

A Characterization of Google Workload Traces

Matthew Giordano*, Akanksha Jain†, Enrico Deiana†, Derek Bruening†, Abhinav Sharma†, Ibrahim Hur†, Parthasarathy Ranganathan†, Thomas Anderson*, and Baris Kasikci*

*University of Washington

†Google

mgjordan@cs.washington.edu

Abstract—In this work, we perform an in-depth characterization of the microarchitectural performance of modern datacenter workloads. Google recently released hundreds of gigabytes of trace data from production, offering a unique view into current-day warehouse-scale workload behavior. Through simulating 359 cores spanning 12 workloads, we make several observations that depart from prevailing assumptions about modern workloads. For instance, while datacenter workloads continue to face frontend pressure, the backend and memory hierarchy dominate CPU stalls due to an order of magnitude higher instruction and data pressure on the memory subsystem than prior traces show. Working set sizes are 7x larger than prior server traces (CloudSuite and Qualcomm), and regular context switches pollute the microarchitectural state, rendering caches and state-of-the-art prefetchers ineffective. The frontend suffers from a 5x larger footprint, causing significant instruction spillover into unified caches. Consequently, we observe that modern datacenter workloads exhibit high DRAM latency sensitivity due to cascading cache misses, and high page size sensitivity due to their large footprint.

We contribute an open-source tool for converting the Google workload traces to a format usable in modern simulators, a characterization of these traces, and methodology guidelines for future research done on these traces.

1. Introduction

Cloud computing platforms underpin modern digital infrastructure, with web services, real-time analytics, and massive data backends relying on warehouse-scale computers to achieve incremental scalability. However, production cloud applications fundamentally exhibit execution behaviors that diverge from traditional desktop and high-performance computing workloads. Modern datacenter workloads operate over massive code and data footprints [3]–[5], [14], [22], [26], [33], [34], [44], [45], [47], [53], [61], introducing microarchitectural bottlenecks that degrade processor efficiency. At warehouse scale, mitigating even a fraction of these processor stalls translates into massive capital, operational, and energy savings. Consequently, identifying the microarchitectural implications of proprietary server-class workloads is critical for informing future core designs.

While numerous open-source benchmark suites [6], [9], [10], [22], [23], [34], [35], [50], [52], [54]–[56], [58], [59]

model microservices, serverless functions, and diverse cloud components, they often fail to capture the complex runtime dynamics of (often proprietary) production workloads [57]. This creates a disconnect: researchers design state-of-the-art microarchitectural optimizations using publicly available benchmarks, but these suites do not faithfully represent the realities of modern cloud workloads. The recently released Google Workload Traces (Version 2) [1] provides a rare bridge across this divide. These traces comprise of 842GB of production trace information across 12 different anonymized workloads from Google’s production environment, offering a unique opportunity to conduct microarchitecture investigations of modern server class applications.

In this work, we characterize the microarchitectural characteristics of Google workload traces. We first build and open source a trace conversion tool and provide methodological guidelines on how to simulate these traces. In particular, we identify two key challenges in simulating the Google traces: (1) the traces are collected for individual threads, whereas microarchitectural simulators require scheduled core-level views, and (2) Google’s recommended methodology—performing *online* thread scheduling while simulating up to 89 cores simultaneously—is computationally intractable on standard cycle-accurate simulators. To overcome these challenges, we develop a trace conversion framework built on top of DynamoRIO which uses offline scheduling to generate deterministic, core-levels traces for ChampSim [24]. This converter tool has already seen impact, providing traces for the 4th Data Prefetching Championship [19] competition.

Using this framework, we perform an in-depth microarchitectural characterization of the Google workloads. We demonstrate that these traces operate at remarkably low instruction-per-cycle (IPC), ranging from 0.5 to 1.0, exposing a performance gap between existing academic benchmarks and real-world execution. Our evaluation reveals the following key insights:

- While recent architectural research has targeted frontend instruction delivery [2], [4], [13], [25], [26], [29], [34], [36], [38], [39], [42], [53], [61], we find that the backend memory subsystem poses a far more critical bottleneck, presenting significant optimization opportunities.
- The Google workloads exhibit 7 times larger data and 5 times larger instruction working set sizes compared

to prior server benchmark suites (CloudSuite [22] and Qualcomm [21]), with significantly lower reuse than these suites. This footprint exceeds standard hardware cache capacities, leading to high compulsory miss frequencies, instruction fetch stalls, and high sensitivity to DRAM latency.

- State-of-the-art data prefetchers achieve poor accuracy and coverage, failing to mitigate cascading memory stalls.
- Data caches suffer from severe capacity contention driven by instruction spillover and speculative prefetches. In the L2 cache, instructions occupy 19% of the lines and prefetches consume 44%, leaving only 38% for actual data.
- Frequent context switches wipe out warm cache microarchitectural state, moving the bottleneck closer to the backend. High static instruction sharing among co-scheduled threads allows execution contexts to cooperatively warm some shared instruction structures, outweighing the ill effects of data cache thrashing.

Broadly, these observations individually align with prior characterizations of datacenter workloads [3], [22], [23], [57]. However, with our simulator analysis, we reveal a tight interplay between different microarchitecture components, and demonstrate the microarchitectural consequences of frequent context switching.

In summary, this paper makes the following contributions:

- We conduct a thorough microarchitectural bottleneck analysis across 359 cores traced at Google, quantifying bottleneck causes within the memory hierarchy and backend execution units.
- We study the effects of oversubscription and context switching in modern server cores.
- We outline data-driven directions for future warehouse-scale microarchitectural research, emphasizing memory-hierarchy redesigns and hardware-software co-design for data placement and context switching.
- We introduce and open-source a tool that translates Google traces into standardized, ChampSim-compatible core traces while establishing reproducible methodological guidelines for the community.

2. Methodology

2.1. Background

The Google Workload Traces (Version 2) [1] are a set of publicly available traces from Google’s data centers. Intended to enable computer architects to target designs for the unique characteristics of warehouse-scale workloads, the traces contain anonymized opcode and register information from production.

In total, Google supplies 13,060 thread-level traces across 12 workloads. Each workload also comes with a "peak live core count," the peak number of cores that Google observes

TABLE 1: The Google workload traces. Each workload features some number of thread-level traces and a "peak live core count". These traces sum up to some billion instructions, of which we simulate 500M/core, reporting the total simulation time. The context switches per million instructions (CSPMI) post-scheduling is also shown.

Workload	# Thread Traces	Peak Cores	Instructions (Billions)	Simtime (CPU hrs)	CS PMI
arizona	128	6	47.9	12.2	1.7
bravo.a	1680	3	45.7	15.5	11.9
charlie	2863	22	13.1	73.6	4.0
delta	236	4	2.8	16.4	5.6
merced	1073	89	70.1	298.3	2.4
sierra.a.3	1303	17	35.2	65.2	5.0
sierra.a.4	1597	24	34.2	117.7	6.0
sierra.a.6	1112	18	31.1	73.9	6.6
tahoe	626	40	59.7	143.1	2.8
tango	1206	51	45.7	177.9	4.4
whiskey	801	2	3.5	7.5	8.7
yankee	435	83	63.4	272.6	1.2

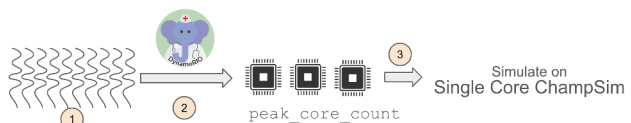


Figure 1: Conversion methodology overview per workload. Google thread-level traces (1) are provided. Our tool uses the DynamoRIO trace scheduler (2) to schedule onto the peak live number of cores offline. For each core (3), we simulate it individually using ChampSim, then aggregate metrics across cores.

to be actively running this workload during tracing. Table 1 presents the thread and core count per workload.

Each Google workload trace captures the execution of an individual thread. Correct execution on a microarchitectural simulator requires scheduling these threads onto cores, requiring an interleaving and core assignment. Figure 1 shows this process. Google advises using their DynamoRIO trace scheduler [12] to schedule the thread-level traces onto cores. The DynamoRIO scheduler provides a loose attempt at remaining accurate to the original program (e.g. switching threads at appropriate times, synchronizing multiple cores) using trace metadata. We build our trace converter on top of the DynamoRIO scheduler by leveraging its API, which yields the next instruction per core.

Google makes three recommendations [1] for creating core level traces using DynamoRIO:

- 1) Schedule onto the peak live number of cores.
- 2) Perform scheduling online, so that the scheduler can put each context switch at the right moment in simulated time.
- 3) Simulate all cores.

However, these are difficult to achieve at the same time, demanding the simultaneous simulation of all cores. Most open source simulators scale sublinearly with the number of cores [11], [24], [41], and in practice simulating more than 4-8 cores is prohibitive. With the high peak live core counts

present in Google workload traces (e.g. 89 in merced), the combination of the three points above proves infeasible.

2.2. Google Trace Conversion Methodology

In order to simulate the Google Workload Traces, we build and open-source a trace converter which creates core-level ChampSim Google workload traces using the following methodology:

- 1) We perform scheduling *offline*¹ onto the peak live core count to make tractable simulating every core.
- 2) We simulate cores individually².

We considered reducing simulation burden via the following approaches, but we find that they significantly alter the behavior under study.

Scheduling onto fewer cores. We find a high degree of sensitivity to the exact core count used for thread scheduling, due to different sharing behaviors and context switch sensitivity.

Sampling cores We find that sampling cores is methodologically dubious. After simulation, cores on a workload show a 7%-20% performance deviation. This data is visualized later in Figure 3. This means that individual cores are not representative of the broader dataset.

Together, these inadmissible simplifications indicate that (1) scheduling changes performance behavior and (2) there is diversity among threads. Figure 2 shows a visualization of threads running on cores over instructions, with instructions per cycle (IPC) overlaid in black. Visually, it validates our claims for the tahoe workload; across all 12 workloads, we see high amounts of variance per core, both at the thread and IPC level. We also see performance changes across some, but not all, context switches.

Observationally, we also see a high sensitivity (over 10%) in IPC on the Google workloads to the number of instructions simulated. Figure 2 confirms that the IPC is not stable, thus simulating more instructions will lead to different performance characteristics. We use a long simulation length per core (500M), but there is still roughly 10% variance in a few select cores between 500M and 1B instructions simulated.

2.3. Conversion Tool

Our Google trace conversion tool works by spawning one thread per simulated core. Each thread queries DynamoRIO for the next record belonging to that core in parallel. We then filter for instruction records, which give information about IP, source, and destination registers; and for memory records, which give information about source and destination memory addresses. Together, we combine

1. When scheduling offline, DynamoRIO assumes a fixed 0.5 instructions per cycle (IPC) for determining when it should perform a preemptive context switch to remain faithful to the original traced execution. Time quantum expiration causes ~12% of all context switches, and since we simulate cores individually, cross-core timings do not matter.

2. ChampSim cannot model intraprocess data sharing, only contention.

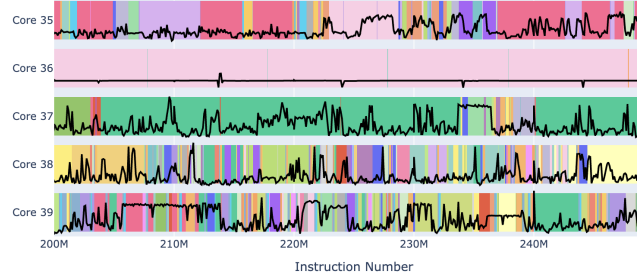


Figure 2: Visualization of various cores in the tahoe workload. Each color region is a thread, and the black line shows IPC, normalized to a 0-3 range. Some cores run long contexts, some short. Some context switches impact IPC, but IPC is highly variable even within threads.

adjacent records to construct ChampSim instructions. This creates 359 core level traces, and our methodology simulates all of them, treating core traces belonging to the same workload like equally weighted SimPoints [43]. As the Google traces contain anonymized register dependency information³, we do not know the program pointer, stack pointer, or flags register ordinals. ChampSim requires this information to identify branch types instead of having a dedicated branch type field. Hence, we must inject that information into branch instructions to properly have the simulator identify the type of branch.

2.4. ChampSim

We choose ChampSim [24] a trace-based microarchitecture simulator, as our conversion target. Originally designed for branch prediction, cache replacement, and prefetching competitions, it has grown in adoption [7], [8], [15]–[21], [26], [28], [30], [34], [37], [40], [46], [49], [51], [53] due to simplicity and speed of simulation. ChampSim models a simple, multi-core out of order processor, featuring a full cache hierarchy and DRAM simulation.

Our ChampSim configuration is shown in Table 2. It features next-generation cache sizes and modern microarchitecture components. We use the 2026 release of ChampSim, which has made significant improvements over the last few releases, and urge future work to do the same.

2.5. Validation

We validate our conversion from DynamoRIO to ChampSim in two ways. First, we compare the data and instruction footprint of each trace, and find the conversion process retains the exact same footprint. Second, we use Google’s DynamoRIO Cache Simulator to simulate the unconverted thread-level traces (which schedules them), and compared that versus our own offline-scheduled ChampSim simulation results. Our conversion/simulation agrees with

3. Each register is remapped to a virtual ISA in a many-to-one fashion. The Google trace register anonymization is consistent across all workloads. Google’s anonymization does not change memory addresses.

TABLE 2: ChampSim configuration. For full configuration details, see our repository.

Component	Configuration
Core	4GHz frequency, 5 wide retire 64KB TAGE-SC-L [48] branch predictor 8192-entry 8-way BTB 4096-entry IBTB, 64-entry RAS 256-entry 8-way DIB
L1I	32KB, 8-way, 4 cycle latency 8 MSHRs, 64/64/32 read/write/prefetch queue size DRRIP [32] replacement, EIP [46] prefetcher
L1D	48KB, 12-way, 5 cycle latency 16 MSHRs, 64/64/8 read/write/prefetch queue size DRRIP replacement, Berti [19], [40] prefetcher
L2C (non-inclusive)	2MB, 16-way, 10 cycle latency 32 MSHRs, 32/32/16 read/write/prefetch queue size DRRIP replacement, Pythia [8], [19] prefetcher
LLC (non-inclusive)	4MB / 1 core, 16-way, 20 cycle latency 64 MSHRs, 32 read/write/prefetch queue size DRRIP replacement, no prefetcher
DRAM	800MHz I/O frequency, 32ms refresh interval 1 channel, 1 rank, 8 banks, 64GB total 6/6/6/13 tCAS/tRCD/tRP/tRAS, ~12.5GBps/core
ITLB, DTLB	64 entries, 4-way, 1 cycle latency, 8 MSHRs
STLB	1536 entries, 12-way, 8 cycle latency, 16 MSHRs
VMem	2MB page size

DynamoRIO’s Cache Simulator to within a mean absolute difference of 2.7% in cache MPKI (L1D/L2/LLC), max 7.9%.

Our conversion has two primary limitations. First, some Google trace instructions have more than 4 register arguments or more than 2 memory operands, which is not supported by ChampSim. These operands are dropped, but those instructions account for a minuscule fraction of converted instructions ($<0.0001\%$). Secondly, our offline trace conversion is nondeterministic because multiple simulated cores race for their next record with many simulated ready threads. However, when aggregated over the entire workload, the performance deviation is small when rescheduling (all workloads stay within 3% IPC except whiskey/arizona which show 7% IPC deviation), and in practice, we only need to schedule once.

2.6. Benchmarks

To provide a helpful comparison, we evaluate the Google workload traces side-by-side with several other benchmark suite traces. We choose these benchmark suites because they represent an extrema of behavior (e.g., GAP) or are classic benchmarks (e.g., SPEC). Each suite consists of several workloads, which consist of various traces at different points in the workload or on different cores. Table 3 provides an overview of the various benchmarks suites, workloads, and traces.

SPEC. The SPEC06 [55] and SPEC17 [56] integer and floating point workloads are simulated according to the SimPoint [43] methodology used in DPC3 [18].

Graph. The GAP [6] suite is included. We source GAP traces from the ML Prefetching Competition at ISCA 2021 [37].

TABLE 3: Overview of all simulated benchmark suites. Each region of interest (ROI) has a warmup time of 50M instructions (not included).

Suite	Workloads	Traces	ROI Instructions
SPEC	49	189	250M
GAP	5	20	250M
CloudSuite	5	120	250M
Qualcomm	2	43	50M
Google	12	359	500M

Server Workloads. The CloudSuite [22] and Qualcomm [21] traces provide rare insight into server applications, with Qualcomm providing production traces. These traces are sourced from CRC2 [16] and IPC1 [28], respectively, dating back to 2017 and 2020. Unfortunately, ChampSim misidentifies CloudSuite branch types. We exclude CloudSuite from branch-related studies.

Google. Along with this paper, we release a usable set of 359 ChampSim traces for the 12 Google workloads, and the conversion tool that was used to create them.

3. Google Workload Analysis

3.1. Key Results

The Google workload traces consist of 12 workloads, which contain 359 cores scheduled from 13,060 threads. We simulate all 359 cores for 500M instructions after a 50M instruction warmup period, and visualize the Google performance results using one point to represent one of these 359 core simulations. Figure 3 shows the IPC (instructions per cycle) for all the Google workloads, following the ChampSim methodology presented in Section 2. As a point of comparison, Figure 4 shows the IPC at a workload level for every simulated benchmark suite. We observe that Google workloads exhibit a low IPC in the range of 0.5-1 compared to other suites.

The Google workloads all exhibit key interacting characteristics that contribute to their poor performance:

- Memory is the primary performance bottleneck due to a large data footprint that existing microarchitecture optimizations (e.g. caches, prefetchers) fail to mitigate, partially due to microarchitectural interference from frequent context switches.
- L1-instruction (L1I) caches do not capture the instruction working set of any workload, leading to significant instruction spillover in the L2, causing data evictions that exacerbate backend issues.

3.2. Bottleneck Analysis

To provide a rough estimate of the bottlenecks in the Google workloads, we approximate the potential speedup of each component. Namely, we model idealized components with perfect prediction for the branch predictor (BP) and

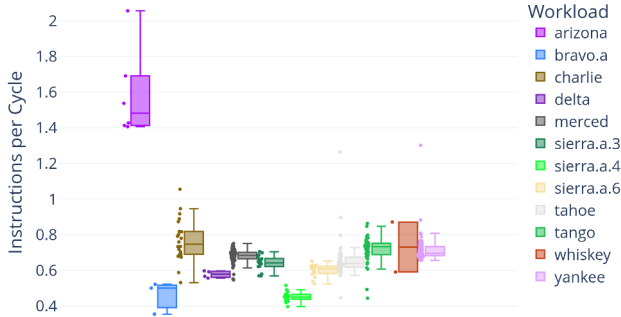


Figure 3: IPC distribution across Google workload traces. Each dot represents one core, and the computed distribution (median, upper/lower quartile, fences) sits alongside it. Cores show a slow IPC.

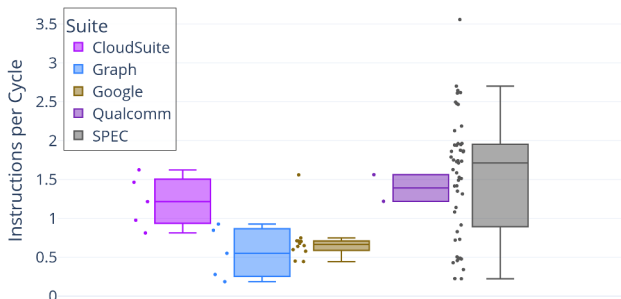


Figure 4: IPC distribution across all workloads. Each dot represents one workload, which is the aggregation of several core simulations. Google workloads perform near the bottom of all workload suites.

branch target buffer (BTB), unlimited size for caches⁴, or zero latency for DRAM (after navigating the standard cache hierarchy). We show these results in Figure 5. A perfect branch unit provides a geomean 8% speedup on Google workloads, an unlimited L1-instruction cache (L1I) 13% speedup, and an unlimited L1-data cache (L1D) 73% speedup. Instant DRAM provides a substantial benefit over unlimited caches, with a 136% speedup. While these are all unrealistic configurations, they provide a first order estimate for where the performance opportunities lie.

Zooming in on the frontend, Figure 6 breaks down the performance potential by frontend component. The branch predictor, branch target buffer, and instruction fetch latency all offer an opportunity for performance improvement in the Google workloads, and the Google workloads uniquely benefit from all three. Additionally, Google workloads are more instruction fetch latency sensitive than any other workload, with a potential 11% geomean IPC gain.

Together, Figures 5 and 6 point to a few interesting insights: (1) We find that the perfect branch predictor and

4. The largest workload accesses 2GB of unique cachelines. Unlimited caches still incur compulsory misses and the resulting DRAM penalty once per cacheline.

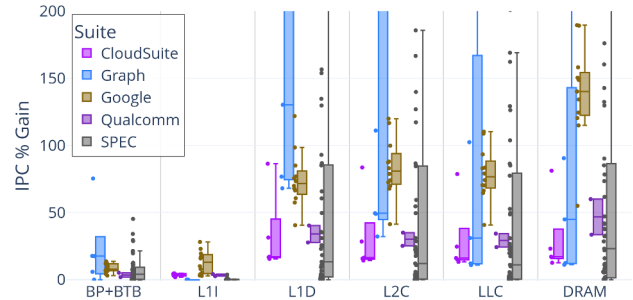


Figure 5: Perfect component IPC speedup. A perfect branch unit results in oracle predictions, a perfect cache means an unlimited cache size, and perfect DRAM means zero latency after an LLC miss. Google traces have more potential for hardware optimization than SPEC, and the backend in particular shows high performance possibility. Instant DRAM, in particular, is even more effective than an unlimited L2C cache.

branch target buffer only yield a modest 8% speedup on Google workloads, even with the second highest median branch MPKI of 7 (compared to SPEC’s 1.4). This tell us that many mispredictions do not hurt performance, as the CPU is stalled on memory. Thus, the branch predictor does not offer as much performance improvement as other components in the microarchitecture. (2) Although the frontend (and in particular, the L1I cache) has considerable opportunity for improvement, the backend and memory hierarchy of these workloads offer more opportunity for performance gain. (3) Google workloads are highly DRAM latency and throughput sensitive. Compared to perfect caches, instant DRAM provides significantly more performance for Google workloads (33% improvement), whereas for SPEC, perfect caches achieve most of the performance headroom of perfect DRAM (8% improvement). This highlights a significant uptick in compulsory misses. Furthermore, while perfect caches improve performance on only a few SPEC workloads, all Google workloads are affected, even though they represent a wide variety of workloads in production. This suggests that modern cloud workloads feature complex data access patterns through large footprints and many compulsory misses, a departure from traditional workloads.

3.3. Data Cache Performance

The caches on Google workloads tend to perform poorly, leading to backend bottlenecks. Figure 7 shows the cache demand misses per kilo instructions (MPKI) across the L1D, L2C, and LLC with prefetchers enabled. We observe poorly performing caches for Google workloads relative to other server workload suites, with high MPKI and low hit rates. Google workload caches have a similar median hit rate to CloudSuite at each level, but a higher frequency of accesses causes a higher overall MPKI.

We identify three key reasons that the data caches are not able to perform well: (1) a supersized data footprint and working set size, high reuse distance, and frequent

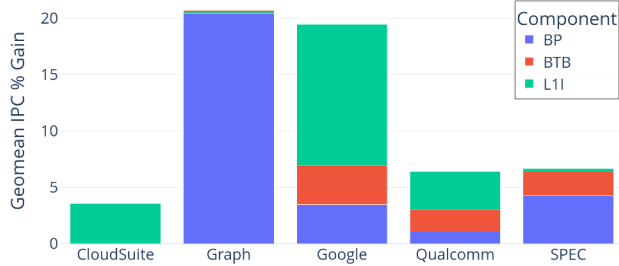


Figure 6: IPC gain from "perfect" frontend components, by part. A perfect BP and BTB refer to predicting the branch direction and target correctly, whereas a perfect L1I refers to an unlimited size. Google's frontend can benefit significantly across all three components. CloudSuite branch headroom is omitted.

TABLE 4: Data working set size per core. The working set size (WSS) is determined by varying the L1D size to unrealistic quantities until performance plateaus. Google features the highest data WSS at 368MB per core, making it impossible to contain in existing hardware caches.

Suite	Mean Data WSS (MB)	Median Data WSS (MB)
CloudSuite	44	48
Google	368	384
Graph	192	192
Qualcomm	54	54
SPEC	127	24

compulsory misses make caching inherently more difficult; (2) instruction spillover and poor prefetching accuracy lead to L2C and LLC pollution; and (3) thread context switches lead to insufficient cache warmup time per thread and wipe out cache predictive state, pushing the bottleneck more towards the backend.

3.3.1. Data Footprint. To investigate why the caches perform poorly, we examine the reuse distance (the number of instructions between accesses to the same data cacheline) and the frequency of compulsory misses in Figures 8 and 9. The Google workloads exhibit a long average reuse distance (median 500k instructions), indicating that caching is challenging. Additionally, the median Google workload experiences a compulsory cache miss every 162 instructions on average (second only to Graph workloads), leading to further pressure on the DRAM. We observe that in a 100M instruction sample of each workload, the Google workloads have the largest data footprint at 41MB; the next largest non-graph workload accesses only 14MB of data.

Because of the long reuse distance, frequent compulsory misses, and large overall footprint, Google features the largest median working set size of any server workload suite by 7 times. This is detailed in Table 4. This several hundred megabytes working set size outsizes any hardware cache⁵ by an order of magnitude, making caching substantially more difficult than in other benchmarks.

5. The last level cache can be hundreds of megabytes, but is shared by hundreds of cores.

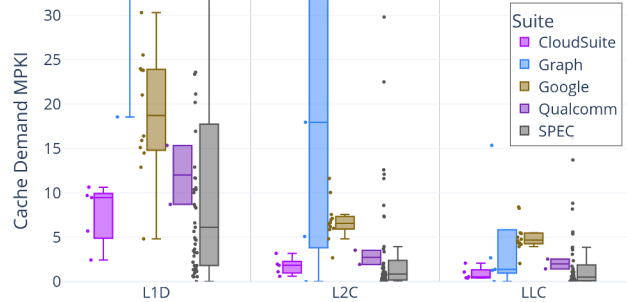


Figure 7: Backend cache demand MPKI. Demand accesses are defined as reads and writes for the L1, and reads and RFOs in all other caches. Prefetching is enabled. Google workloads exhibit strikingly high MPKIs across all caches on all workloads except arizona. Graph workloads are off the chart.

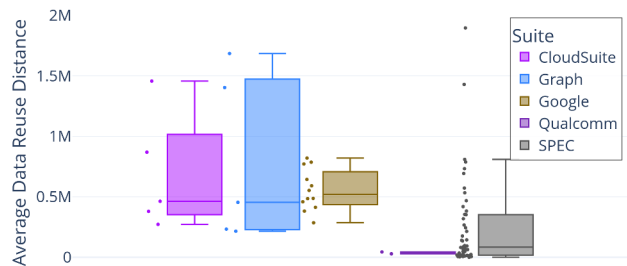


Figure 8: Backend average reuse distance, in instructions. Google's median reuse distance is the highest of any workload suite.

3.3.2. L2C, LLC Sensitivity, Instruction Spillover, and Prefetchers.

Figure 10 showcases the core's performance sensitivity to the L2C capacity within the 0.5MB to 8MB range. When the L2C is 2MB or smaller, Google's performance degrades slightly, down to a geomean 1% performance slowdown at 1MB. However, a larger L2C (4MB and above) benefits the Google workloads more, with an 8MB yielding a geomean 16% speedup.

We see similar results for scaling the LLC. Smaller LLCs do not hurt performance by much; going above the simulated 4MB LLC per core yields performance gains.

To explain cache sensitivity, we look at just the Google workloads' working set sizes in Figure 11. We observe that most of the Google workloads' working set sizes begin at 3MB and end between 192MB and 768MB per core. Thus, increasing the cache sizes beyond 2MB will start to capture the working set⁶. However, increased cache sizes also bring increased latency and area [27], which may mitigate performance benefits and reduce the number of cores on the chip. We therefore cannot conclude that area should be spent on caches.

As a consequence of cache sensitivity, we must keep useful data in the L2C and LLC. Unfortunately, we identify

6. Shared caches, that are hundreds of MB, will cause significant performance variation in the Google workloads, as workloads occasionally acquire more than 2MB of cache space.

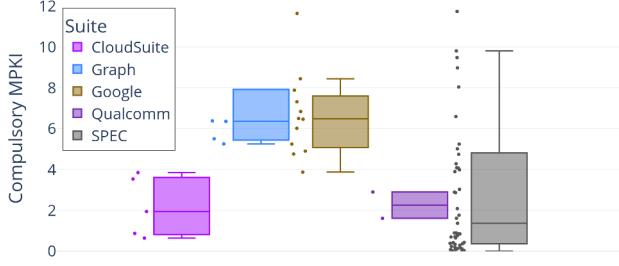


Figure 9: Compulsory misses per kilo instruction. The server workloads experience significantly more compulsory misses than traditional benchmarks, and Google experiences 3x more compulsory misses than the next server workload to put it on par with Graph workloads. Additionally, the 6 compulsory MPKI account for 40% of the LLC total MPKI.

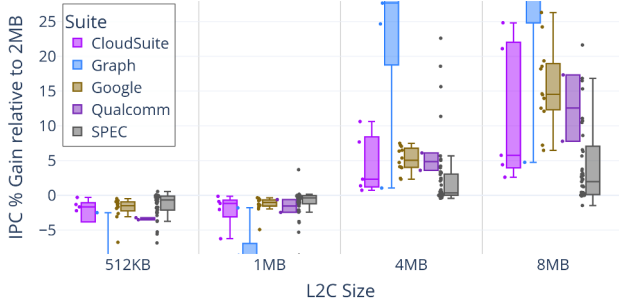


Figure 10: L2C Sensitivity. Even small increases in the L2C size result in significant performance improvements, but going below 2MB shows little drop off.

that prefetchers and instruction spillover cause significant cache pollution, decreasing the effective L2C size by half.

We quantify the percentage of instruction and prefetch cachelines in the L2C in Figure 12. We do this by capturing the cache occupancy every 100K instructions and averaging the series. The Google traces feature a large instruction footprint, studied in Section 3.4.1. As a consequence, many instruction cachelines end up spilling from the L1I into the unified L2C and LLC. Namely, the L2C consists of 44% prefetches and 19% instructions, leaving just 38% of capacity for data. A high percentage of prefetches in the cache is not necessarily a bad thing, as long as those prefetches are eventually used (denoted as accurate prefetches). Ideally, our prefetches also mitigate a large amount of cache misses (denoted as coverage).

Unfortunately, we find that state-of-the-art prefetchers (Berti [40] and Pythia [8]) are not able to effectively reduce cache misses and keep memory traffic low. Figure 13 showcases the IPC difference when enabling Berti at the L1D and Pythia at the L2C. On the Google workloads, the Berti’s IPC gain is a geomean of 7%, the lowest of all suites. Berti suffers from a poor accuracy and coverage (median 53% and 14%, the second lowest next to GAP). Pythia causes the IPC to increase by just 1% compared to just having Berti for Google workloads. The L2C total MPKI goes up

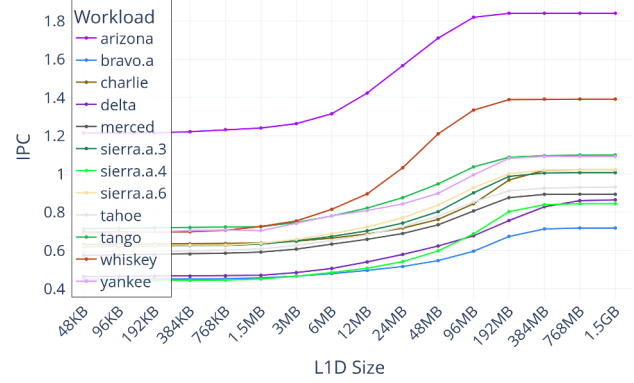


Figure 11: Google data working set size analysis via L1D size ablation. Most workloads only see substantial IPC gain starting with a 3MB cache per core, and stop scaling between 192MB and 768MB per core.

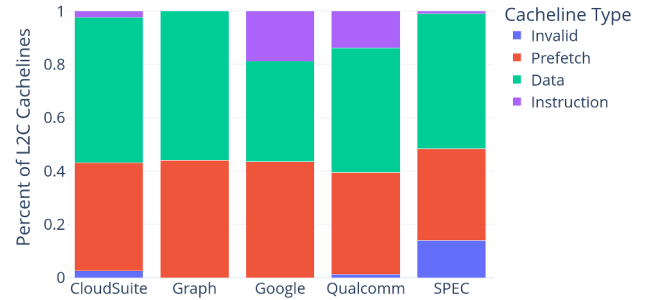


Figure 12: L2C average occupancy. Prefetches denote lines prefetched and yet to be hit. The Google traces feature the unique combination of large prefetch occupancy and large instruction occupancy in the L2C. Of the prefetches in Google’s L2C, 24% are issued by EIP, 29% are issued by Berti, and 47% are issued by Pythia.

by over double on 9 Google workloads due to the poor accuracy of 40% and coverage of 37% (the second lowest next to Qualcomm, and tied with GAP), causing significantly higher DRAM traffic.

Our complex prefetchers are unable to provide much IPC gain. Although prior work [31] identifies low accuracy as a concern, we also identify a low coverage on the Google workload traces, meaning that more L2C space should be reserved for data. Given the high number of first misses in the Google workloads, it is disappointing that current prefetchers cannot mitigate more cache misses. We identify frequent context switches as a key reason for prefetcher ineffectiveness below.

3.3.3. Context Switch Impacts. The Google workload traces are scheduled from threads onto cores. Given this unique characteristic, we study the impact of context switches on microarchitecture, showing that cache performance worsens with the introduction of context switches. Counterintuitively, multithreading causes an overall mild

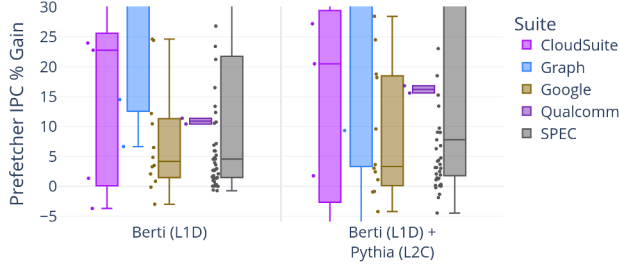


Figure 13: Berti and Pythia IPC % Gain at the L1D and L2C respectively. Berti modestly helps performance on the Google workload traces, while Pythia modestly helps or hurts performance.

performance enhancement due to BTB sharing, but caches and prefetchers suffer.

Figure 14 shows how various statistics change when multithreading is enabled (the default). To study this, we normalize the standard multithreaded results to a version where each thread gets a dedicated core. Each dedicated core runs the same instructions in warmup and in simulation that are run by that thread in the scheduled, multithreaded trace. We accumulate each core and weigh by its instruction count. Notably, we see that all backend cache MPKIs are significantly increased by the introduction of multithreading: the L1I, L1D, L2C, and LLC suffer from a 42%, 54%, 26%, and 25% median demand MPKI increase when running in a multithreaded environment. Furthermore, prefetchers lose accuracy and coverage, with Berti suffering from a 41% coverage loss. Only branch prediction improves with multithreading, with a 8% lower median BP MPKI and 47% lower BTB MPKI. The IPC is overall improved by 18%, suggesting that multithreading moves the bottleneck away from the frontend and towards the backend.

To understand why MPKI suffers, we examine the instruction-weighted distribution of context lengths in Table 5, where a context refers to the time spent between thread switches, voluntary or involuntary. The Google workloads tend to execute in contexts of length 500K-10M. 10M instructions per context is not generally accepted as long enough to fully warm up microarchitecture [8], [18], [19], [40].

Table 6 shows the amount of thread sharing on each workload. Notably, most data accesses by most threads are to private cachelines, and most instruction accesses are to shared cachelines (>1 accessing thread). This implies that many instructions are shared between threads, but threads operate on different data, meaning that there is little data cache reuse between threads.

Given the short contexts and low degree of sharing, we attribute the higher data cache MPKI in multithreaded cores to cold microarchitecture.

3.3.4. Consequences. DRAM Sensitivity. As a consequence of high miss rates in each cache level, the next level cache experiences more accesses and thus more misses, compounding to an overall high sensitivity to DRAM. We

TABLE 5: The instruction-weighted distribution of various thread context lengths. The maximum time quantum is 10M instructions, or 5ms with 0.5 IPC, but a context length can exceed this if the same thread is scheduled twice in a row. Half of the Google workloads spend their median cycle in a context 500K-1M instructions in length, while half spend it in a context 1M-10M instructions in length.

Workload	p1	p5	p25	p50	p75	p95	p99
arizona	46K	5M	10M	10M	10M	10M	33M
bravo.a	4K	31K	162K	585K	6M	10M	10M
charlie	24K	619K	6M	10M	10M	10M	10M
delta	17K	35K	1M	4M	7M	10M	10M
merced	65K	209K	680K	1M	5M	10M	110M
sierra.a.3	21K	76K	464K	1M	3M	10M	10M
sierra.a.4	20K	69K	346K	732K	2M	7M	10M
sierra.a.6	17K	53K	277K	670K	1M	5M	10M
tahoe	50K	151K	547K	2M	5M	10M	293M
tango	27K	94K	309K	710K	2M	10M	54M
whiskey	4K	156K	3M	9M	10M	10M	10M
yankee	109K	352K	1M	2M	3M	11M	111M

TABLE 6: The percentage of accesses that are to shared (2+ thread accesses) cachelines (i.e. dynamic sharing), alongside the shared footprint ratio (i.e. static sharing), for data and instructions. Overall, data is much less shared than instructions. The difference between the footprint and dynamic access percentages indicates that shared lines are on multiple threads' access footprints, so they accumulate references from each thread's stream, not necessarily non-uniform reuse within a footprint.

Workload	Shared D Footprint	Shared D Accesses	Shared I Footprint	Shared I Accesses
arizona	1.0%	2.8%	38.6%	67.1%
bravo.a	17.1%	27.2%	80.7%	82.4%
charlie	1.6%	14.2%	29.8%	79.8%
delta	2.4%	16.2%	51.7%	64.7%
merced	7.5%	18.3%	73.5%	89.0%
sierra.a.3	13.7%	23.6%	62.6%	85.3%
sierra.a.4	12.4%	22.4%	71.1%	85.7%
sierra.a.6	17.5%	30.2%	70.8%	91.9%
tahoe	8.6%	17.9%	74.2%	84.8%
tango	12.6%	20.5%	78.1%	89.0%
whiskey	7.1%	18.7%	52.4%	56.0%
yankee	7.7%	16.5%	89.0%	94.7%

observe that Google gains geomean 23% IPC from halving DRAM latency, whereas the next best non-graph suite, Qualcomm, only gains 10% geomean IPC. As the only way of mitigating the frequent compulsory misses is via the prefetcher, we predict that intelligent prefetching will become a more important component of future CPUs.

Page Size Sensitivity. The Google workloads see a geomean 16% IPC gain when enabling huge pages, the largest of any non-graph suite (9% for CloudSuite next). This is consistent with a large data footprint and random access / low reuse. We note that turning on huge pages wherever possible is highly beneficial; we leave internal fragmentation studies as future work.

TLB Insensitivity. With huge pages enabled, we find that the data TLB has a median MPKI of 6.9. This is significantly higher than any other workload suite. However, when running simulations with an unlimited data TLB, we see

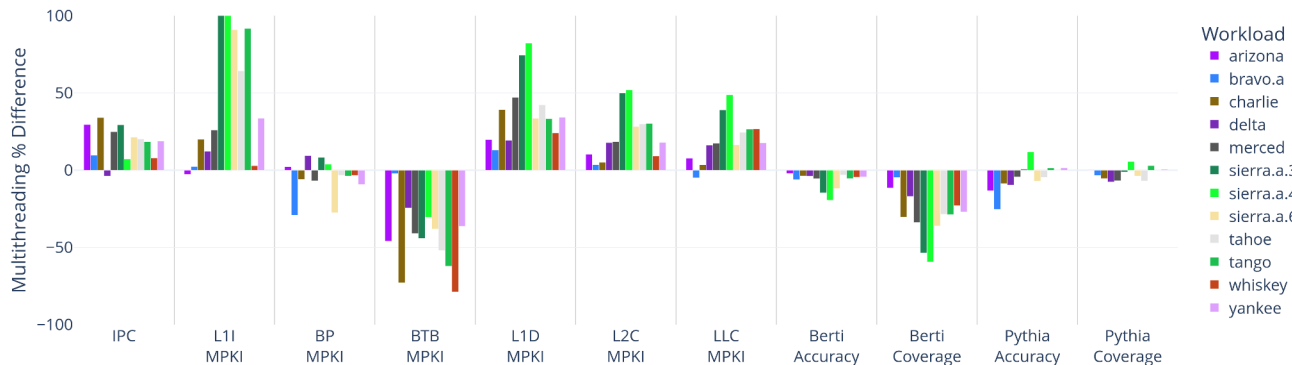


Figure 14: Percent difference from dedicated cores to multithreaded cores across several statistics. Multithreading significantly increases cache demand MPKI, but overall IPC is improved, suggesting that frontend sharing outweighs data cache loss. We find this to be primarily due to BTB sharing. In short, multithreading moves bottlenecks away from the frontend and more towards the backend. The two off-chart workloads experience a demand L1I MPKI of 236 and 169, respectively.

no performance change. We believe this to be due to the shared (level 2) TLB working as intended, which has an MPKI of only 0.04. The instruction TLB has 0 misses after warmup. Note that these results change with huge pages disabled, where the STLB misses frequently.

L1D Size Insensitivity. We find that performance is largely L1D size-insensitive within the 12KB to 192KB range (within geomean of $\pm 3\%$ IPC). We attribute this to the fact that the L1D is much too small to cover the secondary working set size, and seems (overly) sufficient to cache the first working set size. Future CPUs should prioritize other uses of chip area.

3.4. Frontend Performance

Although the backend and data caches are the primary reason for the slow execution of Google workloads, we explore the frontend characteristics of the Google workload suite. Namely, we find (1) the high instruction footprint leads to poor branch predictor, branch target buffer, and L1I performance; and (2) the L1I is size-insensitive, but prefetcher-sensitive.

3.4.1. Instruction Footprint. The L1I, BP, and BTB MPKI are shown in Figure 15, and are all high on the Google workloads. To look at why the frontend microarchitecture fails to perform well on the Google traces, Figure 16 shows the average instruction cacheline reuse distance. The Google workloads experience a significantly higher reuse distance than other workloads, leading to poor cacheability. They also feature one compulsory miss every 3.1k instructions, compared to Qualcomm’s 11k instructions and SPEC’s 685k instructions. Additionally, we examine the instruction footprint of each suite over a 100M instruction window. Notably, Google workloads feature a large instruction footprint with a median 2.8MB of unique instruction cachelines accessed, whereas the second most wide footprint (Qualcomm) accesses only 0.6MB of unique instruction cachelines. This highlights a new opportunity in server class

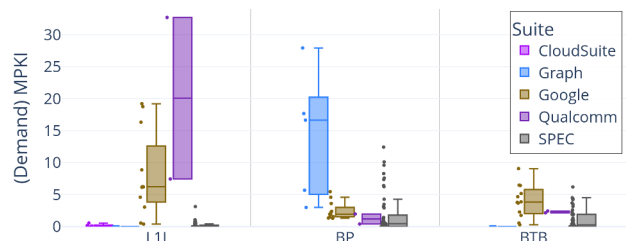


Figure 15: L1I, BP, and BTB MPKI in simulation. The three server workload suites experience more L1I misses than the simpler workload suites. The Google workloads feature the second highest and highest BP and BTB miss rates.

workloads to optimize the instruction cache replacement policy and prefetcher, which otherwise are not exercised, due to the increased instruction footprint. We quantify the working set size of each workload in Table 7. The Google traces require 5MB of instruction cache per core in order to hold the instruction working set size.

We also observe poor branch target buffer hit rates causing frequent mispredictions, shown in Figure 15. The Google workloads feature the lowest BTB hit rate (median 98%) compared to other benchmarks, even though they do not exhibit an unusually high frequency of branches. Similarly, the branch predictor exhibits a hit rate of 98.8%, the second lowest behind the graph workloads.

The increased instruction footprint may eventually lead to new bottlenecks in the L1I, BP, and BTB. Hence, further research should investigate novel policies for all three components.

3.4.2. L1I Sensitivity. We see that the Google workload traces are largely L1I size insensitive, even with the prefetcher disabled, in the 8KB-128KB range ($\pm 1\%$ IPC change for all suites except CloudSuite). The L1I is incapable of caching the necessary instruction footprint, and operates at a smaller, local window for server class workloads. We

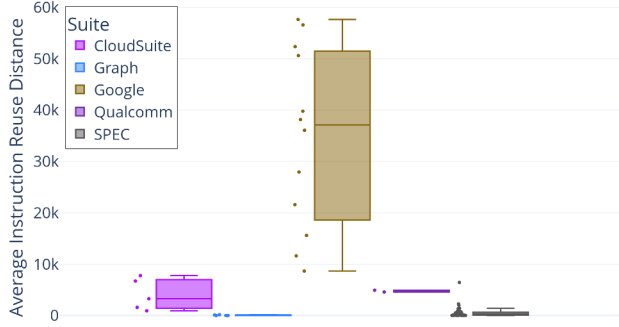


Figure 16: Instruction cacheline average reuse distance. The Google workloads have a substantially higher median reuse distance than any other suite, leading to poor cachability. This does not include the larger number of cachelines that are never reused.

TABLE 7: Instruction working set size per core. The working set size (WSS) is determined by varying the L1I size to unrealistic quantities until performance plateaus. Google features the highest instruction WSS at 5MB, causing spillover from the L1I into the L2C.

Suite	Mean Instruction WSS (KB)	Median Instruction WSS (KB)
CloudSuite	492.8	256
Google	4949.3	4096
Graph	8	2
Qualcomm	1536	1536
SPEC	5.8	2

further observe the Google workload working set sizes in Figure 17, showing that each working set begins at around 512KB and ends either at 2MB or 8MB. This means that feasible L1I sizes will not capture this working set, leading to our observed L2C spillover in Figure 12.

Given that added size does not help the L1I, prefetching is the only technique to reduce L1I-related stalls. Figure 18 shows the impact of enabling the Entangling Instruction Prefetcher (EIP) [46] at the L1I relative to disabling it. The Google workloads benefit from the EIP, implying that the prefetcher is an effective solution to mitigate L1I inefficiencies. Empirically, we see that the EIP does not significantly increase L1D miss latency, meaning that it has minimal effect on the rest of the memory subsystem. On the Google workloads, we see a median coverage of 64%, and an accuracy of 55%, leading to a geomean IPC gain of 2%. Note that the IPC gain is partially mitigated by the ineffective L2C prefetcher, and that disabling this prefetcher yields higher EIP IPC gains.

Given the combined information that the L1I is size insensitive and that correlational instruction prefetching is effective, we can deduce that the Google workloads operate in phases of instructions with long reuse distances.

4. Future Work

The traditional memory hierarchy is failing. The Google workloads feature an instruction footprint larger than

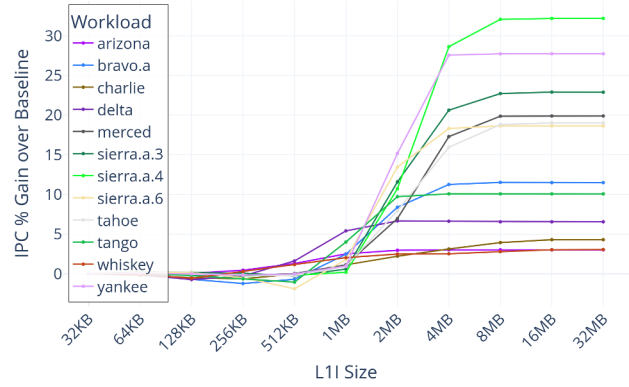


Figure 17: Google working set size estimation via L1I size ablation. The workloads fall between a 512KB and 8MB working set size per core.

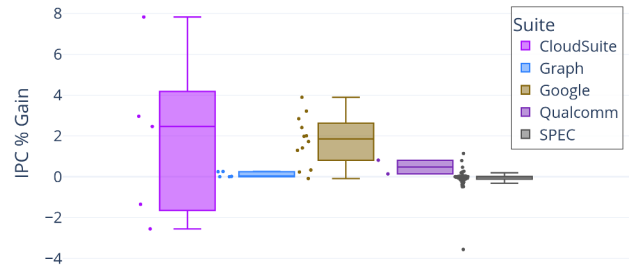


Figure 18: IPC Gain with L1I prefetcher (EIP) enabled. The server workloads benefit significantly more from the EIP than other suites.

other server workloads, and a data footprint comparable to graph workloads. Mitigating this simultaneous frontend and backend pressure would necessitate creative solutions and a rethinking of the memory hierarchy for server CPUs. For example, prior work at Google [3] proposed an L4, latency-optimized eDRAM cache to capture more reuse than the existing memory hierarchy. Similarly, Hermes [7] proposed bypassing the L2C and LLC for cachelines predicted to be non-resident, and [26] proposes bypassing the L2C for instructions. Such solutions hold promise for the next generation of server-class workloads.

Programs must be optimized to microarchitecture. Non-temporal loads and stores can significantly reduce cache pressure, but they are underutilized in current programs. Better support and awareness of microarchitecture hints potentially gives relief in the wake of high cache pressure. Additionally, thread should prioritize switching when microarchitecture sharing is possible, and heavy core over-subscription without core pinning should be avoided to mitigate core migration penalties.

5. Related Work

Clearing the Clouds [22] released CloudSuite, a standardized scale-out benchmark suite including several modern

cloud workloads. They perform a characterization of these workloads in 2012, observing high instruction miss rates, low memory level parallelism, high data set working sizes, and low memory bandwidth sensitivity. DeathStar-Bench [23] is a set of programs created to enable analysis of microservices. They find that around 40-50% of all cycles are stalled on the frontend across all workloads. Additionally, they observe major L1I pressure on half of their workloads. DCPerf [57] is a benchmark suite recently created by Meta that mimics internal Meta workload characteristics. Meta identifies a high L1I miss rate and significant frontend stalls in their production environment and in DCPerf. Finally, Memory Hierarchy for Web Search [3] analyses Google Web Search. They conclude that the backend is the primary performance bottleneck.

We perform the first in-depth microarchitecture analysis for Google traces [1]. Furthermore, our characterization uses simulation tools rather than PMUs, providing us the opportunity to dive deeper into microarchitectural bottlenecks. In particular, while prior top-down analyses [60] have shown frontend stalls, we identify relatively limited frontend potential compared to the backend. We most closely corroborate results with Memory Hierarchy for Web Search, suggesting that our converted traces are a good representation of Google’s datacenter workloads. They also identify that SMT improves IPC significantly ($\sim 1.5\times$) due to the latency bound nature of these workloads, as we expect it to.

6. Conclusion

This paper presents a characterization of the newly released Google workload traces (version 2), providing unique insight into modern cloud applications. We identify the primary bottleneck in Google workloads, identifying that the backend dominates over the frontend (a perfect BP+BTB only provides an 8% speedup). Topdown methodology in prior work conflates stalled cycles with potential performance. We also uncover the source of cache misses. The L2 consists of, on average, 38% data, 19% instructions, and 44% not-yet-hit prefetches. We quantify the extended working set size in Google workloads (7x larger than CloudSuite/Qualcomm). We also quantify the inefficacy of modern prefetchers. Additionally, we highlight that even with perfect caching/prefetching, DRAM latency will be a critical bottleneck for datacenter workloads. Compulsory misses are roughly tied in frequency with graph workloads, at a rate 3x more frequent than previous server benchmarks. We also note the impact of frequent context switches on cache state: multithreading improves net IPC by 18% due to BTB/instruction sharing despite 25-54% cache MPKI increases. We release a tool enabling this microarchitectural research. Overall, these insights provide direction as to where microarchitecture research should focus for the next generation of cloud workloads.

Acknowledgments

We thank our reviewers for their valuable feedback on the paper and artifact. This work was supported in part by PRISM, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. It was also supported in part by NSF graduate research fellowship. Generative AI tools were used with oversight to assist in visualization creations, to aid in finding references, and to edit and fix grammar and spelling.

Appendix

1. Abstract

We provide two artifacts. The first is the conversion tool that turns Google’s DynamoRIO thread-level traces into ChampSim-compatible core-level traces. Additionally, this artifact contains a link to a hosted variant of pre-scheduled traces. The second artifact is our branch of ChampSim which has been modified to add more statistics. Our traces are fully compatible with stock ChampSim; our changes just allow more data extraction. These two artifacts allow one to go from the Google provided traces to the simulated results we show in the paper. All that is needed is CPU hours.

2. Artifact check-list (meta-information)

- **Program:** Custom C++ program, ChampSim
- **Data set:** Google Workload Traces (version 2)
- **Experiments:** IPC, MPKI, etc. with various components.
- **How much disk space required (approximately)?:** 2TiB
- **How much time is needed to prepare workflow (approximately)?:** 30 min
- **How much time is needed to complete experiments (approximately)?:** 1000s of CPU hours
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache 2.0
- **Data licenses (if publicly available)?:** BSD
- **Archived (provide DOI)?:** 10.5281/zenodo.22100881

3. Description

3.1. How to access. The repositories are available at <https://github.com/matthewpipie/google-workload-traces-v2-to-champsim> and https://github.com/matthewpipie/ChampSim/tree/replacement_fixes (note the replacement_fixes branch).

3.2. Software dependencies. Modern Linux installation.

4. Installation

Directions for each repo are in their respective README.md files.

5. Experiment workflow

Download some subset of the Google workloads (e.g. arizona). Convert them to ChampSim format. Simulate the ChampSim traces using a machine configuration.

6. Evaluation and expected results

The expected output is for ChampSim to successfully simulate a core. All core-level statistics must be accumulated to compute aggregate statistics such as IPC or MPKI. These numbers can be directly compared with the numbers presented in the graph.

References

- [1] "Google workload traces version 2," <https://console.cloud.google.com/storage/browser/external-traces-v2>, accessed: 2026-04-21.
- [2] T. Asheim, R. Kumar, and B. Grot, "Fetch-directed instruction prefetching revisited," 2020. [Online]. Available: <https://arxiv.org/abs/2006.13547>
- [3] G. Ayers, J. H. Ahn, C. Kozyrakis, and P. Ranganathan, "Memory hierarchy for web search," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018, pp. 643–656.
- [4] G. Ayers, N. P. Nagendra, D. I. August, H. K. Cho, S. Kanev, C. Kozyrakis, T. Krishnamurthy, H. Litz, T. Moseley, and P. Ranganathan, "Asmdb: Understanding and mitigating front-end stalls in warehouse-scale computers," in *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019, pp. 462–473.
- [5] L. A. Barroso, U. Hözl, and P. Ranganathan, *The datacenter as a computer: Designing warehouse-scale machines*. Morgan & Claypool Publishers, 2018.
- [6] S. Beamer, K. Asanović, and D. Patterson, "The gap benchmark suite," 2017. [Online]. Available: <https://arxiv.org/abs/1508.03619>
- [7] R. Bera, K. Kanellopoulos, S. Balachandran, D. Novo, A. Olgun, M. Sadrosadati, and O. Mutlu, "Hermes: Accelerating long-latency load requests via perceptron-based off-chip load prediction," in *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '22. IEEE Press, 2023, p. 1–18. [Online]. Available: <https://doi.org/10.1109/MICRO56248.2022.00015>
- [8] R. Bera, K. Kanellopoulos, A. Nori, T. Shahroodi, S. Subramoney, and O. Mutlu, "Pythia: A customizable hardware prefetching framework using online reinforcement learning," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 1121–1137. [Online]. Available: <https://doi.org/10.1145/3466752.3480114>
- [9] M. Besta, Z. Vinarburg-Shmaria, Y. Schaffner, L. Schwarz, G. Kwasniewski, L. Gianinazzi, J. Beranek, K. Janda, T. Holenstein, S. Leisinger, P. Tatkowski, E. Ozdemir, A. Balla, M. Copik, P. Lindenberg, M. Konieczny, O. Mutlu, and T. Hoefer, "Graphminesuite: enabling high-performance and programmable graph mining algorithms with set algebra," *Proc. VLDB Endow.*, vol. 14, no. 11, p. 1922–1935, Jul. 2021. [Online]. Available: <https://dl.acm.org/doi/10.14778/3476249.3476252>
- [10] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: characterization and architectural implications," in *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '08. New York, NY, USA: Association for Computing Machinery, 2008, p. 72–81. [Online]. Available: <https://dl.acm.org/doi/10.1145/1454115.1454128>
- [11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, Aug. 2011. [Online]. Available: <https://doi.org/10.1145/2024716.2024718>
- [12] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, ser. VEE '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 133–144. [Online]. Available: <https://dl.acm.org/doi/10.1145/2151024.2151043>
- [13] D. Chasapis, G. Vavouliotis, D. A. Jiménez, and M. Casas, "Instruction-aware cooperative tlb and cache replacement policies," in *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, ser. ASPLOS '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 619–636. [Online]. Available: <https://doi.org/10.1145/3669940.3707247>
- [14] D. Chen, D. X. Li, and T. Moseley, "Autofdo: automatic feedback-directed optimization for warehouse-scale applications," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 12–23. [Online]. Available: <https://dl.acm.org/doi/10.1145/2854038.2854044>
- [15] A. Cho, A. Saxena, M. Qureshi, and A. Daglis, "Coaxial: A cxi-centric memory system for scalable servers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, ser. SC '24. IEEE Press, 2024. [Online]. Available: <https://doi.org/10.1109/SC41406.2024.00101>
- [16] CRC2 Organizers, "The 2nd cache replacement championship (CRC2)," <https://crc2.ece.tamu.edu/>, Jun. 2017.
- [17] DPC2 Organizers, "2nd data prefetching championship (DPC2)," <https://comparch-conf.gatech.edu/dpc2/>, Jun. 2015.
- [18] DPC3 Organizers, "3rd data prefetching championship (DPC3)," <https://dpc3.compas.cs.stonybrook.edu/>, May 2019.
- [19] DPC4 Organizers, "4th data prefetching championship (DPC4)," <https://sites.google.com/view/dpc4-2026/>, Feb. 2026, workshop in conjunction with the 32nd IEEE International Symposium on High-Performance Computer Architecture (HPCA), Sydney, Australia, February 1, 2026.
- [20] Q. Duong and C. Lin, "Streamlined on-chip temporal prefetching," in *2026 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2026, pp. 1–15.
- [21] J. Feliu, A. Perais, D. A. Jiménez, and A. Ros, "Rebasing Microarchitectural Research with Industry Traces," in *2023 IEEE International Symposium on Workload Characterization (IISWC)*. Ghent, Belgium: IEEE, Oct. 2023. [Online]. Available: <https://hal.science/hal-04234887>
- [22] M. Ferdman, A. Adileh, O. Kocberber, S. Volos, M. Alisafae, D. Jevdjic, C. Kaynak, A. D. Popescu, A. Ailamaki, and B. Falsafi, "Clearing the clouds: a study of emerging scale-out workloads on modern hardware," *SIGPLAN Not.*, vol. 47, no. 4, p. 37–48, Mar. 2012. [Online]. Available: <https://dl.acm.org/doi/10.1145/2248487.2150982>
- [23] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinisky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 3–18. [Online]. Available: <https://dl.acm.org/doi/10.1145/3297858.3304013>
- [24] N. Guber, G. Chacon, L. Wang, P. V. Gratz, D. A. Jimenez, E. Teran, S. Pugsley, and J. Kim, "The championship simulator: Architectural simulation for education and competition," 2022. [Online]. Available: <https://arxiv.org/abs/2210.14324>

- [25] B. R. Godala, S. P. Ramesh, G. A. Pokam, J. Stark, A. Sez nec, D. Tullsen, and D. I. August, "Pdip: Priority directed instruction prefetching," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 846–861. [Online]. Available: <https://doi.org/10.1145/3620665.3640394>
- [26] M. Hassan, C. H. Park, and D. Black-Schaffer, "Second-level caches: Not for instructions," *ACM Trans. Archit. Code Optim.*, vol. 22, no. 4, Dec. 2025. [Online]. Available: <https://dl.acm.org/doi/10.1145/3769080>
- [27] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Morgan Kaufmann, 2017.
- [28] IPC1 Organizers, "The 1st instruction prefetching championship (IPC1)," <https://research.ece.ncsu.edu/ipc/welcome/>, May 2020.
- [29] Y. Ishii, J. Lee, K. Nathella, and D. Sunwoo, "Rebasing instruction prefetching: An industry perspective," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 147–150, 2020.
- [30] A. Jain and C. Lin, "Back to the future: leveraging belady's algorithm for improved cache replacement," in *Proceedings of the 43rd International Symposium on Computer Architecture*, ser. ISCA '16. IEEE Press, 2016, p. 78–89. [Online]. Available: <https://dl.acm.org/doi/10.1109/ISCA.2016.17>
- [31] A. Jain, H. Lin, C. Villavieja, B. Kasikci, C. Kennelly, M. Hashemi, and P. Ranganathan, "Limocello: Prefetchers for scale," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ser. ASPLOS '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 577–590. [Online]. Available: <https://dl.acm.org/doi/10.1145/3620666.3651373>
- [32] A. Jaleel, K. B. Theobald, S. C. Steely, and J. Emer, "High performance cache replacement using re-reference interval prediction (rrip)," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 60–71. [Online]. Available: <https://doi.org/10.1145/1815961.1815971>
- [33] S. Kanev, J. P. Darago, K. Hazelwood, P. Ranganathan, T. Moseley, G.-Y. Wei, and D. Brooks, "Profiling a warehouse-scale computer," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ser. ISCA '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 158–169. [Online]. Available: <https://dl.acm.org/doi/10.1145/2749469.2750392>
- [34] T. A. Khan, A. Sriraman, J. Devietti, G. Pokam, H. Litz, and B. Kasikci, "I-spy: Context-driven conditional instruction prefetching with coalescing," in *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020, pp. 146–159.
- [35] T. A. Khan, M. Ugur, K. Nathella, D. Sunwoo, H. Litz, D. A. Jiménez, and B. Kasikci, "Whisper: Profile-guided branch misprediction elimination for data center applications," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 19–34.
- [36] W. Lin, Y. Lin, Y. Chen, S. Cai, Z. Jin, J. Xu, Y. Zhang, and W. Chen, "Btip: Branch triggered instruction prefetcher ensuring timeliness," *Electronics*, vol. 13, no. 21, 2024. [Online]. Available: <https://www.mdpi.com/2079-9292/13/21/4323>
- [37] ML-DPC Organizers, "ML-based data prefetching competition (ML-DPC)," <https://sites.google.com/view/mlarchsys/isca-2021/ml-prefetching-competition>, Jun. 2021.
- [38] S. Mostofi, S. Gupta, A. Hassani, K. Tibrewala, E. Teran, P. V. Gratz, and D. A. Jiménez, "Light-weight cache replacement for instruction heavy workloads," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1005–1019. [Online]. Available: <https://doi.org/10.1145/3695053.3730993>
- [39] N. P. Nagendra, B. R. Godala, I. Chaturvedi, A. Patel, S. Kanev, T. Moseley, J. Stark, G. A. Pokam, S. Campanoni, and D. I. August, "Emissary: Enhanced miss awareness replacement policy for l2 instruction caching," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: <https://doi.org/10.1145/3579371.3589097>
- [40] A. Navarro-Torres, B. Panda, J. Alastruey-Benedé, P. Ibáñez, V. Viñals Yúfera, and A. Ros, "Berti: An accurate local-delta data prefetcher," in *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '22. IEEE Press, 2023, p. 975–991. [Online]. Available: <https://doi.org/10.1109/MICRO56248.2022.00072>
- [41] S. Oh, M. Xu, T. A. Khan, B. Kasikci, and H. Litz, "Udp: Utility-driven fetch directed instruction prefetching," in *Proceedings of the 51st International Symposium on Computer Architecture (ISCA)*, ser. ISCA 2024, Jun. 2024.
- [42] —, "Udp: Utility-driven fetch directed instruction prefetching," in *Proceedings of the 51st Annual International Symposium on Computer Architecture*, ser. ISCA '24. IEEE Press, 2025, p. 1188–1201. [Online]. Available: <https://doi.org/10.1109/ISCA59077.2024.00089>
- [43] E. Perelman, G. Hamerly, M. Van Biesbrouck, T. Sherwood, and B. Calder, "Using simpoint for accurate and efficient simulation," in *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 318–319. [Online]. Available: <https://doi.org/10.1145/781027.781076>
- [44] P. Ranganathan, D. Stodolsky, J. Calow, J. Dorfman, M. Guevara, C. W. Smullen IV, A. Kuusela, R. Balasubramanian, S. Bhatia, P. Chauhan, A. Cheung, I. S. Chong, N. Dasharathi, J. Feng, B. Fosco, S. Foss, B. Gelb, S. J. Gwin, Y. Hase, D.-k. He, C. R. Ho, R. W. Huffman Jr., E. Indupalli, I. Jayaram, P. Kongetira, C. M. Kyaw, A. Laursen, Y. Li, F. Lou, K. A. Lucke, J. Maaninen, R. Macias, M. Mahony, D. A. Munday, S. Muroor, N. Penukonda, E. Perkins-Argueta, D. Persaud, A. Ramirez, V.-M. Rautio, Y. Ripley, A. Salek, S. Sekar, S. N. Sokolov, R. Springer, D. Stark, M. Tan, M. S. Wachsler, A. C. Walton, D. A. Wickeraad, A. Wijaya, and H. K. Wu, "Warehouse-scale video acceleration: co-design and deployment in the wild," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 600–615. [Online]. Available: <https://dl.acm.org/doi/10.1145/3445814.3446723>
- [45] 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, p. 65–79, Jul. 2010. [Online]. Available: <https://doi.org/10.1109/MM.2010.68>
- [46] A. Ros and A. Jimborean, "The entangling instruction prefetcher," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 84–87, 2020.
- [47] K. Seemakhupt, B. E. Stephens, S. Khan, S. Liu, H. Wassel, S. H. Yeganeh, A. C. Snoeren, A. Krishnamurthy, D. E. Culler, and H. M. Levy, "A cloud-scale characterization of remote procedure calls," in *Proceedings of the 29th Symposium on Operating Systems Principles*, ser. SOSP '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 498–514. [Online]. Available: <https://dl.acm.org/doi/10.1145/3600006.3613156>
- [48] A. Sez nec, "TAGE-SC-L Branch Predictors," in *Proceedings of the 4th Championship Branch Prediction*, Minneapolis, United States, Jun. 2014. [Online]. Available: <https://inria.hal.science/hal-01086920>
- [49] I. Shah, A. Jain, and C. Lin, "Effective mimicry of belady's min policy," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 558–572.
- [50] M. Shahrad, J. Balkind, and D. Wentzlaiff, "Architectural implications of function-as-a-service computing," in *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-52. New York, NY, USA: Association for Computing Machinery, 2019, p. 1063–1075. [Online]. Available: <https://dl.acm.org/doi/10.1145/3352460.3358296>

- [51] Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 861–873. [Online]. Available: <https://dl.acm.org/doi/10.1145/3445814.3446752>
- [52] J. Shun and G. E. Blelloch, "Ligra: a lightweight graph processing framework for shared memory," in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 135–146. [Online]. Available: <https://dl.acm.org/doi/10.1145/2442516.2442530>
- [53] S. Song, T. A. Khan, S. M. Shahri, A. Sriraman, N. K. Soundararajan, S. Subramoney, D. A. Jiménez, H. Litz, and B. Kasikci, "Thermometer: profile-guided btb replacement for data center applications," in *Proceedings of the 49th Annual International Symposium on Computer Architecture*, ser. ISCA '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 742–756. [Online]. Available: <https://dl.acm.org/doi/10.1145/3470496.3527430>
- [54] A. Sriraman and T. F. Wenisch, " μ suite: A benchmark suite for microservices," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 1–12.
- [55] Standard Performance Evaluation Corporation. [Online]. Available: <https://www.spec.org/cpu2006/>
- [56] Standard Performance Evaluation Corporation. [Online]. Available: <https://www.spec.org/cpu2017/>
- [57] W. Su, A. Dhanotia, C. Torres, J. Gandhi, N. Gholkar, S. Kanaujia, M. Naumov, K. Subramanian, V. Andrei, Y. Yuan, and C. Tang, "Dcperf: An open-source, battle-tested performance benchmark suite for datacenter workloads," in *Proceedings of the 52nd Annual International Symposium on Computer Architecture*, ser. ISCA '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 1717–1730. [Online]. Available: <https://doi.org/10.1145/3695053.3731411>
- [58] D. Ustiugov, P. Petrov, M. Kogias, E. Bugnion, and B. Grot, "Benchmarking, analysis, and optimization of serverless function snapshots," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 559–572. [Online]. Available: <https://dl.acm.org/doi/10.1145/3445814.3446714>
- [59] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The splash-2 programs: characterization and methodological considerations," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*, ser. ISCA '95. New York, NY, USA: Association for Computing Machinery, 1995, p. 24–36. [Online]. Available: <https://dl.acm.org/doi/10.1145/223982.223990>
- [60] A. Yasin, "A top-down method for performance analysis and counters architecture," in *2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2014, pp. 35–44.
- [61] K. Zhu, Y. Zhao, Y. Gao, P. Braun, T. A. Khan, H. Litz, B. Kasikci, and S. Deng, "From optimal to practical: Efficient micro-op cache replacement policies for data center applications," in *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2025, pp. 716–731.