2018 IEEE International Symposium on High Performance Computer Architecture

Memory Hierarchy for Web Search

Grant Ayers*
Stanford University
ayers@cs.stanford.edu

Jung Ho Ahn*
Seoul National University
gajh@snu.ac.kr

Christos Kozyrakis
Stanford University
christos@ee.stanford.edu

Parthasarathy Ranganathan
Google
partha.ranganathan@google.com

***Abstract*—Online data-intensive services, such as search, serve billions of users, utilize millions of cores, and comprise a significant and growing portion of datacenter-scale workloads. However, the complexity of these workloads and their proprietary nature has precluded detailed architectural evaluations and optimizations of processor design trade-offs. We present the first detailed study of the memory hierarchy for the largest commercial search engine today. We use a combination of measurements from longitudinal studies across tens of thousands of deployed servers, systematic microarchitectural evaluation on individual platforms, validated trace-driven simulation, and performance modeling – all driven by production workloads servicing real-world user requests.**

Our data quantifies significant differences between production search and benchmarks commonly used in the architecture community. We identify the memory hierarchy as an important opportunity for performance optimization, and present new insights pertaining to how search stresses the cache hierarchy, both for instructions and data. We show that, contrary to conventional wisdom, there is significant reuse of data that is not captured by current cache hierarchies, and discuss why this precludes state-of-the-art tiled and scale-out architectures. Based on these insights, we rethink a new cache hierarchy optimized for search that trades off the inefficient use of L3 cache transistors for higher-performance cores, and adds a latency-optimized on-package eDRAM L4 cache. Compared to state-of-the-art processors, our proposed design performs 27% to 38% better.

***Keywords*-web search; warehouse-scale computing; memory systems;**

I. Introduction

On-line, data-intensive (OLDI) workloads, such as search, social networks and Software as a Service (SaaS), represent the majority of activity on the Internet. These user-facing services use thousands of servers to mine through massive datasets and answer queries in near real-time. Search is one of the first OLDI services and by far the most ubiquitous. Over the past 20 years, search has spearheaded the switch to scale-out clusters of commodity servers [2], [58], motivated the deployment of reconfigurable accelerators [17], [45], and inspired research on a wide range of topics for datacenter computing [13], [34], [35], [40], [41], [47]. Essentially, search is for OLDI what the sort benchmark [29] is for the database domain: a canary workload that drives the development and adoption of new hardware and software technologies.

*Work done while authors were at Google.

Despite the importance of search, there is limited understanding of its microarchitectural behavior, in particular with respect to the memory hierarchy. This is mainly due to the proprietary and complex nature of search workloads. Hence, it is difficult to calibrate simpler benchmarks, which are essential for obtaining microarchitectural insights, evaluating design trade-offs, and proposing architectural optimizations.

This paper presents the first detailed study of the memory hierarchy behavior for Google's web search, the largest and most popular commercial search service today. We collect data from tens of thousands of production servers, system-level microarchitectural evaluations from two representative platforms, and validated trace-driven simulation results and performance modeling – all driven by production workloads servicing real-world user requests. This data allows us to make three contributions.

First, we present *a detailed microarchitectural characterization of production search services using fleet-wide measurements. Among our findings, we identify significant bottlenecks and calibrate against traditional benchmarks* (§II). Our characterization shows that search exhibits high misses-per-instruction for branches, L2 instruction misses, and L3 data misses, which makes it quite different from SPEC CPU2006 workloads and even from academic search applications [13]. Production search benefits significantly from key features of contemporary multi-core chips such as increased core counts, multi-threaded cores, support for large pages, and prefetching. However, a detailed Top-Down [60] breakdown of execution cycles shows that just 32% of instruction slots are spent retiring instructions. The biggest bottlenecks are data access related; back-end stalls (20.5% of slots), followed by branch mispredictions (15.4% of slots) and instruction cache misses (13.8% of slots).

Second, we provide *a detailed quantitative analysis of the cache hierarchy for search with the goal to identify opportunities for reducing data access related back-end stalls and for making efficient use of transistor resources* (§III). We show that L3 caches capture the working set for code and stack accesses nearly perfectly, but experience significant capacity misses for index shard and heap accesses. The locality in code and stack accesses can be captured with much smaller L3 caches, allowing for more area-effective uses of on-chip transistors. While the working set for the index shard is huge and exhibits no temporal locality, heap

2378-203X/18/$31.00 ©2018 IEEE
DOI 10.1109/HPCA.2018.00061

accesses exhibit high locality and sharing, but in a working set of roughly 1 GiB – an order of magnitude higher than what current last-level caches can capture. This indicates that state-of-the-art tile-based and scale-out designs [5], [37], [44] are not appropriate due to their limited sharing capacity, and that, contrary to conventional wisdom, it is actually practical to target data reuse in search with very large caches. Finally, we show that the lack of memory-level parallelism in L3 cache accesses leads to a linear relationship between search performance and L3 average memory access time (AMAT). As long as memory and core bandwidth is not saturated, search performance primarily depends on memory latency, and analytical models can be developed to study performance with good accuracy.

Third, we build on these insights to *evaluate two non-traditional memory hierarchy optimizations targeted at OLDI workloads like search* (§IV). We evaluate an iso-area design that trades off the inefficient use of L3 cache transistors in current processors for higher-performance cores. We demonstrate that such an approach leads to a 14% performance improvement. Next, we propose adding a high-capacity, on-package L4 cache implemented with low cost, eDRAM technology [10]. In addition to preferring faster eDRAM over slower DRAM chips, we optimize the L4 cache for low latency by using a direct-mapped design [46], co-locating tags and data in eDRAM rows, and performing L4 tag lookups and main memory access scheduling in parallel. The overall L4 design is well within the technology limits validated by recent L4 cache implementations in commercial systems [42], [52]. The combination of the efficient use of on-chip transistors with a low-latency L4 cache provides a novel yet practical design point that balances the computational demands of a scale-out workload with the flexibility to share large data working sets. Overall, our design improves performance by 27% over a state-of-the-art baseline, with our sensitivity results showing even higher potential in the future (38%).

II. Characterizing Search in the Wild

A. Overview of Production Search

We study production search workloads serving actual queries from what is likely the largest and most popular commercial search service. Google's search framework consists of three components: crawling, indexing, and serving [11]. The crawling system gathers content from various data sets (e.g., web, images, and tweets). The indexing system extracts useful information from this data and generates the search index. Due to its sheer volume, the index is partitioned into shards which are spread across a large number of servers called *leaves*. The serving system receives queries and returns the most relevant content including snippets based on various criteria such as PageRank [8], personal preferences, and language. The crawling and index systems are batch workloads that focus on high computational throughput [12], whereas the serving system is a user-facing, latency-critical workload.

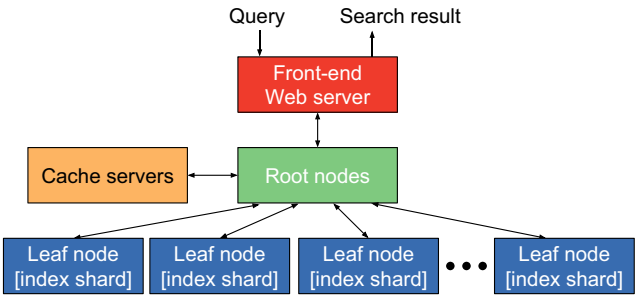


Figure 1: The serving system of Google's search service. Queries propagate down to all leaf nodes. Results propagate up the tree, with intermediate parents scoring and ordering content.

We focus on the search serving system shown in Figure 1. A query first arrives at a front-end web server that forwards it to a back-end root for preprocessing such as spell checking [11]. Next, the query is delivered to thousands of leaf servers, each holding a shard of the index. Each leaf interacts with the root or intermediate parents in a tree hierarchy to retrieve the corresponding results from its index and score them. The root server extracts the snippets of the chosen results, and asks the front-end web server to return the resulting web page. Popular queries can consume a significant amount of resources, so caching is used in various levels of the hierarchy to improve throughput and latency.

Search has the following important attributes: First, because of limited memory and storage capacity of any single machine, we need a large number of servers to store the ever-growing index. Second, query processing often requires billions of instructions to retrieve and score relevant results using algorithms that rapidly evolve [9], [45]. Third, search has ample request-level parallelism as each query is mostly independent [3]. Fourth, search is latency-critical, as a sub-second increase in response time has a big impact on user experience and revenue potential [49].

B. Methodology

Table I characterizes production search with data collected from longitudinal studies across a large number of production servers as well as more specific platform-level studies on individual machines. For the former, we use a fleet-wide methodology similar to that described in [27], [48] and focus on four key metrics: IPC, misses per kilo-instruction (MPKI) for L2 and L3 caches, as well as branch MPKI. The production web search we consider is a rapidly-evolving large code base with big launches often at a weekly granularity. Additionally, there are multiple search services specialized for different needs. There is also heterogeneity of hardware platforms. Confidentiality reasons prevent us

	Search (fleet-wide) By components						Search (lab) By platforms		SPEC CPU2006				CloudSuite
	Leaf			Root			S1 Leaf						
Metric	S1	S2	S3	S1	S2	S3	PLT1	PLT2	400. perlbench	429. mcf	445. gobmk	471. omnetpp	Web Search
Per-core IPC	1.34	1.63	1.46	1.03	1.14	1.08	1.27	1.92	2.72	0.15	1.43	0.30	1.61
L3$ load MPKI	2.20	1.89	1.78	4.20	3.05	3.19	2.43	1.15	0.48	56.92	0.29	24.92	0.03
L2$ instr MPKI	11.83	12.44	14.10	12.02	19.62	13.97	10.78	2.53	0.58	0.31	3.02	0.63	0.28
Branch MPKI	8.98	6.17	7.99	4.71	4.84	5.37	9.47	11.50	1.80	11.32	18.40	5.32	0.51

Table I: Key performance metrics for search, SPEC CPU2006 [18], and Web Search of CloudSuite v3 [13]. The first six columns present data from different production search servers serving real-world queries. The columns marked PLT1/2 present data from the most popular service S1 on the hardware platforms detailed in Table II.

	PLT1 [16]	PLT2 [53]
Microarchitecture	Intel Haswell	IBM POWER8
Number of sockets	2	2
Cores	18 per socket	12 per socket
SMT	2	8
Cache block size	64 B	128 B
L1-I$ (per core)	32 KiB	32 KiB
L1-D$ (per core)	32 KiB	64 KiB
Private L2$ (per core)	256 KiB	512 KiB
Shared L3$ (per socket)	45 MiB	96 MiB

Table II: Key attributes of PLT1 and PLT2 platforms.

from providing more details, but to get a high-level sense of the similarities and differences across these workloads, we present leaf node measurements for three search services, S1, S2, and S3. Each service represents a different software code base, running on mostly homogeneous hardware.

The fleet-wide data includes averages across multiple machines over time. We also focus on the S1 service that is the biggest consumer of search cycles in our fleet and present individual server-level characterization for its leaf role on two representative platforms. While these server-level experiments are run in the lab, they still use the production code and service real-world queries. Table II summarizes the key attributes of the two hardware platforms: PLT1 (Intel) and PLT2 (IBM). All systems use 256 GiB of memory and are balanced in their use of Flash and disks. We chose these systems because they provide the maximum flexibility for the experiments we want to conduct. Both allow us to evaluate a large number of cores (4-18 per socket). PLT1 supports Cache Allocation Technology (CAT [20]), which allows us to study different cache configurations. PLT2 allows us to illustrate increased Simultaneous Multithreading (SMT) effects and larger cache capacity. In the interest of space, we present results only for PLT1 unless otherwise mentioned.

C. Key Search Characteristics

Table I shows that, while there is substantial fluctuation across the various search measurements, there are several common characteristics for all cases:

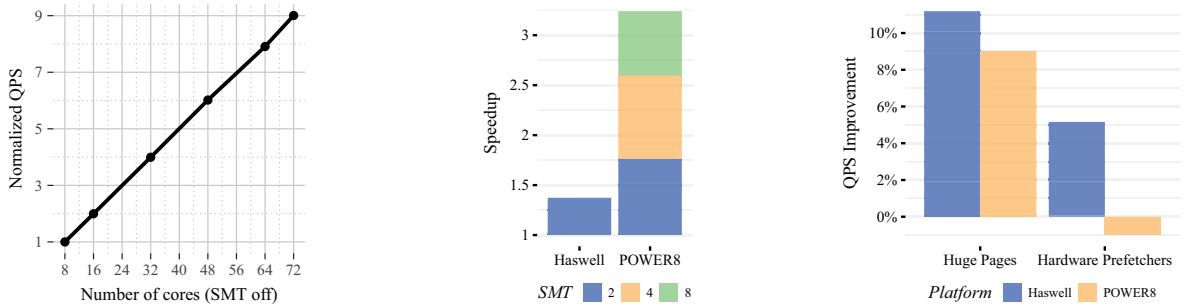
- Branch MPKI is uniformly high, suggesting a complex workload with numerous data-dependent branches.
- L2 MPKI for instruction accesses is high, suggesting a large code working set. Yet, the L3 MPKI for instructions is negligible, suggesting that shared L3 caches are sufficiently large to capture the code working set.
- In contrast, the L3 MPKI for data is significant, which is expected as each server searches through a large index shard.
- Nevertheless, the per-core IPC is reasonably high primarily due to exploiting request-level parallelism with SMT.

Comparing across leaf and root servers, we observe leaf servers have higher branch MPKI. Comparing across platforms for service S1, the variability is fairly consistent with differences in microarchitecture, ISA, number of cores and threads, and aggregate L2 and L3 cache sizes. However, the previously stated observations hold across platforms.

D. Comparison with Other Benchmarks

Table I also contrasts search to representative workloads from the SPEC CPU2006 suite [18] and the Lucene-based Web Search in CloudSuite v3 [13] for equivalent performance characterizations. Compared to SPEC workloads, search has a much larger code working set; hence its L2 MPKI for instructions is at least 3.6× higher than the most code-intensive SPEC application (445.gobmk). While search stresses the L3 cache for data more than compute-bound SPEC benchmarks like 400.perlbench and 445.gobmk, it is less intensive than memory-bound SPEC benchmarks like 429.mcf and 471.omnetpp. In general, there are three major differences between search and the SPEC CPU2006 workloads: Frequent branch mispredictions, a large code working set that overflows the private L2 caches, and significant L3 misses for data. [27].

Surprisingly, our production search is also significantly different from the CloudSuite Web Search, which shows much lower MPKI for branches, L2 instruction accesses, and L3 data accesses. The original CloudSuite paper shows that its Web Search consumes roughly 1% of peak DRAM bandwidth, while our production search consumes 40-50% of peak DRAM bandwidth. While the CloudSuite Web



(a) Search Scalability (b) Simultaneous Multithreading (SMT) (c) Huge Pages and Prefetching

Figure 2: The impact of scaling core count, thread-level parallelism, large pages, and prefetching on search throughput.

`Search` may be representative of workloads like Elastic-Search or Apache Solr [15], it is not as complex or data-intensive as our production search.

These observations are a good illustration of the need to have fleet-wide characterizations of microarchitectural behavior of search "in the wild" to help calibrate the use of existing benchmarks and guide the design of new benchmarks.

E. Hardware Optimizations

We next quantify the benefits from four hardware optimizations, all available on the deployed machines in our fleet: (a) increased number of cores, (b) increased SMT, (c) huge pages, and (d) hardware prefetching.

Figure 2a shows the relative throughput of a search `S1` leaf (queries per second (QPS)) as a function of the number of cores used. We present data from a *4-socket* `PLT1` system to show the scaling behavior for larger core counts. SMT (Hyper-Threading) is off for this experiment. The results show excellent scaling with the number of cores, even to relatively high counts. The near-perfect scaling implies that search has a limited amount of read/write sharing or locking in the memory system, does not saturate the bandwidth of shared caches and main memory, and is not I/O-bound on network or storage[1].

Figure 2b presents the improvement in search throughput with SMT for both `PLT1` (Intel x86 Haswell) and `PLT2` (IBM POWER8) platforms. SMT with 2 threads on `PLT1` provides a 37% performance boost by hiding front-end and back-end issues of single-thread execution (see Figure 3). SMT on `PLT2` provides improvement between 76% for SMT-2 and 224% (3.24x speedup) for SMT-8. As expected, we see diminishing returns in both architectures as thread count grows due to increased contention for shared resources.

[1] When scaling the number of cores, the L3 capacity per core varies. However, within the range of 72 cores, the impact is small enough that the scaling factor remains very close to 1.

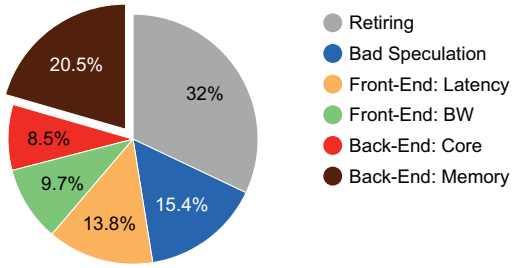


Figure 3: The first two levels of the Top-Down breakdown [60] of a `S1` leaf node on `PLT1`.

Figure 2c shows the throughput increase when we turn on large pages (2 MiB on `PLT1`, 16 MiB on `PLT2`) and enable the default hardware prefetchers. Large pages benefit performance by roughly 10% on both platforms, which is expected for a data-intensive program that touches nearly all physical memory. The benefits for `PLT1` (x86) are slightly higher partly because of the higher ratio of small to large page size (4KiB:2MiB for `PLT1`, 64KiB:16MiB for `PLT2`).

Our `PLT1` system has four configurable hardware prefetchers; two for the L1-D cache and two for the L2 cache [56], while the `PLT2` system has a user-tunable hardware prefetch engine [6], [25]. We see a 5% benefit on `PLT1` and slight degradation for `PLT2` when prefetchers are enabled. The contrast is due to several hardware differences between the systems, including cache block sizes (64 bytes for `PLT1`, 128 bytes for `PLT2`), cache capacities, and prefetch algorithms. Unless otherwise mentioned, all of our results are with both features on, except for `PLT2` where we disable hardware prefetching. Overall, search benefits significantly from hardware features like high core counts, threads, large pages, and prefetching.

F. Key Opportunities

We use the Top-Down [60] methodology to identify and categorize execution stalls. This methodology reveals architectural bottlenecks in the presence of the many latency-

hiding mechanisms of out-of-order processors. Bottlenecks are expressed in units of cycles or *instruction slots*, where an *n*-wide processor has a maximum throughput of *n* instruction slots per cycle. The slots are categorized and each category contributes a certain percentage to the overall execution profile.

Figure 3 shows the execution breakdown of a `S1` leaf in `PLT1`. Search retires instructions for only 32% of all slots. Its performance is limited by inefficiencies in all parts of the processor core: Mispredicted branches are responsible for 15% of wasted slots. Front-end issues take 24% of slots, including instruction cache misses (latency) and decoding inefficiencies (bandwidth). Back-end issues take 29% of slots, including data cache misses (memory) and execution serialization (core).

This data shows the memory hierarchy as a big opportunity for improvement. If all 21% of memory stall slots included in the back-end memory component were converted to useful retired slots, the upper-bound gain would be approximately 64% performance improvement to the retired instruction count. This is a significant gap considering that caches routinely occupy roughly half the area of processor chips and that the pin requirements and power consumption of memory channels are frequently limiting factors for many multi-core designs. Moreover, as the adoption of accelerators increases compute throughput by integer factors [9], [26], [45], the memory hierarchy challenges for data-intensive workloads like search will only increase.

III. Characterizing Memory Hierarchy for Search

A. Analysis Methodology

We now perform an in-depth characterization of the memory hierarchy for search, looking for the *intuition* for what is happening and answers to *what-if* questions. Specifically, the key questions we answer include: How does the working set and footprint scale with the number of cores? How do the misses relate back to software structures? What are the sources of misses (capacity, conflict, cold)? What is the breakdown of instruction and data misses throughout the hierarchy? What would the miss rate be with different cache sizes? How does improving the cache properties (hit rate, latency) improve search performance (IPC, QPS)?

Answering such questions exceeds the capabilities of existing performance counters and hardware knobs alone. Moreover, performance counters can be riddled with errata and require careful validation before they can be of any use [21]. Furthermore, we are not able to use any of the existing microarchitecture simulators with timing models for the cache hierarchy, as none of them is capable of running production search for a non-trivial amount of virtual time.

To address these challenges, we adopt a methodology that combines validated measurements from real machines, trace-driven functional cache simulation, and analytical models based on curve-fitting data from the fleet. When possible, we experiment with multiple hardware configurations and collect actual performance statistics. For example, changing the core and uncore frequency allows us to study sensitivity to latency. Intel's Cache Allocation Technology (CAT) [20] allows us to change cache sizes and create scenarios with different miss rates and effective memory latencies.

For certain detailed insights, we used a validated trace-driven cache simulator. We use Intel's Pin tool [38] to capture full instruction and data traces from a machine that is in steady state and serving real user queries. Our trace-based analyses utilize 16 threads worth of traces containing a sum total of 135 billion instructions. We present data based on one collection, but note that the results are qualitatively similar over multiple such collections over a year. The simulator models inclusive and non-inclusive caches, various allocation policies, associativities, block sizes, and LRU replacement. It does not model coherence, which is acceptable as there is negligible read-write sharing between search threads. Unless mentioned otherwise, all simulation analyses are based on a `PLT1`-like system with SMT off: Each thread uses private L1 caches (32 KiB each for instructions and data) and a private L2 cache (256 KiB unified). Private caches are modeled as 8-way set-associative. We model a 40 MiB, 20-way set-associative, unified L3 cache. All caches use LRU replacement. Our simulator provides miss rates and MPKI data, but not timing information for the cache hierarchy. Instead, we incorporate miss rates into our measurement-based analytical model to evaluate performance (see §III-D).

B. Footprint and Working Set Scaling

Figure 4 analyzes the measured *memory footprint* as we scale the used cores from 6 to 36 on a 2-socket, `PLT1` system with SMT off. This is the steady-state total allocated memory for the various code, heap, and stack segments. All remaining main memory is allocated to the index shard, whose size is in the 100s of GiB and does not change with the number of cores. There are two interesting observations: First, the heap dominates the non-shard memory footprint as it is about one order of magnitude larger than the code and stack segments. Second, even though performance scales linearly with the number of cores, the heap size grows slower as there are several shared data-structures between search threads.

Figure 5 analyzes the search *working set*, which includes any data or code accessed at least once within a trace collected from a server already in steady state. We focus on the heap and shard segments, as the code and stack segments are much smaller (e.g., code remains constant near 4 MiB). As expected, the working set is smaller than the total footprint. The shard footprint is constant, but its working footprint grows rapidly with the number of cores. This is primarily due to accesses to disjoint parts of the large shard by the

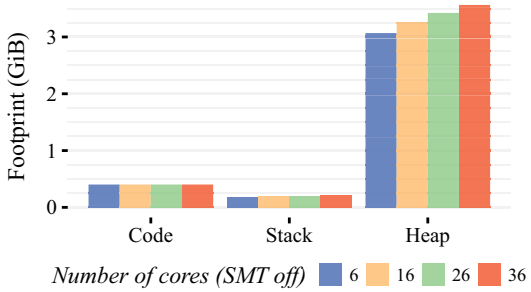


Figure 4: Allocated memory footprint as we scale cores. The shard segment (not shown) is in the 100s of GiB.

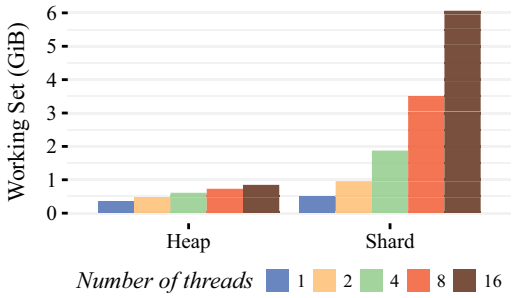


Figure 5: Accessed working set for the heap and shard segments as we scale cores.

parallel threads, as there is little locality left in search queries after intermediate cache servers are used (see Figure 1). With enough time, even a single thread's shard working set would eventually converge to the full shard footprint. In contrast, the working set for the heap grows significantly slower as there is sharing in heap structures. At 16 threads, the total heap size accessed is 1 GiB. The heap working set is significantly smaller than its allocated footprint, suggesting that many heap-allocated data-structures are relatively cold in steady-state.

C. Miss Analysis

Figure 6a illustrates the overall effectiveness of each caching level (L1, L2, and L3) in terms of misses. We simulate and sum the number of misses from all of the caches at each level and then present it as MPKI. We further break down misses based on the type of access: code (instruction), heap (data), or shard (data). Collectively, the L1 and L2 caches experience significant misses for code, heap, and shard. Heap and shard accesses frequently miss in the L3 as well, and are serviced by main memory. Modern, out-of-order processors are often unable to hide main memory accesses and these misses typically stall the pipeline. (We will present data about this in Figure 8.) However, the shared L3 cache eliminates virtually all instruction cache misses.

Figures 6b and 6c also quantify the impact of L3 cache capacity, from 4 MiB to 2 GiB. We continue to assume the same L1 and L2 caches as `PLT1`.

- A 16 MiB, shared L3 cache is sufficient to eliminate *code misses* even for a complex workload like search.
- In contrast, the L3 cache is ineffective with *shard accesses* due to its size and the limited locality. Even a 2 GiB L3 cache barely reaches a 50% shard hit rate.
- However, *large shared caches are highly effective for heap accesses*. At 1 GiB capacity, the hit rate for heap references is 95% and the combined MPKI drops from 3.51 with a 32 MiB cache to 1.37 with a 1 GiB cache.

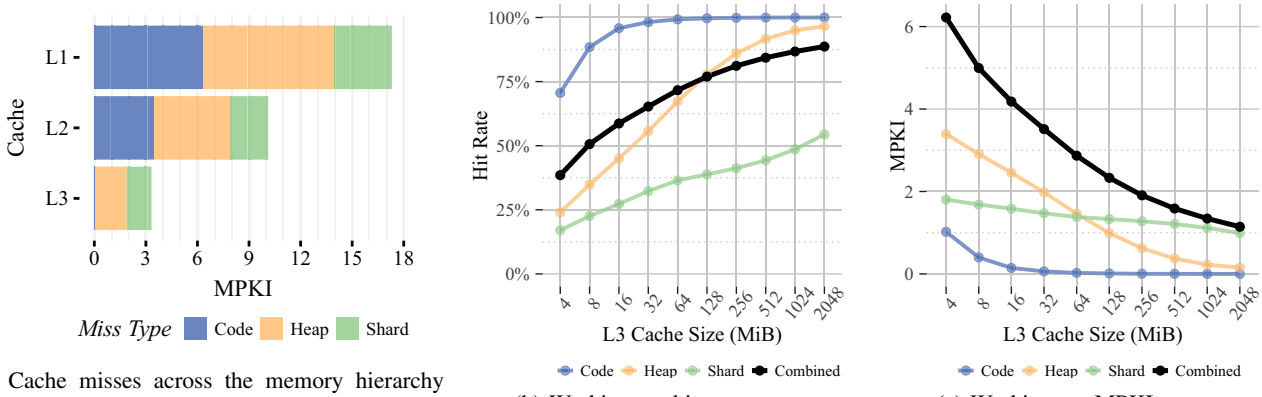
These results suggest that a very large shared cache can improve overall performance by capturing the significant locality in heap accesses, challenging the conventional wisdom that workloads like search have mainly random access patterns and no locality. It is also important to relate this result to the observation from Figure 5 that the heap working set grows slowly with core count. In other words, to capture heap access locality, a shared cache needs to be large for any core count, but does not need to grow linearly with the number of cores.

Further analysis of the *type* of miss shows that conflict misses are not as significant as capacity misses. Figure 7a presents results with same-sized but *fully-associative* caches throughout the hierarchy to eliminate all conflicts. MPKI is reduced by approximately 7.4% in the L1 caches, but by less than 1% in the L2 and L3, indicating that the default associativity of the caches is a good design point. Looking at other types of misses, the shard working set is so large and exhibits so little reuse that most of its accesses can be considered cold misses. In contrast, heap accesses exhibit mostly capacity misses. There are virtually no coherence misses due to the lack of read-write sharing between threads.

Figure 7b illustrates the benefits from spatial locality with varying cache block sizes. In most cases, the baseline 64-byte block size captures most of the spatial locality without leading to degraded performance. There is limited benefit from larger cache lines, which is consistent with the benefit seen from hardware prefetchers in Figure 2c. For example, of the 5% performance benefit from hardware prefetchers in `PLT1`, nearly 1% stems from spatial locality exploited by the L2 adjacent-line prefetcher.

D. Impact of Cache Design on Performance

Before we can make informed choices about the memory hierarchy design, we need to go beyond basic miss-rate curves and understand the impact of cache parameters on overall search performance. As a first step, we establish that the *L3 hit rate is a reliable predictor of overall search performance*. Figure 8a reports performance (IPC) on a `PLT1` system (measured by performance counters) as a function of the L3 hit rate. We adjust hit rate by varying the L3 capacity available to search in a `PLT1` system via

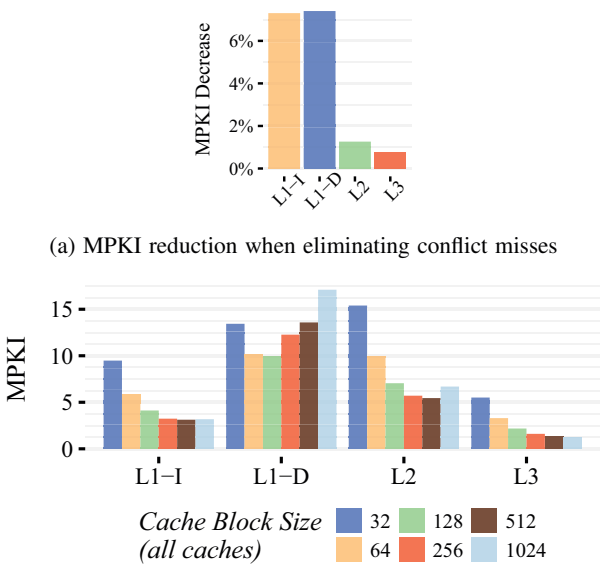


(a) Cache misses across the memory hierarchy classified by type.

(b) Working set hit rate curve

(c) Working set MPKI curve

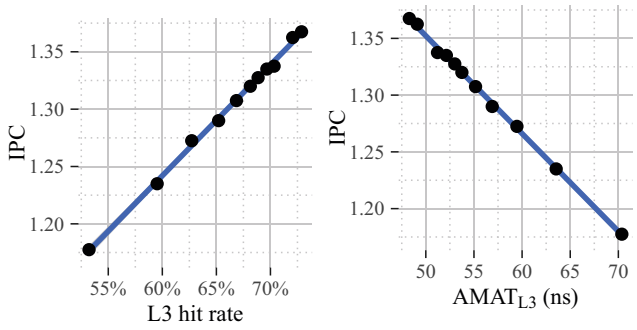
Figure 6: Cache misses and hit-rate curves categorized by access type.



(a) MPKI reduction when eliminating conflict misses

(b) MPKI for various block sizes

Figure 7: MPKI sensitivity to associativity and block size.



(a) IPC and L3 hit rate

(b) IPC and average L3 access time

Figure 8: Performance vs. L3 hit rate and latency. The linear relationships indicate low per-request memory-level parallelism in the L3 cache, which allows us to model performance as a function of cache hit rates and latencies.

CAT cache partitioning. Hit rates between 53% and 73% reflect a strong linear relationship with the resulting IPC. Intuitively, we expect the L3 hit rate to affect performance somewhat, as the Top-Down analysis in Figure 3 showed significant memory-bound slots. More surprisingly, Figure 8a shows that there is insufficient per-thread memory-level parallelism (MLP) to effectively hide L3 misses.

Second, we show that *we can model performance based on cache hit rates and memory latencies*, specifically for post-L2 caches and within the domain of our measurements. We use performance counters to measure the L3 hit rate h_{L3}, the L3 access latency t_{L3}, and the total, round-trip memory latency t_{MEM}. This allows us to calculate the average memory access time (AMAT) of the L3 cache as $\text{AMAT}_{L3} = h_{L3}t_{L3} + (1-h_{L3})t_{MEM}$. Figure 8b extends Figure 8a by showing a linear relationship between IPC and AMAT_{L3} for the range of latencies we were able to exercise, also expressed by the linear model

$$\text{IPC} = -8.62 \times 10^{-3} \cdot [h_{L3}t_{L3} + (1-h_{L3})t_{MEM}] + 1.78 \quad (1)$$

In summary, our model allows us to evaluate web search performance for any memory hierarchy configuration starting with the L3 cache, knowing only hit rates and access latencies. We will use this model in §IV. We conducted experiments to explore further latency data points by modifying core and uncore frequency in a `PLT1` system. While not plotted here, these measurements also showed strong linear correlation between performance and latency.

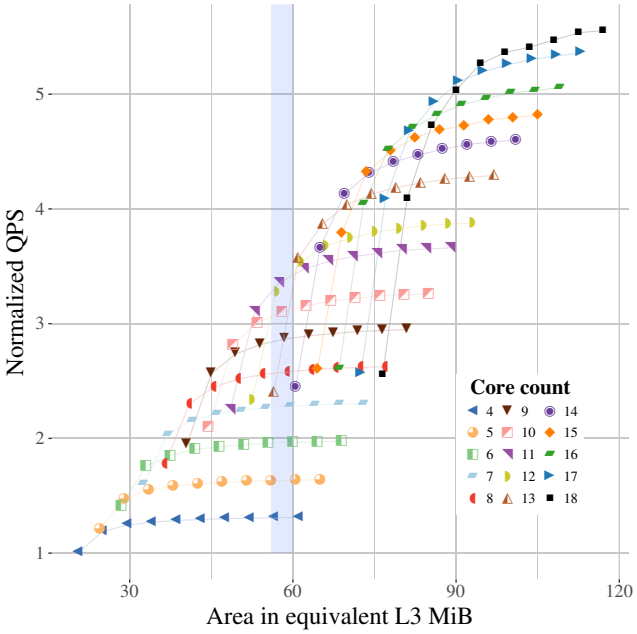


Figure 9: Search performance (QPS) vs. L3-equivalent area for various core count and cache size combinations.

IV. Optimized Memory Hierarchy for Search

A. Key Insights

Our analysis thus far demonstrates that search is fundamentally different from traditional benchmarks. While there is good thread-level parallelism, the memory hierarchy is a significant bottleneck. We find that current cache hierarchy decisions are effective in some ways and ineffective in other ways. Instruction working sets fit in the on-chip cache hierarchy, but at the L3 cache level. For data accesses, however, the first-level working set fits in relatively small shared caches, but the second-level working sets (driven primarily by heap data reuse) need capacities larger than that supported in current on-chip hierarchies.

These insights motivate two contrasting optimizations that together rethink cache hierarchy design for search:

1 Repurpose expensive, on-chip transistors currently yielding diminishing returns in L3 caches for higher-benefit cores. Effectively, we are rethinking the compute-to-cache *on-chip* system balance sweet spot.

2 Exploit the available locality in large heap working sets using cheaper and higher-capacity DRAM transistors, incorporated into a *latency-optimized* L4 cache. Effectively, we are rethinking the compute-to-cache *on-package* system balance sweet spot.

B. Trading L3 Cache Capacity for Cores

Our first step is to rightsize the portion of the processor die that is used for the shared L3 cache. Since throughput scales linearly with number of cores (see Figure 2a), we want to have a balanced system that maximizes the number of cores, keeping the L3 capacity at the minimum needed to avoid a significant increase in off-chip memory accesses.

Cache vs. Cores Trade-off: Figure 9 quantifies the trade-off between L3 cache capacity and core count. It shows the relative throughput (QPS) measured on our (`PLT1`) platform as we scale the number of cores used from 4 to 18 and the amount of total L3 cache capacity used from 4.5 to 45 MiB (150 measurements total). The x-axis represents "total area" occupied by cores and caches in terms of L3 capacity. We measure that 1 core and its private caches occupy roughly the same area as a 4 MiB slice of the L3 cache, as verified from die photos of the Haswell processor [7]. We limit L3 cache capacity using the CAT cache partitioning mechanism [20], which allows us to selectively enable 2 to 20 ways of the 20-way, 45 MiB L3 cache in increments of 1 way (2.25 MiB). For each core count, the left-most point is the one with 2 ways of L3 cache enabled, while the right-most point has 20 ways of the L3 cache enabled, with a spacing of 2 ways between points. A vertical line through this graph represents designs of equal area (cost). A horizontal line through this graph represents designs of equal QPS (performance).

There are three main observations from Figure 9. First, *some cache transistors currently used for L3 caches could be better used for cores.* For example, focus on the highlighted vertical line near 60 MiB. The default L3 ratio of 2.5 MiB/core leads to the 9-core/10-way design that performs much worse than the similar area, 11-core/6-way design ($\simeq$1.23 MiB/core). Second, *core count is not all that matters*. For example, all 18-core designs with less than 1 MiB/core perform worse than other designs with fewer cores. A corollary to this observation is that, unlike prior work [37], *the LLC must hold more than the instruction working set*, since any capacity less than 18 MiB is detrimental despite a 4 MiB code working set (§III-B). In other words, we must carefully rightsize the L3 cache.

Optimal L3 Capacity per Core: While Figure 9 provides actual system measurements on the benefits of trading L3 cache transistors for cores, it is limited to design points we can exercise in existing systems. To further reason about the ideal balance point between cache and cores, we incorporate the data in Figure 9 in a linear performance-area model that allows further design space exploration. We calculate area as $A = n(s+c)$, where n is the number of cores, s is the area cost of a core (roughly equal to 4 MiB of L3 cache [7]), and c is L3 cache capacity per core. We model performance as a linear function of core count (see Figure 2a), and use the data in Figure 9 to curve-fit this model at various cache sizes.

Figure 10 shows the estimated performance improvement for various designs relative to the `PLT1` baseline, 18-cores with 45 MiB L3 cache ($c = 2.5$ MiB per core). The groups show eight bars representing designs that reduce L3 capacity per core in increments of 0.25 MiB, with an equivalent

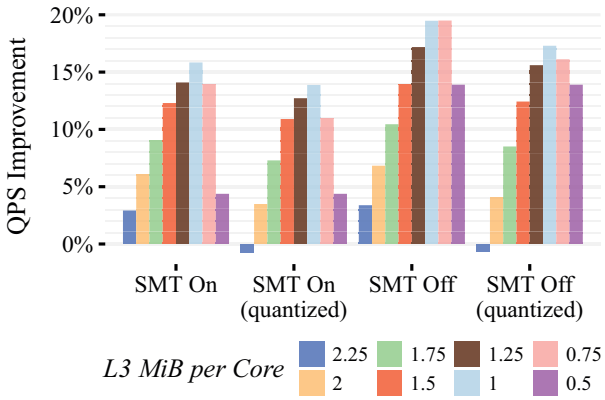


Figure 10: Search performance when trading cache capacity for cores.

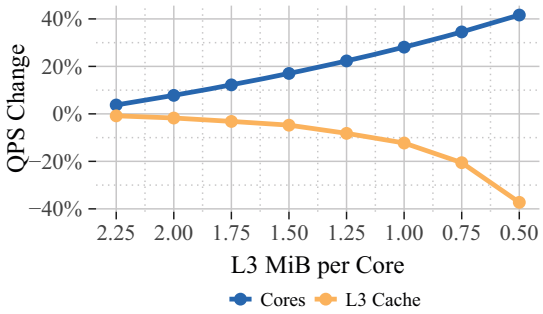


Figure 11: Trade-off between adding cores and removing L3 cache per core.

increase in core count appropriate to the number of transistors saved. The `non-quantized` groups of bars represent the upper-bound in performance in an ideal scenario where transistors can be reused for cores, even getting fractional cores. The `quantized` bars show a more realistic scenario when we quantize to whole-core granularity and, obviously, waste some transistors. In § IV-C, we use this extra slack in the floor plan to add an L4 controller.

The optimal performance is with $c = 1$ MiB/core and 23 cores, providing a 14% QPS increase over the default original design with $c = 2.5$ MiB/core, 18 cores, and SMT enabled. Figure 11 provides additional insights. The x axis corresponds to the amount of L3 cache transistors repurposed for cores. The negative and positive curves respectively illustrate the decrease in performance from the lost L3 capacity and the increase in performance from the equivalent-area new cores. The different slopes of the two lines clearly illustrate the insight motivating this optimizations. $c = 1$ MiB leads to the maximum difference between the QPS boost from multiple cores and the QPS loss from the increased L3 misses. Figure 10 also presents results when SMT is off (`SMT off`). The benefits are somewhat higher due to decreased cache pressure. However, they are not sufficient to offset the benefits of SMT (see Figure 2b).

Note that the L3 miss rate is also increased by conflicts within each bank when using CAT (since CAT reduces associativity to partition caches), as well as increased back-invalidations due to the inclusion property of the specific L3 cache. Our results are *conservative* in that eliminating these effects could further increase search throughput. We also evaluated per-query average and tail latency, and found it remained well within the margins of our service level objective.

C. Latency-optimized L4 Cache

The next opportunity in improving the memory hierarchy for search is to target the large data working set with additional cache capacity. Figure 6 showed that, while the large shard segment exhibits limited locality, there is enough locality in heap accesses that it can be almost fully captured with a 1 GiB cache. However, such a large SRAM cache is extremely difficult to build on the processor chip at this point. Using our previous area estimates, this would be the equivalent of 256 cores. In contrast to on-die SRAM, DRAM technologies such as eDRAM or RLDRAM [42], [52] offer lower cost for high capacities and competitive latencies. Therefore, we evaluate the addition of an off-chip but on-package latency-optimized large L4 cache built out of DRAM technologies.

L4 Cache Design and Implementation: Our L4 cache design optimizes for three constraints: First, it is optimized for low-latency access. As we showed in Figure 8b, search performance is very sensitive to L3 AMAT. In contrast, L4 throughput is not a primary consideration because search exhibits low MLP and there is still headroom for bandwidth in main memory (see §II-D). Second, we optimize the L4 cache for area to make its large capacity practical. Finally, we optimize for low design complexity which helps reduce latency and cost. The latter is particularly important for warehouse-scale environments [3].

Amongst a large body of studies on large L4 caches [19], [23], [24], [36], [43], [46], [50], [57], we use the latency-optimized Alloy cache as the starting point for our design [46]. Similar to Alloy, we store a tag and data pair within the same DRAM row (page) in order to access them with a single DRAM command. We also use a direct-mapped organization to reduce cache hit latency and to better exploit spatial locality by mapping consecutively-addressed cache lines to the same DRAM row. The consequent impact of reduced associativity is relatively minor and modeled in our results below. Unlike the Alloy cache, we use on-package eDRAM instead of ordinary DRAM chips in order to further reduce access latency. We also perform L4 cache lookups in

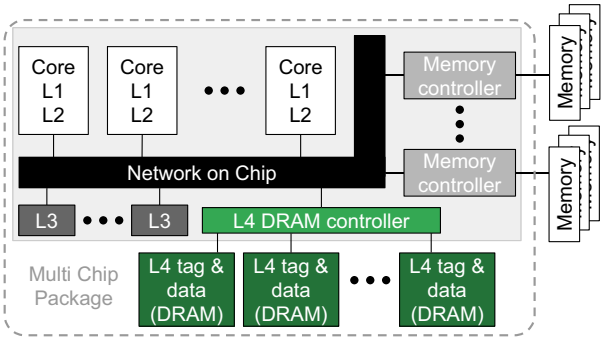


Figure 12: The proposed L4 cache design based on eDRAM [42]. Similar to Alloy cache [46], we place tags and data in the same eDRAM row and use a direct-mapped organization.

parallel with main memory scheduling. If the request hits in the L4, the operation is canceled before it is served by main memory DRAM. This optimization avoids any latency penalty for accesses that miss in the L4 cache.

While physically located on-package, our L4 cache operates as a virtual memory-side cache, essentially acting as a large victim cache for L3 cache evictions. This avoids the complexity of managing coherence in the L4 cache, especially because the large L4 capacity will likely stress the capabilities of coherence filters. The trade-off for the memory side placement is that, in a multi-socket system, we have to pay NUMA penalties for cached data in remote sockets. We account for this in our model below. Our L4 design also uses the same cache block size as the L3 in order to simplify its operation as a victim cache. Recent studies advocate larger L4 lines (e.g., 2 KiB) to reduce tag overhead and exploit spatial locality exposed by the longer residence in the L4 cache [23], [24]. However, these proposals assume that the program counter for each hardware thread is delivered to the L4 cache on each access in order to predict access patterns with high accuracy. This feature is complex and not yet popular in contemporary CPU designs [57].

Figure 12 shows how our design can be implemented using eDRAM dies on a multi-chip package (MCP). Similar approaches have been used on other commercial L4 cache designs from Intel [16] and IBM [52], but with smaller capacities. The processor and eDRAM dies are connected through a high-bandwidth link, such as Intel's OPIO [32] or IBM's Agnostic Memory Channel [52]. The close proximity enabled by on-package connections reduces latency and power. Architecture-specific floor planning and packaging optimizations can minimize the area and latency overheads of interfacing to the core interconnects, and reduce the length of the I/O traces for improved latency. The latter can be a challenge for larger L4 cache sizes that use multiple eDRAM dies. However, 128 MiB eDRAM dies are already available and memory trends point to increased capacity per die. Our analysis indicates that L4 hit latency of roughly 40 ns is possible for 1 GiB capacity, which is consistent with measurements from commercial L4 designs [16], [52]. Our area estimates show that, with careful floor-planning, the increase in the processor die area for the L4 controller and interface is less than 1%.

L4 Cache Evaluation: We build on the optimized design from §IV-B (23-core processor chip with 1 MiB/core of L3 cache) and evaluate the benefits from adding our L4 cache. Note that since we start with the `quantized (SMT on)` sweet spot from §IV-B, the extra area left over from rounding down the number of cores is more than sufficient for the small die area increase for the L4 cache controller. So, the processor is still iso-area relative to the 18-core baseline. We use the same performance model as before from §III-B, but $AMAT_{L3}$ now accounts for the L4 hit rate and L4 hit latency. We assume 40 ns for the L4 hit latency but examine sensitivity further below. To determine the L4 hit rate, we simulate a direct-mapped, 64-byte block L4 cache for capacities ranging from 64 MiB to 8 GiB.

Figure 13 again plots the hit rate and MPKI changes, but this time assumes a fixed L3 (23 MiB) and varies the L4 capacity. The L4 cache is quite effective in capturing additional locality, with a 1 GiB capacity achieving most of the benefits for the heap. The additional breakdown of cache misses per data type shows that, as expected, a majority of the remaining misses come from the shard data. At the higher capacities, the heap hit rate trends close to 90%. We can also see the effects of conflict misses due to the direct-mapped design by noting the decreased hit rates compared to the large theoretical L3 cache of equal capacity from Figure 6.

The first set of bars in Figure 14 summarizes the cumulative QPS improvement when a 40 ns L4 cache and the rebalancing from §IV-B (5 additional cores and 1 MiB/core of L3 capacity) are applied to the 18-core `PLT1` baseline. In all cases, we have a 14% benefit from rightsizing the L3 cache and having 5 additional cores. The L4 cache achieves an *additional* 8% to 14% throughput improvement over the design of §IV-B. The combined benefit of rightsizing the L3 cache and a 40 ns, 1 GiB L4 cache is a *27% improvement* over the `PLT1` baseline. An L4 cache capacity of 8 GiB (not pictured) improves the net benefits to 30%. Note that there are synergistic benefits from the two optimizations. Intuitively, having a smaller L3 cache means that the L4 cache will service hotter data than it otherwise would, thus improving its hit rate by roughly 10% for all configurations we evaluated.

Figure 14 also presents the sensitivity of our results to various assumptions. The second set of bars presents the net improvements for a more pessimistic set of assumptions about the L4 cache – a 60 ns hit latency and a 5 ns additional miss penalty. The miss latency is due to no overlapping

between L4 tag check and main memory access scheduling. The results show correspondingly lower savings, but still, a 1 GiB L4 cache improves performance more than 23% over the baseline. We also evaluated the impact of conflict misses by modeling a fully associative L4 cache. A 1 GiB capacity improves over the direct-mapped cache by one percentage point, validating that our use of a direct-mapped cache to reduce complexity was not actually detrimental. The final set of bars evaluates sensitivity to the impact of two generational trends we have been seeing in the fleet; increased memory latencies across successive platform generations, and increased last-level cache miss rates with growing shard sizes. The `future` configuration increases memory latency and L3 cache misses by 10%. Our results show that the benefit of our redesigned cache hierarchy for search is even higher. Our 1 GiB baseline now achieves 38% net improvement over the equivalent baseline.

Power and Energy: Our design goal is to maximize the utility of on-die transistors and identify the optimal system balance from a performance perspective. However, we also expect benefits in terms of energy (joules per query).

First, due to the near-perfect performance scaling with additional cores (see Figure 2a), our cache-for-cores trade-off is energy-neutral. To support this, we measured processor power when scaling the number of enabled cores from 4 to 18 and observed an expected linear increase.[2] The linear increase in performance and power cancel each other out in terms of energy efficiency.

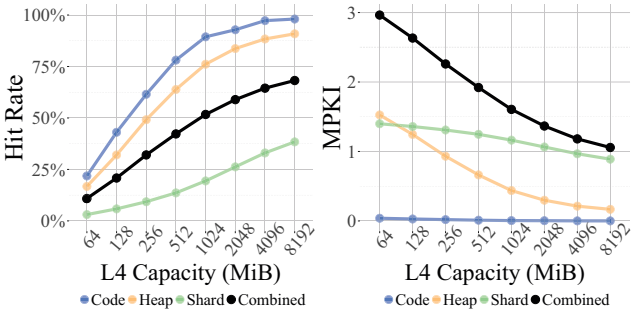
Second, our 1 GiB L4 cache provides an additional performance boost, while slightly reducing power consumption. Since the L4 filters approximately 50% of DRAM accesses (see Figure 13a) and energy per access for eDRAM is significantly lower than for DRAM [10], [54], the L4 reduces power consumption in the memory system. Note, however, that because core power typically dominates memory power [3], the power benefits of the L4 cache are small. Most of the energy benefits from the L4 cache stem from its impact on performance.

From a power perspective, we observed that each core contributed 3.77% of the total baseline socket power of `PLT1`. For our optimal configuration of 1 MiB/core of L3 cache which yielded five additional cores, this translates to an 18.9% socket power increase (or approximately 27 watts) for a 27% performance improvement. This is a feasible design point as it is within 3.8% of published TDP of commercial processors [7]. For completeness, an *iso-power* configuration of 18 cores with 1 MiB/core L3 cache would reduce the core and cache area by 23% while maintaining performance within 5%.

V. Discussion

Further benefits from an L4 cache: While not quantified in our results, an L4 cache can have further benefits discussed in prior studies [52]. Specifically, it can be used as a write buffer in order to reduce the write-to-read delay (`tWRT`) of DRAM and decrease effective DRAM latency for reads (faster L4 misses). The L4 can also be used as an aggressive prefetch buffer to avoid pollution at other cache layers.

[2]We could not power gate portions of the L3 to account for the reduced capacity per core. Hence, our measurements are pessimistic.



(a) L4 hit rate vs. size (b) L4 MPKI vs. size

Figure 13: L4 capacity sweep.

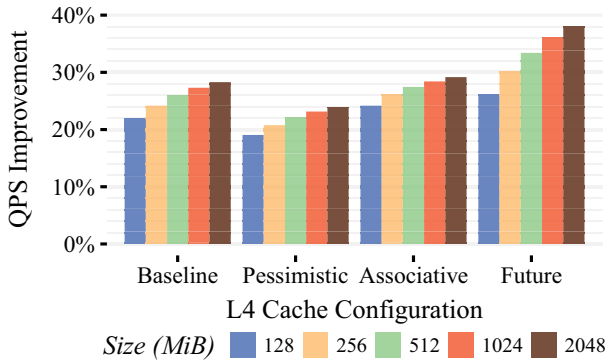


Figure 14: Performance improvement when combining an L4 cache with trading cache for cores.

Other search workloads: § IV evaluates the two proposed optimizations using traces from the `S1` service. While there are small variations across search workloads, the benefits from the two optimizations are consistent across services. Similarly, while we focus on `PLT1`, `PLT2` will show qualitatively-similar results.

Alternate SoC optimizations: One can consider alternative uses for the transistors saved by rightsizing the L3 cache. For example, we could increase the L2 cache capacity, albeit with slight increase in latency, and then add further cores. Similarly, search could likely benefit from prefetching techniques for shard and instruction accesses beyond the simple strided prefetching used in our baseline [1], [14], [28], [30], [39], [59]. There may also be further optimization opportunities if we co-optimize all levels of the memory hierarchy (L2, L3, L4, DRAM, and potentially NVM). We

leave these opportunities to future work.

Split I/D L2 caches: A large fraction of stall cycles is due to instruction accesses that miss in the L2 cache but hit in the L3 cache (see §II-F). Given that the L2 cache is a unified cache, one potential optimization is to split I and D caches at the L2 level or leverage a CAT-like partitioning of the L2 cache for instructions and data. Our analysis shows, however, that is unlikely to be beneficial since the improved L2 hit rate for instructions is offset by the decrease in L2 hit rate for data.[3]

VI. Related Work

OLDI workload studies: Kozyrakis et al. [31] analyzed email, search (Bing), and analytics applications from Microsoft, focusing primarily on balanced server design. Kanev et al. [27] characterized the performance of general datacenter workloads exploiting the fleet-wide profiling infrastructure at Google [48]. A 1998 study on the AltaVista search engine showed its memory behavior to resemble that of decision support workloads (DSS) [4], where a modestly-sized L2 cache contains most of the working set, and memory bandwidth is mostly under-utilized [55]. A few studies have examined the use of low-power, simpler cores for web search – for Nutch [33] and for Bing [47]. More recently, the Catapult project has studied FPGA acceleration for the ranking part of search [45]. Compared to these studies, our work is distinguished by its focus on in-depth analyses and insights on the memory hierarchy, including simulations and what-if analyses, and its focus on Google web search, the largest commercial search service today, about which data has previously not been published.

Lotfi-Kamran et al. [37] argued that last-level cache capacities larger than the instruction footprint (a few MiB) provide limited benefits for the CloudSuite benchmark, and proposed on-chip "pods" which contain small LLCs and run their own operating systems. Our work validates their observation that LLCs can be better provisioned. However, our production search workload benefits from shared caches much larger than the instruction footprint (§IV-B) due to its large shared heap and shard, and we argue for *increasing* overall cache capacity rather than reducing it. Furthermore, we find that replicating the instruction footprint on each pod would be an inefficient use of on-chip transistors. Overall, our proposed design improves performance density for search-like OLDI workloads while also allowing requests to share state.

Large DRAM caches: There is a large body of work on large DRAM caches (both block- and group-based) [19], [23], [24], [36], [43], [46], [50], [57]. Loh and Hill [36] proposed a block-based design that co-locates tags and data on the row of an off-chip DRAM cache, adding a special MissMap structure to hold cache miss information. The high latency of MissMap accesses and highly-set-associative tags were addressed in Alloy cache using a direct-mapped allocation to reduce latency and a memory access predictor to save main memory bandwidth [46]. Jevdjic et al. [24] focused on a relatively small number of recurrent access patterns within pages and advocated the page-based Footprint cache which addresses the inherent problem of main memory bandwidth amplification by fetching the portion of a page that is likely to be accessed in the near future. The Unison cache combines the primary ideas of the Footprint and Alloy designs [23]. The Fat cache [57] work argues against the access pattern prediction suggested by these studies citing the impracticality of delivering program counter information from the cores to the DRAM caches. Orthogonal to these designs, Meza et al. [43] and ATCache [19] suggested caching the tags of recently-accessed DRAM cache lines in a smaller, on-chip structure to further reduce DRAM cache hit latency.

Several commercial systems include large DRAM caches, mainly optimized for bandwidth. Knights Landing [51] includes 16 GiB on-package memory (MCDRAM), which can be configured as a memory-side direct-mapped cache that stores both tag and data [22]. Hammarlund et al. discussed an implementation of Haswell which used an optional 128 MiB eDRAM cache in order to provide high bandwidth to graphics units integrated with cores [16]. POWER8 [52] adds a 16 MiB eDRAM buffer cache per Centaur memory-buffer chip. It is located on the memory side but its low capacity allows for lower access time compared to the Knights Landing cache. As discussed in §IV-C, our large DRAM cache design builds on this rich body of prior work, and optimizes for a balance of low latency, low cost, and high capacity, in the context of an on-package eDRAM cache.

VII. Conclusions

Though OLDI services such as search comprise an increasingly growing fraction of datacenter cycles, there is a lack of microarchitectural analysis and optimization, particularly in the context of production workloads. This paper provided the first in-depth study of the memory behavior of large-scale, production search services, including Google-fleet-wide longitudinal studies, simulations, and what-if analyses of the world's largest search engine serving real traffic.

Our detailed microarchitectural characterization using fleet-wide measurements showed that search exhibits significant differences from popular benchmarks, including much higher MPKIs for branches, L2 instruction misses, and L3 data misses. We presented new insights on how search stresses the cache hierarchy, relating them back to software. Our detailed quantitative analysis identified

[3]Unlike with other optimizations discussed so far, our analysis can only capture changes in L2 miss rate and cannot conclusively identify the impact on IPC for L2 optimizations.

that current L3 caches of a few tens of MiB are over-provisioned for search, and are inefficient in their use of critical on-die transistors. At the same time, and contrary to conventional wisdom, there is significant reuse in shared heap accesses that can be captured with much larger last-level caches than what we can fit on a processor die. Based on these observations, we evaluated a new state-sharing-enabled SoC cache hierarchy optimized for OLDI workloads like search that trades off some of the L3 cache capacity for higher performance processing cores, and added a GiB-capacity, latency-optimized L4 cache based on on-package eDRAM technology. Compared to a state-of-the-art baseline, our proposed design achieves a 27% performance improvement. The improvements are even higher (38%) with next-generation memory hierarchy parameters.

Our results are notable in that they present an opportunity for an entire Moore's law generation of performance benefits at the same technology node. But perhaps, more importantly, our work demonstrates the high potential from new classes of *SoCs specifically designed* for OLDI workloads. Looking ahead, we believe we have only scratched the surface of what is possible. Our analysis highlights several additional opportunities for future research in this area (shard memory misses, instruction misses, branch stalls, new system balance ratios, and new system evaluation methodologies).

VIII. Acknowledgments

This work reflects the combined efforts of numerous people across the search and platforms teams at Google. We would specifically like to thank Stephane Eranian, Oleg Slezberg, David Lo, Changkyu Kim, Greg Johnson, and Jichuan Chang for their guidance and help in collecting some of the data for this paper. We are grateful to Junwhan Ahn, Luiz Barroso, Bob Cypher, Nadav Eiron, Rama Govindaraju, Urs Hölzle, Beckett Madden-Woods, Dan Meredith, and Amin Vahdat for their feedback.

References

[1] T. M. Aamodt, P. Chow, P. Hammarlund, H. Wang, and J. P. Shen, "Hardware Support for Prescient Instruction Prefetch," in *HPCA*, 2004.

[2] L. A. Barroso, J. Dean, and U. Holzle, "Web Search for a Planet: The Google Cluster Architecture," *IEEE Micro*, vol. 23, no. 2, pp. 22–28, 2003.

[3] L. A. Barroso, J. Clidaras, and U. Hölzle, "The Datacenter as a Computer: An Introduction to the Design of Warehouse-scale Machines," *Synthesis Lectures on Computer Architecture*, vol. 8, no. 3, pp. 1–154, 2013.

[4] L. A. Barroso, K. Gharachorloo, and E. Bugnion, "Memory System Characterization of Commercial Workloads," in *ISCA*, 1998.

[5] S. Bell *et al.*, "TILE64 - Processor: A 64-Core SoC with Mesh Interconnect," in *ISSCC*, 2008.

[6] P. Bergner *et al.*, *Performance Optimization and Tuning Techniques for IBM Power Systems Processors Including IBM POWER8*. IBM Redbooks, 2015.

[7] B. Bowhill *et al.*, "The Xeon® processor E5-2600 v3: A 22nm 18-core product family," in *ISSCC*, 2015.

[8] S. Brin and L. Page, "The Anatomy of a Large-scale Hypertextual Web Search Engine," in *Proceedings of the Seventh International Conference on World Wide Web*, 1998.

[9] A. M. Caulfield *et al.*, "A Cloud-Scale Acceleration Architecture," in *MICRO*, 2016.

[10] M.-T. Chang, P. Rosenfeld, S.-L. Lu, and B. Jacob, "Technology Comparison for Large Last-level Caches (L3cS): Low-leakage SRAM, Low Write-energy STT-RAM, and Refresh-optimized eDRAM," in *HPCA*, 2013.

[11] J. Dean, "Challenges in Building Large-scale Information Retrieval Systems: Invited Talk," in *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, 2009.

[12] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *OSDI*, 2004.

[13] M. Ferdman *et al.*, "Clearing the Clouds: a Study of Emerging Scale-out Workloads on Modern Hardware," in *ASPLOS*, 2012.

[14] M. Ferdman, C. Kaynak, and B. Falsafi, "Proactive Instruction Fetch," in *MICRO*, 2011.

[15] A. S. Foundation, "Solr." [Online]. Available: http://lucene.apache.org/solr/

[16] P. Hammarlund *et al.*, "Haswell: The Fourth-Generation Intel Core Processor," *IEEE Micro*, vol. 34, no. 2, pp. 6–20, 2014.

[17] J. Hauswald *et al.*, "Sirius: An Open End-to-End Voice and Vision Personal Assistant and Its Implications for Future Warehouse Scale Computers," in *ASPLOS*, 2015.

[18] J. L. Henning, "SPEC CPU2006 Benchmark Descriptions," *ACM SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[19] C.-C. Huang and V. Nagarajan, "ATCache: Reducing DRAM Cache Latency via a Small SRAM Tag Cache," in *PACT*, 2014.

[20] Intel, "Improving Real-Time Performance by Utilizing Cache Allocation Technology." [Online]. Available: http://www.intel.com/content/www/us/en/communications/cache-allocation-technology-white-paper.html

[21] Intel Corporation. (2017) Intel Xeon Processor E5 v3 Product Family. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/specification-updates/xeon-e5-v3-specification-update.pdf

[22] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Elsevier Science, 2016.

[23] D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison Cache: A Scalable and Effective Die-Stacked DRAM Cache," in *MICRO*, 2014.

[24] D. Jevdjic, S. Volos, and B. Falsafi, "Die-stacked DRAM caches for Servers: Hit Ratio, Latency, or Bandwidth? Have It All with Footprint Cache," in *ISCA*, 2013.

[25] V. Jiménez, R. Gioiosa, F. J. Cazorla, A. Buyuktosunoglu, P. Bose, and F. P. O'Connell, "Making Data Prefetch Smarter: Adaptive Prefetching on POWER7," in *PACT*, 2012.

[26] N. P. Jouppi *et al.*, "In-Datacenter Performance Analysis of a Tensor Processing Unit," in *ISCA*, 2017.

[27] S. Kanev *et al.*, "Profiling a Warehouse-scale Computer," in *ISCA*, 2015.

[28] C. Kaynak, B. Grot, and B. Falsafi, "SHIFT: Shared History Instruction Fetch for Lean-core Server Processors," in *MICRO*, 2013.

[29] C. Kim *et al.*, "Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-core CPUs," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1378–1389, 2009.

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

[31] C. Kozyrakis, A. Kansal, S. Sankar, and K. Vaid, "Server engineering insights for large-scale online services," *IEEE Micro*, vol. 30, no. 4, pp. 8–19, Jul. 2010.

[32] N. Kurd *et al.*, "Haswell: A family of IA 22nm processors," in *ISSCC*, 2014.

[33] K. Lim, P. Ranganathan, J. Chang, C. Patel, T. Mudge, and S. Reinhardt, "Understanding and Designing New Server Architectures for Emerging Warehouse-Computing Environments," in *ISCA*, 2008.

[34] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards Energy Proportionality for Large-scale Latency-critical Workloads," in *ISCA*, 2014.

[35] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving Resource Efficiency at Scale," in *ISCA*, 2015.

[36] G. H. Loh and M. D. Hill, "Efficiently Enabling Conventional Block Sizes for Very Large Die-stacked DRAM Caches," in *MICRO*, 2011.

[37] P. Lotfi-Kamran *et al.*, "Scale-out Processors," in *ISCA*, 2012.

[38] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005, pp. 190–200.

[39] C.-K. Luk and T. C. Mowry, "Cooperative Prefetching: Compiler and Hardware Support for Effective Instruction Prefetching in Modern Processors," in *MICRO*, 1998.

[40] K. T. Malladi, B. C. Lee, F. A. Nothaft, C. Kozyrakis, K. Periyathambi, and M. Horowitz, "Towards Energy-proportional Datacenter Memory with Mobile DRAM," in *ISCA*, 2012.

[41] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power Management of Online Data-intensive Services," in *ISCA*, 2011.

[42] M. Meterelliyoz *et al.*, "2nd Generation Embedded DRAM with 4X Lower Self Refresh Power in 22nm Tri-Gate CMOS Technology," in *VLSI*, 2014.

[43] J. Meza, J. Chang, H. Yoon, O. Mutlu, and P. Ranganathan, "Enabling Efficient and Scalable Hybrid Memories Using Fine-granularity DRAM Cache Management," *IEEE Computer Architecture Letters*, vol. 11, no. 2, pp. 61–64, 2012.

[44] S. Novakovic, A. Daglis, E. Bugnion, B. Falsafi, and B. Grot, "Scale-out NUMA," in *ASPLOS*, 2014.

[45] A. Putnam *et al.*, "A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services," in *ISCA*, 2014.

[46] M. K. Qureshi and G. H. Loh, "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-tags with a Simple and Practical Design," in *MICRO*, 2012.

[47] V. J. Reddi, B. Lee, T. Chilimbi, and K. Vaid, "Web Search Using Mobile Cores: Quantifying and Mitigating the Price of Efficiency," in *ISCA*, 2010.

[48] 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, 2010.

[49] E. Schurman and J. Brutlag, "The User and Business Impact of Server Delays, Additional Bytes, and HTTP Chunking in Web Search," in *Velocity*, 2009.

[50] J. Sim, G. H. Loh, H. Kim, M. O'Connor, and M. Thottethodi, "A Mostly-Clean DRAM Cache for Effective Hit Speculation and Self-Balancing Dispatch," in *MICRO*, 2012.

[51] A. Sodani *et al.*, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, Mar 2016.

[52] W. Starke *et al.*, "The Cache and Memory Subsystems of the IBM POWER8 Processor," *IBM Journal of Research and Development*, vol. 59, no. 1, pp. 3–1, 2015.

[53] J. Stuecheli, "Power8," in *Hot Chips*, 2013.

[54] S. Thoziyoor, J. Ahn, M. Monchiero, J. B. Brockman, and N. P. Jouppi, "A Comprehensive Memory Modeling Tool and its Application to the Design and Analysis of Future Memory Hierarchies," in *ISCA*, 2008.

[55] P. Trancoso, J.-L. Larriba-Pey, Z. Zhang, and J. Torrellas, "The Memory Performance of DSS Commercial Workloads in Shared-Memory Multiprocessors," in *HPCA*, 1997.

[56] V. Viswanathan, "Disclosure of H/W prefetcher control on some Intel processors." [Online]. Available: https://software.intel.com/en-us/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors

[57] S. Volos, D. Jevdjic, B. Grot, and B. Falsafi, "Fat Caches for Scale-out Servers," *IEEE Micro*, vol. 37, no. 2, pp. 90–103, Mar. 2017.

[58] A. Woodruff, P. M. Aoki, E. Brewer, P. Gauthier, and L. A. Rowe, "An Investigation of Documents from the World Wide Web," *Computer Networks and ISDN Systems*, vol. 28, no. 7, pp. 963–980, 1996.

[59] C.-J. Wu, A. Jaleel, M. Martonosi, S. C. Steely, Jr., and J. Emer, "PACMan: Prefetch-aware Cache Management for High Performance Caching," in *MICRO*, 2011.

[60] A. Yasin, "A Top-Down Method for Performance Analysis and Counters Architecture," in *ISPASS*, 2014.