmentation to the deep inlining that the compiler needs to perform when optimizing typical WSC flat execution profiles. Hence, this suggests that combining inlining with more aggressive hot/cold code splitting can achieve better i-cache utilization, freeing up the scarce capacity.

On a finer granularity, we find that the individual cache lines are also often fragmented and waste cache capacity, especially for small functions. Unlike cold cache lines within a function, cold bytes in a cache line are always brought in along with the hot ones, introducing an even more significant performance issue. This suggests that there exist opportunities to improve the basic-block layout, at link or postlink time, when compiler profile information is precise enough to reason about specific cache lines.

We provide a concrete example of optimizing code bloat and fragmentation by focusing on memcmp, one of the hottest functions contributing to cache misses. memcmp clearly stands out of the correlation between call frequency and function size in Figure 2. It is both extremely frequent, and at almost 6 KiB of code, 10× larger than memcpy, which is conceptually of similar complexity. Examining its layout and execution patterns (see Figure 4) suggests that it does suffer from a high amount of fragmentation, as we observed fleet wide in the previous section. While covering 90% of executed instructions in memcmp only requires two cache lines, getting up to 99% coverage requiring 41 lines or 2.6 KiB of cache capacity. Not only is more than 50% of the code cold, it is also interspersed with hot regions, increasing the likelihood to be brought in by next-line prefetchers. Such code bloat is costly—

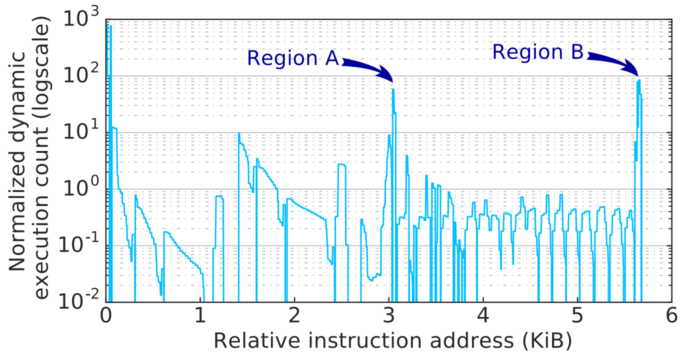

Figure 4. Instruction execution profile for memcmp. 90% of dynamic instructions are contained in 2 cache lines, covering 99% of instructions requiring 41 i-cache lines.

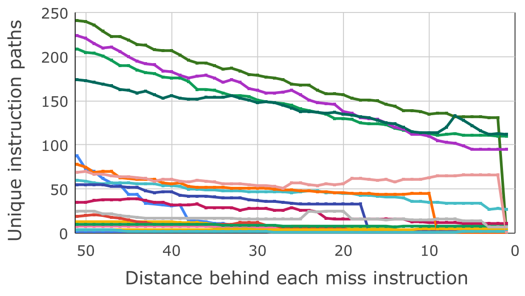

Figure 5. Fan-in for some misses can grow very fast with distance, especially for library functions.

performance counter data collected by GWP indicate that 8.2% of all i-cache misses among the 100 hottest functions are from memcmp alone.

While conceptually simple, our version of memcmp was highly optimized for microbenchmarks and contained many code paths for specific input variations. We show that in WSC environments where cache capacity is especially constrained, it is actually better to provide a *reduced* version of memcmp containing only a few paths and that doing so improves fleet-wide performance by up to 1%.

SOFTWARE PREFETCHING FOR CODE

Looking into the instructions that lead to i-cache misses, we find that, while not particularly concentrated in specific code regions, most i-cache misses still share common characteristics. Specifically, missing instructions are often the target of control-flow-changing instructions with large jump distances.[3] We find that distant branches and calls that are not amenable to traditional cache locality or next-line prefetching strategies account for a large fraction of cache misses among WSC applications.

For misses at the target of a distant jump, we propose and evaluate a profile-driven optimization technique that intelligently injects software prefetch instructions *for code* into the binary during compilation. We outline the design of the necessary "code prefetch" instruction, which is similar in nature to existing data prefetch instructions, except that it fetches into the L1 i-cache and utilizes the I-TLB instead of the D-TLB. The implementation of such an instruction has negligible hardware cost and complexity compared to pure hardware methods and is commercially viable today. While it can be implemented on top of a wide variety of hardware front-ends, we demonstrate its viability on a system that employs only a next-line instruction prefetcher.

Prefetching represents a prediction problem with a limited window of opportunity. Effective prefetches are both accurate and timely—they only bring in useful miss targets and do so neither too early nor too late in order to minimize early evictions and cache pollution. As a result, an effective prefetcher would have high overall miss coverage. Our prefetch insertion algorithm uses profile feedback information from AsmDB and performance counterprofiles to ensure timely prefetches with minimal overhead.

Some of the challenges that arise among software prefetching techniques include—*fan-in*, the number of potential paths leading to a miss increases as the prefetch injection site is moved backward from a missed target. Figure 5 shows the fan-in for the top 20 i-cache misses from a web search profile. In several cases, the number of paths leading in to a single miss exceeds 100 even with a lookback of only ten instructions. Our approach leverages profiling information to only insert helpful prefetches, increasing coverage and minimizing fan-in. *Fan-out* poses another challenge in finding the prefetch injection site as not all execution paths are likely to lead to the miss. We address this by pruning paths that exceed a maximum fan-out threshold. Furthermore, instruction prefetches themselves increase the code footprint and hence need to be inserted carefully.

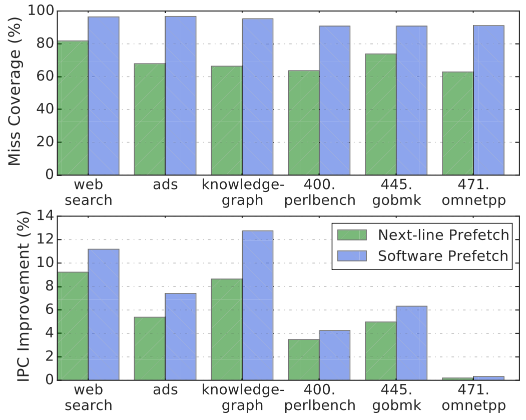

Figure 6. Miss coverage and performance improvement for the best-performing configuration for each workload.

At its core, our prefetch injection strategy leverages the observation that the injection site of a prefetch instruction can be freely moved within the window of opportunity to minimize fan-in and fan-out. We call this approach *dynamic window injection*. At a high level, our prefetch procedure first constructs the execution history for each miss and then traverses the control flow graph in the reverse direction until it reaches the end of the instruction window, calculated based on the application-level IPC. Next, prefetch injection sites are searched for each miss among each of its execution paths, which have minimal fan-in and fan-out. Prefetch instructions are then automatically inserted in the selected injection sites for the corresponding misses as part of the final linking steps.

We prototype the effects of our proposed software prefetching technique on memory traces from several WSC workloads. We evaluate on a modified version of the zsim simulator,[5] by using the system parameters modeled against an Intel Haswell datacenter-scale server processor. We focus primarily on three WSC applications—a web search leaf node, an ads matching service, and a knowledge graph back-end. For each workload, we collect traces during a representative single-machine load test, which sends realistic loads to the server under test.

Figure 6 shows that our prefetching technique is able to eliminate 91%–96% of all i-cache misses, with a performance improvement proportional to the front-end boundedness of the application and the gap left from NLP. In all cases, fewer than 2.5% of additional dynamic instructions are added for code prefetches.

LONG-TERM IMPLICATIONS

With increased technological growth, WSCs now serve billions of devices and applications across the planet. Due to their success, we expect an ever-greater reliance on WSCs in the near future, providing faster, more reliable, and more secure services to society. These increasing demands necessitate achieving higher performance for WSCs in order to be cost- and energy-efficient for WSC companies and their customers while simultaneously reducing the environmental impact on our world.

In combination with the slowdown of Moore's law, improving the efficiency of existing hardware in WSCs becomes even more critical. We analyzed a web search binary, showing that 68% of the CPU performance potential is lost due to pipeline stalls, of which 13.8% are due to the front-end not being able to deliver instructions fast enough.

This article addresses the front-end bottleneck on following fronts.

- First, we have built a tool that is capable of collecting data from live datacenter applications at the granularity of instructions and at the scale of a WSC. We have described the architecture design decisions in detail, enabling other WSC operators to reproduce our system.
- Second, this article is the first work that shows detailed characterization studies of the processor front-end at the scale of a WSC describing previously unreleased performance characteristics of WSC workloads.
- Third, we have proposed and evaluated a novel software-based code prefetch strategy to automatically and effectively reduce i-cache misses across large WSC workloads.

This work provides a powerful methodology to perform further at-scale research to obtain a detailed understanding of the microarchitectural characteristics and the interplay between current software and hardware. In addition, its reproducibility enables other WSC companies to perform similar research. Overall, such research would enable hardware vendors to work closely with software developers to better design future processors.

Our front-end characterization studies benefit the compiler and architecture communities both in academic and industrial settings. Our results on micro-optimizations, fragmentation, and code-bloat can help in fine-tuning compiler passes, optimizing inlining strategies, and basic block layouts. Similarly, our studies provide valuable information to architecture researchers exposing existing software loop holes that can be addressed with next-generation hardware designs.

We developed AsmDB, a database for instruction and basic-block information across thousands of WSC production binaries, to characterize i-cache miss-working sets and miss-causing instructions.

Our work on software code prefetching proves as a strong case study for hardware vendors to provide support for a software code prefetch instruction and to implement such an instruction in the instruction set architecture (ISA). With this, compiler writers and software developers can leverage code prefetching and its resulting performance improvements in an automatic and scalable way.

More broadly, this article provides two insights, which we believe will have a significant and long-lasting impact on future research in the performance optimization and computer architecture domain. The first insight teaches the importance of enabling fleet-wide performance optimizations, which we also refer to as the *Amdahl's law of WSC performance*. Traditionally, performance optimizations have been focused on individual applications. In this approach, applications are profiled to determine the most compute-intensive regions, resulting in the largest performance gains when optimized. However, this approach no longer applies to WSCs as datacenters run thousands of different applications simultaneously. As a result, compute-intensive application-specific kernels are no longer worth optimizing. Instead, performance engineers need to focus on code that is shared among many applications in the fleet, representing the largest aggregated percentage of compute cycles.

The second insight teaches the importance of designing domain-specific general-purpose processors. WSCs have grown to a size at which designing domain-specific accelerators becomes feasible and cost-efficient. However, while this approach has proven successful for domains such as deep learning, most of the fleet cycles are still executed on general-purpose processors as many applications are too complex and rapidly changing to render custom-designed hardware feasible. Nevertheless, as this article showed, the performance characteristics of WSC applications are fundamentally different from traditional applications such as the SPEC benchmark suite. WSC processors may differ with capabilities such as our proposed instruction prefetching mechanism, which may be of little use to SPEC applications, but which delivers significant performance gains for datacenter applications.

In summary, the evidence is strong that this article will promote the research and development of new compiler techniques, new processor designs, and new ways of collecting and analyzing behaviors at the warehouse scale.

CONCLUSION

This work focused on understanding and improving i-cache behavior, which is a critical performance constraint for WSC applications. We developed AsmDB, a database for instruction and basic-block information across thousands of WSC production binaries, to characterize i-cache miss-working sets and miss-causing instructions. We used these insights to motivate fine-grain layout optimizations to split hot and cold codes and better utilize limited i-cache capacity. We also proposed a new feedback-driven optimization that inserts software instructions for code prefetching based on the control-flow information and miss profiles in AsmDB. This prefetching optimization can cover up to 96% of i-cache misses without significant changes to the processor and while requiring only very simple front-end fetch mechanisms.

ACKNOWLEDGMENTS

This work was supported by the NSF Award CCF-1823559. Nayana Prasad Nagendra and Grant Ayers contributed equally to this work.

REFERENCES

1. G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, “Memory hierarchy for web search,” in *Proc. IEEE Int. Symp. High Perform. Comput. Archit.*, 2018, pp. 643–656.
2. S. Kanev *et al.*, “Profiling a warehouse-scale computer,” in *Proc. Int. Symp. Comput. Archit.*, 2015, pp. 158–169.
3. R. Kumar, B. Grot, and V. Nagarajan, “Blasting through the frontend bottleneck with shotgun,” in *Proc. Archit. Support Program. Lang. Oper. Syst.*, 2018, pp. 30–42.
4. 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*, vol. 30, no. 4, pp. 65–79, Jul./Aug. 2010.
5. D. Sanchez and C. Kozyrakis, “ZSim: Fast and accurate microarchitectural simulation of thousand-core systems,” in *Proc. Int. Symp. Comput. Archit.*, 2013, pp. 475–486.

Nayana Prasad Nagendra is currently working toward the Ph.D. degree with the Department of Computer Science, Princeton University. Her research interests include performance analysis and microarchitectural design with a focus on data centers. This work was done while she was an intern at Google. She is a student member of IEEE and ACM. Contact her at nagendra@cs.princeton.edu.

Grant Ayers is currently a Software Engineer at Google. His research interests include computer architecture, security, and accelerators. He joined Google after receiving the Ph.D. degree in computer science from Stanford University. This work was done while he was an intern at Google. Contact him at ayers@cs.stanford.edu.

David I. August is currently a Professor with the Department of Computer Science, Princeton University, where he directs the Liberty Research Group. His research interests include compilers and computer architectures. August received the Ph.D. degree in electrical and computer engineering from the University of Illinois at Urbana–Champaign. Contact him at august@princeton.edu.

Hyoun Kyu Cho is currently a Software Engineer at Google. His research interests include compiler optimization, parallel computing, and performance analysis. Cho received the Ph.D. degree in computer science and engineering from the University of Michigan at Ann Arbor. Contact him at netforce@google.com

Svilen Kanev is currently a Software Engineer at Google, working on translating datacenter performance analysis insights into performance and TCO gains. He is broadly interested in anything that straddles the hardware-software interface. Kanev received the Ph.D. degree in computer science from Harvard University. Contact him at skanev@google.com

Christos Kozyrakis is currently a Professor of electrical engineering and computer science with Stanford University. His research interests include hardware architectures and system software for cloud computing and emerging workloads. Kozyrakis received the Ph.D. degree in computer science from the University of California Berkeley. He is a Fellow of IEEE and ACM. Contact him at christos@cs.stanford.edu.

Trivikram Krishnamurthy is currently a Senior Engineering Manager at Nvidia. Before joining Nvidia, he was a Software Engineer at Google. Krishnamurthy received the M.S. degree in electrical and computer engineering from the University of California Santa Barbara. Contact him at trivikram.krishnamurthy@gmail.com.

Heiner Litz is currently an Assistant Professor in the Computer Science and Engineering Department, University of California, Santa Cruz (UCSC) and the Associate Director of the Center for Research in Storage Systems. His main research interests include computer architecture, operating systems, and storage with a focus on data centers. Before joining UCSC, he was a Researcher at Google. Litz received the Ph.D. degree from Mannheim University. He is a member of IEEE and ACM. Contact him at hlitz@ucsc.edu.

Tipp Moseley is currently a Principal Software Engineer at Google, where he works on datacenter-scale performance analysis. His research interests include compilers, operating systems, performance analysis, runtime systems, fault tolerance, and optimized lock-free data structures. Moseley received the Ph.D. degree in computer science from the University of Colorado at Boulder. Contact him at tipp@google.com.

Parthasarathy Ranganathan is currently a Distinguished Engineer at Google, where he is designing their next-generation systems. His research interests include systems architecture and management, power management, and energy efficiency for servers and datacenters. Ranganathan received the Ph.D. degree in computer engineering from Rice University. He is a Fellow of IEEE and ACM. Contact him at partha.ranganathan@google.com.