Simple Cache Partitioning for Networked Workloads

Thomas Kim*, Sol Boucher*, Hyeontaek Lim*, David G. Andersen*, and Michael Kaminsky†

October 2017
CMU-CS-17-125

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

*Carnegie Mellon University Computer Science Department
†Intel Labs

This research was funded in part by the Intel Science and Technology Center for Cloud Computing.

Keywords: cache partitioning, isolation, resource utilization, tail latency, multi-tenancy

Abstract

Modern datacenters often run a mix of latency-sensitive and throughput-oriented workloads, which are consolidated onto the same physical machines to improve resource utilization. Performance isolation under these conditions is an important component in satisfying service level objectives (SLOs). This paper examines *cache partitioning* as a mechanism to meet latency targets. Cache partitioning has been studied extensively in prior work; however, its effectiveness is not well understood for modern datacenter applications that enjoy recent improvements in low-latency networking, such as stack bypass and faster underlying technologies (e.g., DPDK, InfiniBand). These applications have tight tail latency SLOs of tens to hundreds of microseconds. We find that cache partitioning is effective in achieving performance isolation among diverse workload mixes for such environments, without requiring complex configurations or online controllers. On our modern multi-core server using Intel Cache Allocation Technology, we show that cache partitioning can protect the performance of latency-sensitive networked applications from local resource contention. Our code is publicly available on Github.

1 Introduction

The efficiency of datacenter and cloud services hinges on achieving two partially-conflicting objectives: Maintaining high overall machine utilization while meeting per-service throughput and latency SLOs. Aggressively consolidating services on to a smaller set of nodes is good for the former and bad for the latter; providing independent hardware for each service suffers the opposite tradeoff.

Recent literature highlights the importance of low-latency responses in complex distributed systems, such as Google and Bing's search indexers, or assembling complex, multi-component views such as Facebook and Amazon's user front pages [10]. These applications typically perform tens to thousands of lookups for each user request, such that the overall user-visible response time depends both on the *slowest* chain of dependent responses, as well as the 99.9th percentile latency [19, 34, 25, 20].

In this paper, we demonstrate that *cache partitioning* is beneficial for workload consolidation of throughput-oriented background processing jobs with latency-critical networked applications. To be concrete, in this work we focus on "mite" workloads such as would be served by key-value stores or lock servers, and on "contender" workloads such as training machine learning models, advanced image re-compression, or big data analytics. For the cache partitioning itself, we use Intel Cache Allocation Technology (CAT), available in recent processors [16].

The high speed of modern stack-bypassing networking technologies, such as 10/40/100 Gbps Ethernet and InfiniBand, combined with a userland I/O stack, makes the cache-related effects of sharing the CPU a major factor in a mite's response time. As we show, cache contention can increase the 99.9th percentile tail latency by a factor of 5, from 55 µs to 274 µs for a key-value store serving requests at 5 Mops. The appropriate use of cache partitioning can nearly eliminate this penalty while allowing contenders to consume all of the (otherwise) idle CPU.

Our results also show that cache partitioning provides strong and effective performance isolation for diverse workload mixes in low-latency environments with strict SLOs. Unlike previous work, we examine simple configurations that avoid online controllers or complex schedulers that are less suited to very low latency environments. Finally, our results highlight two additional insights: partitioning has diminishing—or even negative—returns that can harm tail latency and throughput in certain cases, and cache partitioning can provide more effective isolation than just reducing the number of cores available to contender processes.

2 Background and Related Work

2.1 Workload Consolidation

From early time-sharing systems to modern services based on virtual machines (e.g., VMware, Amazon EC2, Google Compute Engine) and containers (e.g., Kubernetes), running many applications on a single physical host has been a key technique for improving resource utilization, and, thereby, the cost-effectiveness of large-scale computing [3].

Current best practices for large-scale services specify service-level objectives, or SLOs, on the

response time, throughput, and uptime of running services. SLOs are critical for ensuring the performance of large, complex systems composed of individual sub-services. In these settings, there is no free lunch: Resource consolidation pits overall system efficiency (consolidating all services onto a few highly-loaded physical hosts) against low latency and predictable performance (physically isolating each service).

In this paper, we examine performance isolation techniques to permit the collocation of low-latency tasks ("mites") and high-throughput, batch processing tasks ("contenders"). Mites have strict latency SLOs. In many common situations, the mite's 99.9th percentile latency is of paramount concern. For high fan-out services, such as searching within a sharded index or creating a user's Facebook home page, a single request by a user may involve communicating with hundreds or thousands of nodes [30]. This fan-out means that the user request's latency is determined by the *longest* response time of any contacted node; with 100s or 1000s of requests, the per-node 99.9th percentile response latency is the *de facto* lower bound on end-to-end latency.

Unlike mites, contenders behave elastically: they derive more utility the more throughput they obtain. Typical approaches support mites by overprovisioning the resources promised to them, thus achieving low overall utilization, because slight increases in load can rapidly degrade the mites' 99.9th percentile latency. This approach wastes resources: the contenders' elasticity means that, were a perfect isolation mechanism to exist, a scheduling system could "soak up" all spare capacity in a cluster by packing contenders into the resources not used by mites.

We consider as contender applications CPU-intensive machine learning training using Google's TensorFlow system [33], background lossless re-compression of JPEG images using Dropbox's Lepton re-compressor [24], and I/O-intensive big data analytics using Apache Hadoop [1].

2.2 Modern Low-Latency Applications

Advances in networking have made it possible for latency-critical applications to handle requests in tens of microseconds, a boon for high fan-out designs, high-performance computing, and computational finance applications. These advances cross hardware (cost-effective and fast NICs and networks with microsecond processing times, such as InfiniBand) and software (OS stack bypass techniques, such as DPDK [11]). The consequence is very high throughput for small messages and latency an order of magnitude smaller than what was typical only a few years ago. Such networks have become common in modern Internet service datacenters.

Latency requirements in the microseconds make performance isolation more important and challenging. Kernel context switches are too slow and often incompatible with the "spin and wait for packets" approach used in low-latency stack-bypass frameworks. Even TLB and cache flushes from a context switch may be sufficient to double end-to-end application latency. As a consequence, lower-overhead isolation and sharing mechanisms are needed.

2.3 Software-Based Performance Isolation

Cache-aware software-based performance isolation has attempted to address the isolation challenge of shared caches via both scheduling and direct cache partitioning; however, the limitations of both techniques have been significant enough to motivate a recent shift to hardware solutions.

Some prior work seeks to manage cache contention by changing the per-machine scheduling policy. The *cache-fair* scheduling algorithm adjusts the time slice of each thread based on its instructions per cycle (IPC) [12]. This approach, however, suffers context switching overhead and shares resources at millisecond, not microsecond, granularity. The *Elfen* scheduler executes a batch processing task on a simultaneous multithreading (SMT) core while no latency-critical task is running on the same physical core [35], but its target SLO (100 ms of 99th percentile latency) is 3 orders of magnitude higher than ours.

Other systems instead focus on scheduling at the cluster level. In this category is *Bubble-Up*, which predicts the performance penalty of executing applications in parallel by measuring how much pressure a task creates on the memory subsystem and how sensitive it is to the pressure generated by other tasks [28]. To achieve good resource utilization, this approach assumes the availability of many tasks with diverse resource requirements.

Various software systems have addressed the shared cache problem more directly; this usually involves *page coloring*, a mechanism for implementing cache partitioning purely in software. By selectively allocating physical pages based on the common bits in their physical page number and cache index (the pages' "colors"), such programs can control which regions of a cache particular tasks' memory can map to [4, 36]. Unfortunately, this approach is incompatible with the 2 MiB hugepages and 1 GiB superpages employed by modern memory-intensive applications: such pages may not have any bits in common between the physical page number and cache index. Recent work proposes special hardware support to overcome this limitation [9].

2.4 Hardware Cache Partitioning

Cache partitioning regulates the cache-level interference among application tasks [32]. The motivation for this technique is that consolidation of multiple tasks on the same CPU causes contention for shared resources, in particular shared cache, and this contention can have a significant effect on the performance of tasks. Cache partitioning allows changing which portion of shared CPU cache is available to individual tasks. A controller using cache partitioning can prevent tasks from evicting other tasks' data from the cache by assigning them disjoint cache partitions. Alternatively, by assigning overlapping partitions, the controller can enable opportunistic cache sharing.

A common form of cache partitioning is *way partitioning* [6]. Modern CPUs use a set-associative cache design to implement approximate LRU replacement. For example, Intel Xeon E5-2683 v4 has 20-way set associative last-level cache (LLC), which allows 20 different cache lines to share the same cache bucket. Way partitioning exploits this existing structure of the cache; it treats cache ways as the unit of cache partitions, realizing cache partitioning by allocating subsets of cache ways to processes.

Way partitioning is applicable to a wide range of applications. It does not require explicit modifications to the application software because it only alters the hardware-level cache eviction protocol; it is even possible to change a task's partition dynamically [27, 38]. In contrast, software-only techniques such as page coloring require precise control over how application memory is allocated and accessed in the physical address space, and thus experience high overhead to change partitioning frequently [36].

2.5 Intel Cache Allocation Technology

Previous conventional wisdom said that cache partitioning was complex and costly to implement [12], and most earlier studies of cache partitioning operated entirely in simulation [7, 21]. Some recent CPUs, however, implement hardware-based cache partitioning.

Intel Cache Allocation Technology (CAT) is the first commodity hardware implementation of cache partitioning, and is available in certain recent Xeon CPUs [15]. CAT is implemented as way partitioning [7], and is currently operate on the last-level cache (LLC). It works by limiting the cache ways from which a thread is allowed to evict cache lines, but does not prevent reads from hitting in ways outside the thread's current allocation.

The CAT software interface is centered around the notion of Classes of Service (CoSes), numeric identifiers used to group jobs that are assigned the same partition of cache. Each logical core has a model-specific register (MSR) that the OS or VMM can use to assign a CoS to the process currently running on that core. Each CoS has an associated Capacity Bitmask (CBM) that specifies which cache ways the processes in that group can evict lines from. Like CoSes, CBMs are stored in MSRs, but are CPU-wide instead of being local to each core. For instance, the CAT implementation on the Intel Xeon E5-2683 v4 supports 16 CoSes, each of which can be associated with an arbitrary CBM. There are restrictions placed on these CBMs: they must specify a single logically consecutive set of cache ways and must include at least one way.

CAT-enabled CPUs also provide Cache Monitoring Technology (CMT), which allows hardware performance counter-style monitoring of other important LLC performance metrics. Similarly to CAT, CMT exposes a software interface featuring per-core Resource Monitoring IDs (RMIDs), which are analogous to CoSes. CMT presently comprises two subfeatures: LLC occupancy measurement and Memory Bandwidth Monitoring (MBM). In this work, we make use of the latter feature, which provides the real-time local (NUMA node managed by the current CPU) and total memory bus bandwidth [17, Section 17.15.5.2].

2.6 Performance Isolation using CAT

Recent studies have begun to exploit CAT to achieve performance isolation. Prior work, however, makes it hard to reason about their effectiveness under modern low-latency workloads and realistic background tasks. They often focus on local processing with no network I/O, relatively high latency SLOs in milliseconds, and/or limit their studies to synthetic workloads, which are far from high-speed networked workloads running on low-latency networks. Our work clarifies the effectiveness and pitfalls of hardware cache partitioning for recent low-latency workloads.

Cache QoS describes various CAT configurations that demonstrate CAT's wide applicability [15]. Although it applies CAT to a networked application that uses DPDK, Cache QoS examines only the application throughput, but not latency.

Ginseng lets cloud applications bid for LLC space based on their valuation of cache space [13]. Its game-theoretic technique optimizes the benefits perceived by the applications instead of traditional resource utilization metrics such as IPC and LLC hit ratios.

Dirigent maintains low variance across the execution times of latency-sensitive tasks. By monitoring execution time and LLC misses, Dirigent dynamically adjusts a task's cache partitions and

applies conventional dynamic voltage and frequency scaling (DVFS) [38].

Heracles aims to guarantee QoS for latency-sensitive tasks while running best-effort tasks that may consume a diverse set of resources including CPU cycles, cache space, memory bandwidth, network I/O, and power [27]. It uses a CAT-enabled software scheduler that regulates the resource consumption of the best-effort tasks. In particular, this work points out that allocating more cache space to a latency-critical task, and by extension less to best-effort tasks, can be detrimental to the latency-critical task's performance. The best-effort task will incur more cache misses and thus cause more memory bandwidth contention, driving up the tail latency of latency-sensitive tasks. Our work explores this performance penalty in depth (Section 5).

Heracles and Dirigent are both online systems that use heavy-weight instrumentation to dynamically tune a variety of scheduling and resource allocation parameters. Our work shows that using simple offline performance analysis and applying CAT in a static manner offers simple and robust performance isolation. Furthermore, we examine how the mite and contender's performance react to different CAT configurations, which neither Heracles nor Dirigent explores in depth. We are the first amongst these related projects to release our source code publicly.

3 Workload Consolidation with CAT for Low-Latency Networked Mites

We use Intel CAT to achieve performance isolation for consolidated environments. We assign a disjoint set of cache ways to mite and contender tasks, and evaluate the effectiveness of CAT to answer three main questions:

- How much can CAT improve mite tail latency?
- How much does CAT limit contender throughput?
- How should CAT be applied to maximize its benefit?

Unlike prior work using CAT-based performance isolation such as Heracles [27], Dirigent [38], Cache QoS [15], and Ginseng [13], we strive to (1) examine the effectiveness of CAT on *full networked services*, instead of using local simulations or benchmarks, to better reflect datacenter services; (2) target modern low-latency applications achieving low tail latencies of *100–200 microseconds*; and (3) investigate detailed performance implications of workload consolidation by measuring *99.9th percentile latency*. Previous tail latency studies [19, 34, 25, 20] show that the 99.9th percentile latency is a better indicator of large-scale clustered service performance than metrics such as 95th and 99th percentile latencies.

Our experiments use one of three mites: a lock server, an in-memory object store, or an echo server. We implement the first two using MICA, a low-latency networked key-value store [26], varying the number of key-value items based on which system we are modeling. We implement the echo server using gRPC [14].

Our contenders model machine learning, data compression, and distributed computation applications using TensorFlow [33], Lepton [24], and Hadoop [1]. TensorFlow is a heavy-weight task

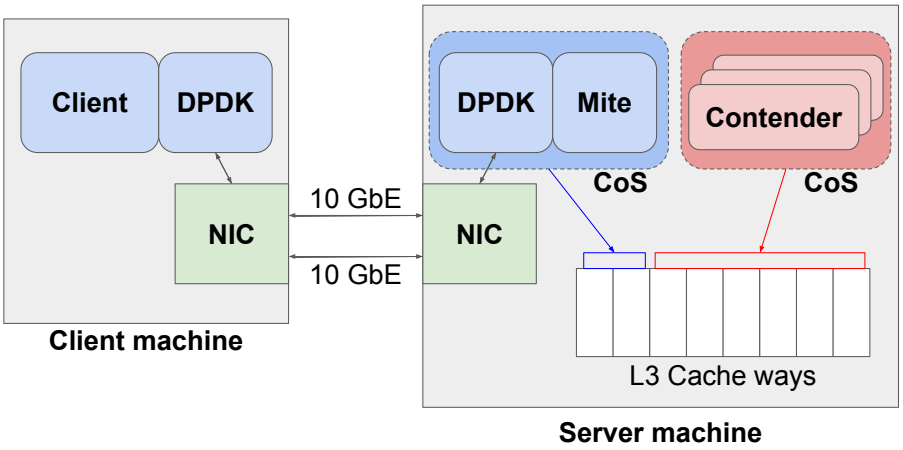


Figure 1: Testbed setup: The server machine runs both MICA (a key-value store) server and contender processes. The client machine sends and receives key-value messages over the network. 10 GbE links directly connect two machines.

that has large working set of tens of GB, while Lepton is a streaming task that processes a few MB of data at a time. TensorFlow uses multithreading for parallel processing on multiple CPU cores. For Lepton, we launch multiple instances of single-threaded Lepton processes that are pinned to different cores. Hadoop runs I/O-intensive big data analytics. For our experiments, we use word count as the workload. Hadoop also uses multithreading for parallel processing on multiple CPU cores, but it does not fully use of all CPU cores as TensorFlow does, so multiple multi-threaded instances are launched similarly to Lepton. All three are throughput-oriented background tasks without strict SLOs in our consolidation scenario; nevertheless, achieving high throughput is still important for improving the overall resource utilization.

The key results of our evaluation are both a demonstration of the overall effectiveness of CAT on representative workloads (Section 4) and a sensitivity analysis to determine empirically how robust, or not, CAT parameters are to mite and contender performance (Section 5).

3.1 Testbed Overview

Figure 1 depicts our experimental infrastructure. Each experiment consists of a *client* machine that issues requests over the network to a *server* machine tasked with responding to them. The client machine runs only a client process, and the server machine runs both a server process (the *mite* process) and numerous *contender* processes that compete for resources.

MICA. Each data point in our results represents a fresh run in which all MICA server instances and all contenders are spawned with the given experiment's parameters. A run consists of an initial 60 s warmup period followed by a ≈60 s period during which the benchmarking infrastructure collects data. This 60 s period corresponds to roughly 300 million requests at the throughputs seen

in most of our trials. The mite server's throughput and latency are measured on the client machine (i.e., they are end-to-end numbers).

gRPC. Each data point for echo server experiments also represents a fresh run in which all server, client, and contender instances are spawned. However, each run does not include any warmup period, and gathering data lasts for ≈330 s. Unlike MICA, the echo server does not need to warm up, as it performs no additional processing when serving requests. The reason why the echo server experiments last longer is because the throughput of the echo server is much lower than MICA, so we must run longer experiments to gather enough samples. Similar to MICA, the mite server's throughput and latency are measured on the client machine.

3.2 Client-Side Measurement Setup

Our client is an open-loop workload generator: it makes new requests at a constant rate without waiting for responses to arrive. For the MICA client, we implement an upper bound of 1024 on the total number of outstanding requests, corresponding to the size of the server and client's receive buffer size on NIC (total 512 packets) because sending more requests would only cause the server and client to drop additional request packets without doing useful work. For the gRPC-based echo server, we rely on the TCP buffers to limit the total number of outstanding requests. This open-loop system better models datacenter services accessed by a large number of clients than do closed-loop systems [31, 37, 22].

For MICA experiments, MICA client itself is responsible for the latency measurements. For each new key-value request, it obtains the current timestamp using the `RDTSC` x86 instruction and stores it in a local descriptor associated with the request. When the client receives a response for the request from the server, it calculates the latency as the elapsed time between the stored timestamp and the current `RDTSC` value.

For echo server experiments, the client is also responsible for the latency measurements. For each RPC, it sends the current timestamp in the same way as the MICA client. The echo server echoes back the same timestamp, and the client calculates the latency as the elapsed time between the echoed timestamp and the current `RDTSC` value.

3.3 Server-Side Measurement Setup

To help understand the end-to-end latencies recorded at the client, we instrument the server-side processes to measure several underlying performance metrics, including execution time, memory bandwidth, and cache misses.

Execution time. When measuring the performance of the mite, we are primarily concerned with its tail response latencies, but for the contenders, we are more interested in overall throughput. As such, we use the UNIX `time` utility to capture the jobs' execution times, then divide a measure of work done by the resulting duration.

		server	client
CPU	Codename	Broadwell	Haswell
	Model	Xeon E5-2683 v4	Xeon E5-2697 v3
	Clock speed	2.1 GHz	2.6 GHz
	Turbo Boost	Disabled	Enabled
	Cores	16	14
Cache	L3 (LLC)	40 MiB	35 MiB
	L3 set-associativity	20-way	20-way
	L2d	256 KiB	256 KiB
	L2i	256 KiB	256 KiB
	L1d	32 KiB	32 KiB
	L1i	32 KiB	32 KiB
Mem	Size	128 GiB	64 GiB
	Frequency	DDR4-2400	DDR4-2133
	Channels	4	4
	Rank	2	2
NIC	Model	Intel X520-T2	Intel X520-T2
	Speed	10 GbE	10 GbE
	Ports	2	2

Table 1: Hardware specifications of our test machines. Note that this only displays the resources available on socket 0/NUMA node 0.

Cache misses. We use common x86 hardware performance counters [17, Section 19.1] to report cache statistics. For example, we calculate the number of misses per kilo-instruction (MPKI) by dividing the number of LLC misses by the number of instructions and multiplying it by 1000.

Memory bandwidth. Our bandwidth measurements are collected using Memory Bandwidth Monitoring, a feature of CMT. Because we run all processes on socket 0 and exclusively use NUMA node 0, we use the "Local External Bandwidth" counter, which includes all traffic between LLC and the main memory on the NUMA node managed by the measuring CPU [17, Section 17.15.5.2]. We allow `perf` to configure the RMID to sample bandwidth from all cache ways.

3.4 Details

Hardware configuration. We list our machine specs in Table 1. Note that, although both machines have two physical CPUs, we only use CPU 0 and NUMA node 0 on each. To permit comparison between the baseline results and those recorded under system contention, the server machine has Turbo Boost disabled via a BIOS option and uses the Linux `performance` CPU governor.

Both machines' L3 caches are inclusive, meaning that no cache line can be present in L1 or

L2 unless it corresponds to a cache line in L3. All involved caches approximate an LRU eviction policy.

Two Cat 6 Ethernet cables link the two machines, patched at each end into a dual-port Intel 10 GbE NIC connected to socket 0 via PCIe 2.0 x8. For the MICA experiments, one of these ports is managed by DPDK [11]'s userland driver which carries MICA messages, while the other port uses the Linux TCP stack to carry SSH control traffic needed by our experiment driver. For the echo server experiments, we used the second port, which uses the Linux TCP stack, for all traffic.

Software. The server runs Linux kernel 4.7.0 patched with the Intel Resource Director Technology patchset to enable process-granularity CAT control [18]. The client, which does not need or use CAT, runs Linux 3.16.0 from Ubuntu 14.04.4. Both systems use DPDK 16.07. With the exception of memory bandwidth figures, all hardware performance counter measurements were obtained using `perf stat` built from the Linux 4.7.0 sources. Bandwidth numbers were collected using a separate `perf stat` process configured to use the `intel_cqm/local_bytes/` counter provided by the CMT/MBM support included in the aforementioned Linux patchset. The contender software versions used are TensorFlow 0.10.0rc0, Lepton 1.2.1, and Hadoop 2.7.3.

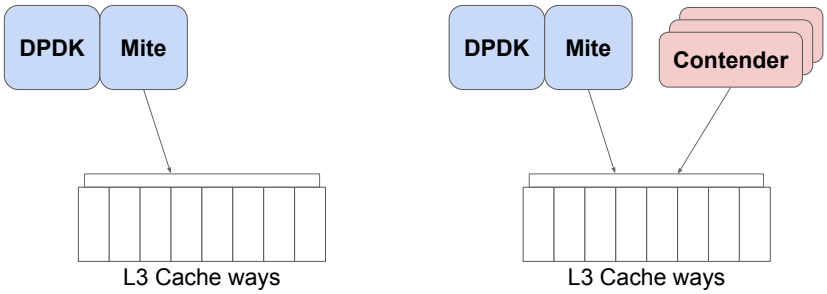
4 Overall Effectiveness of CAT

The primary metric we evaluate is the mite's *99.9th percentile end-to-end tail latency*. Ideal performance isolation would preserve this tail latency regardless of contenders running on the same system. We vary the load on the mite server by changing the request rate of the mite client, and measure the response rate, which we denote as mite throughput. We run each experiment 3 times and plot the median of measured tail latencies, along with that run's throughput, LLC miss rate, and memory bandwidth usage.

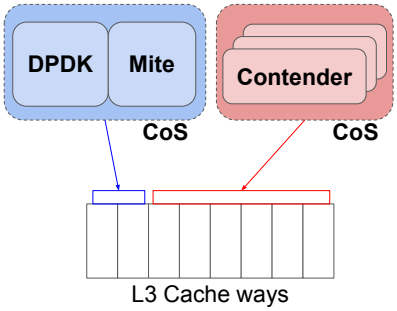
As shown in Figure 2, we examine three different combinations of consolidation and CAT to investigate how contenders and CAT alter the tail latency of the mite:

- *NoContention:* Only the mite runs. We do not apply CAT, so its process may use all of the LLC. This configuration models an unconsolidated node.
- *Contention-NoCAT:* Both the mite and contender tasks run. This configuration does not use CAT, so the mite and the contenders share the cache and can evict each other's cache lines. This configuration examines a consolidation scenario without CAT.
- *Contention-CAT:* The mite and contenders run, and we assign a disjoint set of cache ways to them. This represents a CAT-enabled consolidation scenario.

In the CAT data series, the `rscctrl sysfs` filesystem is used to apply one Capacity Bitmask (CBM) to the mite and a non-intersecting one to the group of contender threads, such that the masks together account for the entire LLC. We assign the mite low ranges of cache ways starting at bit 0 of the CBM, and the contender the rest. For example, if the mite is allocated 2 cache ways, its CBM is `0x3` (cache ways 0 and 1) and the CBM of the contenders is `0xffffc` (cache ways 2 through 19, inclusive).



(a) NoContention. (b) Contention-NoCAT.



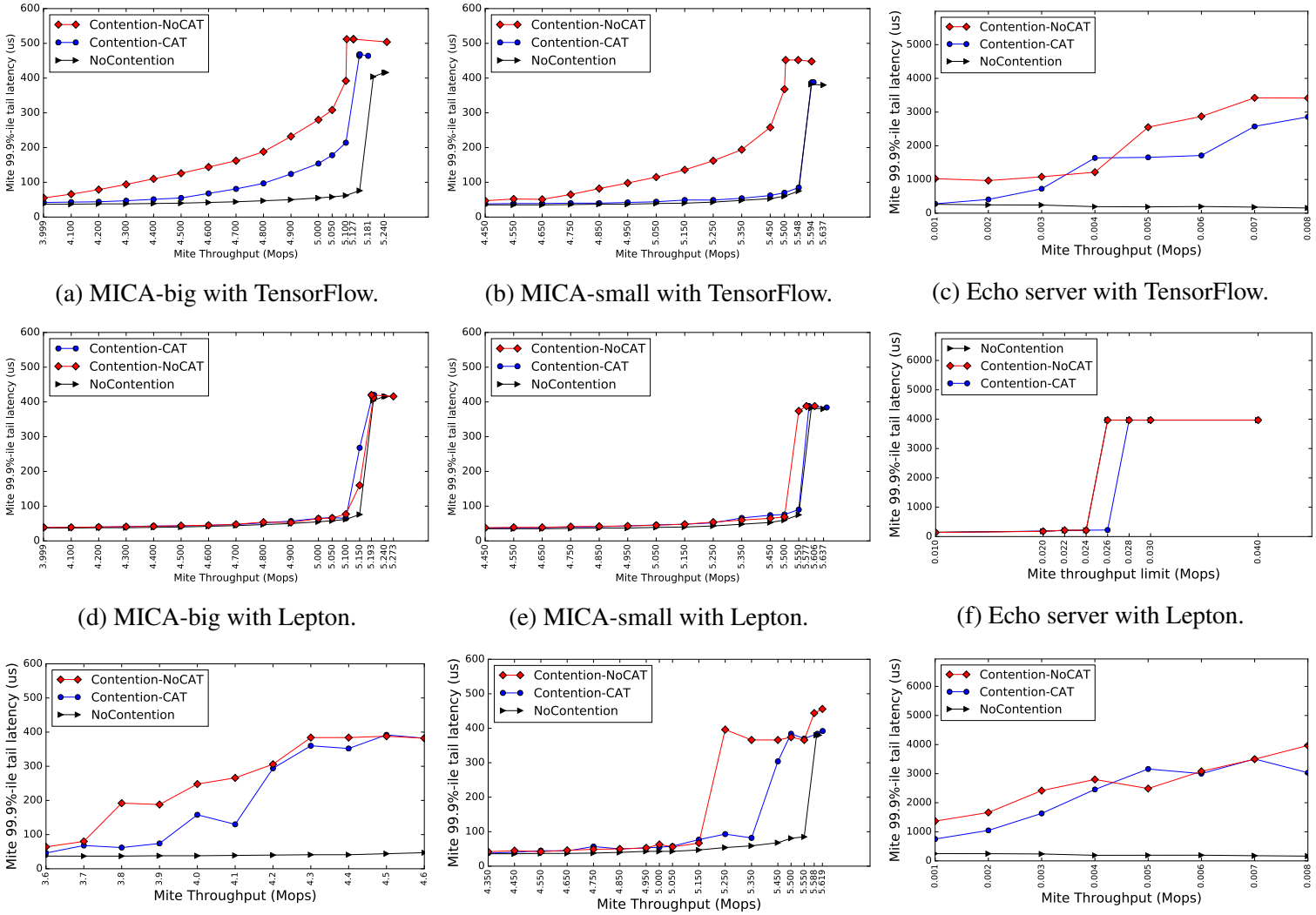
(c) Contention-CAT.

Figure 2: Combinations of consolidation and CAT.

4.1 Workloads

Mites. We consider three mites: a lock server, an object store, and echo server. The lock server and object store are implemented using MICA, and the echo server is implemented using gRPC.

We consider two representative workloads to drive MICA. Both workloads follow YCSB Workload A [8] for their request type ratio and key popularity distribution. The client sends an even split of read and write requests at constant throughput. Request keys follow a Zipf distribution with skew of 0.99. Production key-value store clusters also show a skewed key popularity distribution [2].



(a) MICA-big with TensorFlow.

(b) MICA-small with TensorFlow.

(c) Echo server with TensorFlow.

(d) MICA-big with Lepton.

(e) MICA-small with Lepton.

(f) Echo server with Lepton.

(g) MICA-big with Hadoop.

(h) MICA-small with Hadoop.

(i) Echo server with Hadoop.

Figure 3: Mite 99.9th percentile tail latencies.

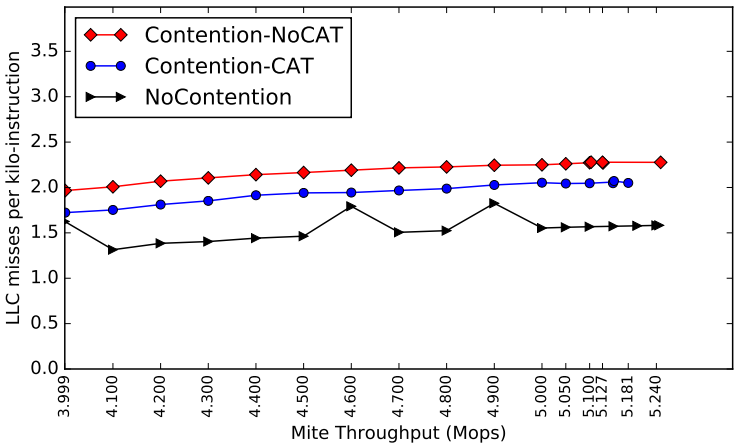
- *MICA-big* models a `memcached` node [29]. It serves a large number of in-memory key-value items over the network. Because its working set is too large to fit in the CPU cache, processing requests typically requires accessing the main memory. We use 44.2M objects, which occupy 2.31 GiB of memory. Consequently, 95% of requests hit 1 GiB of hot data (see Appendix A for the hot set size calculation).
- *MICA-small* models a distributed lock server node [5]. The total number of locks in such a system is small, but they must be replicated throughout a cluster for high throughput and fault tolerance. Without contenders vying for space, the working set of MICA-small fits in LLC. The total number of items for the lock server is 65.0k, which corresponds to 3.47 MiB of memory and 2 MiB of hot data.
- *Echo server* is implemented using gRPC. The mite process receives an RPC containing a timestamp, and echoes this timestamp back to the client. Although there is no additional processing on the server, gRPC involves the kernel network stack and thus incurs large overheads which do not exist in systems like MICA that use user-level networking.

The mite process is pinned to physical core 0 on the server machine using `taskset`, and the contending processes are collectively pinned to the remaining 15 physical cores of that socket.

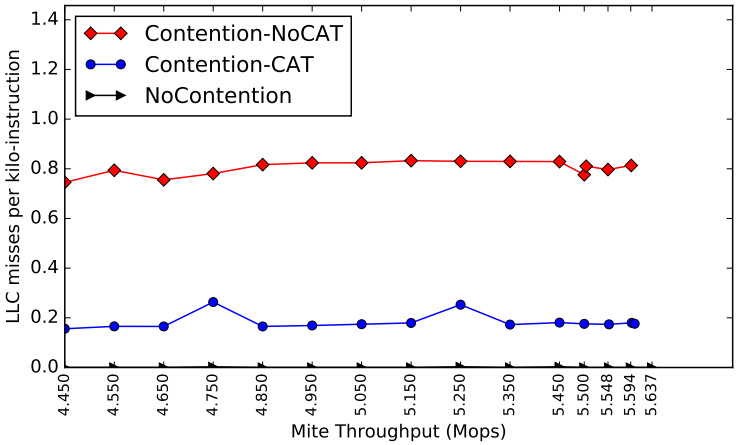
Contenders. To account for the large differences in throughput between MICA and gRPC, we configure the contenders to ensure enough time to collect a sufficiently large number of samples. The two parameters that we change are the number of iterations and the dataset size, depending on the mite. In experiments using TensorFlow as a contender, we train a neural network on the MNIST handwriting dataset [23] and express throughput in epochs per second. For mites MICA-big and MICA-small, we train for one epoch (a single pass through all of the training data); for the gRPC-based echo server we train for five epochs. When using Lepton as the contender, we compress/decompress the test images found in its repository; throughput for Lepton is the average number of images compressed per second. For the MICA-based mites, we run 3 passes over the image set, and for the echo server we run 15. Finally, the Hadoop contender experiments run MapReduce wordcount on a corpus of randomly generated text. The experiments using MICA as the mite run wordcount on 400 MB of text; the experiments using the echo server as the mite run wordcount on 1.8 GB of text.

4.2 Mite Tail Latency and Throughput

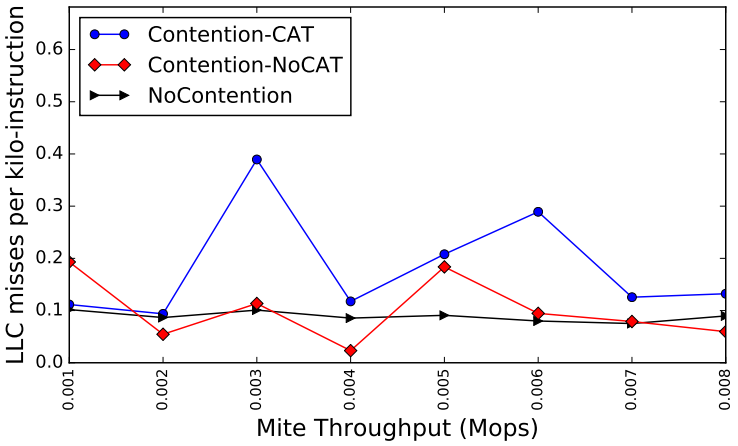
Figure 3 plots the tail latency versus throughput curve for the mite. For Contention-CAT, we allocate a sufficient number of cache ways to hold the mite's working set without excessive over-allocation (as we will discuss in Section 5.1). In the experiments using CAT and MICA, we assign 4 cache ways to the mite and the remaining 16 cache ways to the contender. In the experiments using CAT and the echo server, we assign 7 cache ways to the mite and the remaining 13 cache ways to the contender. Note that the x-axis begins at 3.6–4 Mops (MICA-big), 4.35–4.45 Mops (MICA-small), or 0.001–0.01 Mops (echo server) to focus on throughputs where the tail latencies of the different consolidation approaches start to differ.



(a) MICA-big with TensorFlow.



(b) MICA-small with TensorFlow.



(c) Echo server with TensorFlow.

Figure 4: LLC misses per kilo-instruction (MPKI).

MICA-big & TensorFlow. Figure 3a compares MICA-big mite performance with and without TensorFlow-based contenders. We make two observations:

First, contention-NoCAT suffers high tail latency because of the cache contention between MICA-big and TensorFlow. For example, MICA's tail latency is 274 µs at 5 Mops, which is 5× higher than without contenders at the same load.

Second, contention-CAT successfully reduces the performance interference; however, it does not eliminate the cache contention completely for two reasons: (a) MICA-big's large working set requires some requests to access memory ("large mite working set"); and (b) TensorFlow consumes large memory bandwidth ("expensive memory access due to contenders"). We investigate these factors in detail in Section 5.2.

MICA-small & TensorFlow. Figure 3b uses MICA-small as the mite, and runs TensorFlow as a contender. Since MICA-small has a small working set (i.e., the above "large mite working set" concern is absent), Contention-CAT provides low tail latency even though TensorFlow is making memory accesses more expensive by consuming memory bandwidth. However, because the contender is a heavy-weight task and can evict the mite's data from the cache, Contention-NoCAT suffers high tail latency.

Echo server & TensorFlow. Figure 3c uses the echo server as the mite and TensorFlow as the contender. gRPC uses the kernel TCP stack as opposed to DPDK. Thus, the echo server's tail latencies are an order of magnitude higher than MICA's, such that the performance loss from cache misses is much less significant than it is with MICA. Nevertheless, CAT shows some limited benefit to the echo server's tail latency when using TensorFlow as a contender.

Lepton. Figures 3d–3f use Lepton as a contender. Lepton is more *arithmetic*-intensive than memory- or cache-intensive; its working set fits in a few megabytes. As expected, Lepton contenders have less impact on the mite than TensorFlow does. The tail latency of the contended cases, with and without CAT, is very close to that of the uncontended case. (The core question for which we chose Lepton is to understand whether CAT will reduce the contender's throughput, discussed later.)

MICA & Hadoop. Figures 3g and 3h use Hadoop as a contender with MICA-big and MICA-small as the mite. Hadoop is disk I/O-intensive, but it is less CPU- and memory-intensive than TensorFlow. The results show that CAT is less effective in shielding the mites from contender-based interference. Hadoop's I/O path consumes additional non-memory resources, such as kernel CPU time to process the disk I/O. CAT is unable to protect MICA-big against this type of resource contention at high throughputs. For MICA-small, CAT provides some benefit because the working set fits in cache and thus the potential for Hadoop to affect tail latency is reduced.

Echo server & Hadoop. Figure 3i uses the echo server as the mite and Hadoop as the contender. The echo server's relatively heavy-weight gRPC packet processing path competes with the also heavy-weight Hadoop for resources, which affects the mite's tail latency. Here too, given the

absolute latency numbers (ms) and the nature of the contention, CAT is ill-suited and unable to provide the necessary performance isolation.

4.3 Mite Cache Misses

We briefly describe the effect of CAT on the mite's cache misses for the consolidation scenarios. We perform a more detailed analysis on the relationship between the number of cache ways assigned to the mite and its cache misses in Section 5.1.

Figure 4a shows the LLC misses per kilo-instruction (MPKI) of MICA-big when consolidated with TensorFlow. LLC misses are higher in Contention-NoCAT than in Contention-CAT, causing more memory accesses. Their relative MPKIs differ by 9.07% when the throughput is 5 Mops. This difference in MPKIs is large enough to cause large tail latency differences because the 99.9th percentile tail latency is determined by the latency of 1 request out of 1000. Additionally, slow requests can cause temporary periods of lower throughput, causing requests to build up in the queue, which in turn increases the tail latency.

Figure 4b depicts the MPKI of MICA-small. With CAT, the miss ratio is close to 0, approaching the non-consolidation scenario. The low miss ratio shows that we achieve almost perfect isolation for the consolidation of MICA-small and TensorFlow.

Figure 4c plots the MPKI of the echo server. The echo server's overall miss rate is weakly correlated with its tail latency shown in Figure 3c. CAT reduces the 99.9th percentile tail latency of the echo server, but it often increases the overall miss rate of the echo server. This result suggests that the cache miss rate of the mite alone is insufficient to estimate how effectively CAT improves the tail latency of the mite. That is, reducing the mite's cache misses is not necessarily beneficial for the mite performance; this observation is consistent with the result of cache overallocation as we discuss in Section 5.2.

We omit the cache miss results for the mite with Lepton and Hadoop because they show similar results.

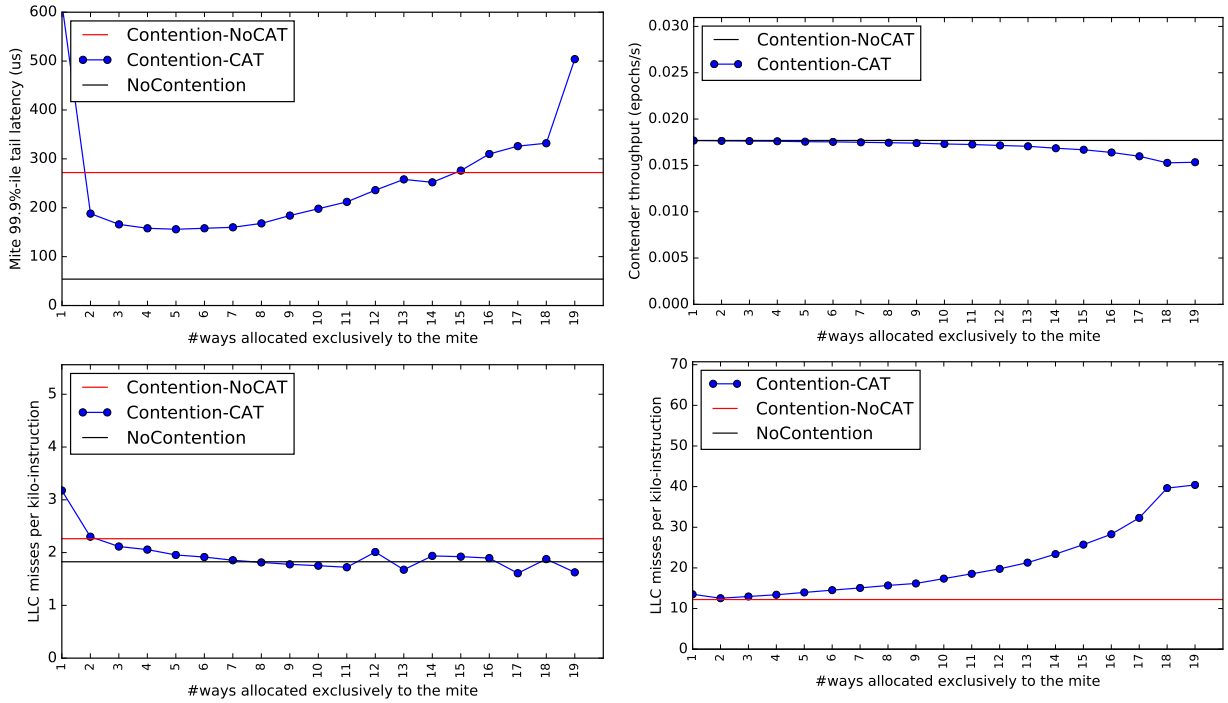
4.4 Contender Throughput

Regardless of mite throughput, using CAT has little effect on contender throughput when properly configured (4 ways for MICA and 7 ways for the echo server). The contender throughput difference between CAT and NoCAT for Hadoop when running alongside MICA-big was at most about 2.7%. In all other cases, the contender throughput difference between CAT and NoCAT was less than 0.5%.

Section 5 discusses counterexamples where misusing CAT can both reduce contender throughput and harm the mite's tail latency.

4.5 Summary

CAT provides performance isolation for various consolidation scenarios. Its effectiveness depends on (1) the intensity of memory access by contenders and (2) the working set of the mite. If the contender creates a low load on memory bandwidth (e.g., Lepton), both consolidation with and



(a) Tail latency and LLC misses of MICA-big.

(b) Throughput and LLC misses of TensorFlow.

Figure 5: Performance of MICA-big and TensorFlow under consolidation when varying cache ways.

without CAT rarely increase the mite's tail latency. However, if the contender accesses memory intensively (e.g., TensorFlow, Hadoop), leveraging CAT substantially improves the mite's tail latency. CAT is most effective when the mite has small working set that fits in the cache, providing nearly perfect performance isolation. If MICA's working set is larger than the cache, consolidation with CAT provides only partial performance isolation for the mite, but it still allows much lower tail latency than without CAT. With CAT, the contender sustains high throughput in these experiments where the mite uses 4 or 7 cache ways.

5 In-Depth Performance Analysis

This section analyzes how CAT affects the performance of the mite and the contender using various CAT configurations. Because of space constraints, we focus on the TensorFlow contender, which causes more performance interference on the mite's tail latency than Lepton (see Section 4.2), and exhibits memory bandwidth behavior that makes it prone to CAT misconfiguration which is not present in experiments using Hadoop as the mite (Section 5.2). Unless specified, we set the request throughput of the mite in this section to 5 Mops, which provides high throughput at low latency (i.e., it is before the knee of the latency-throughput curve). We chose 5 Mops to avoid exaggerating the benefit of CAT while still exploring workload space that shows the potential benefits of CAT.

5.1 How Many Cache Ways?

Figure 5 plots the performance (tail latency or throughput) and LLC misses of MICA-big and TensorFlow versus the number of cache ways allocated to the mite (MICA-big) process.

MICA-big's latency is at its lowest when it has 3–8 cache ways (Figure 5a, top). In this range, the reduction in TensorFlow throughput (Figure 5b, top) is less than 1% despite a 42.6% improvement in tail latency for MICA-big over the no contention baseline.

The two ends of the graph, however, have worse performance than not using any cache-based performance isolation at all. Unsurprisingly, with only one cache way, MICA-big's tail latency is at its worst because a single cache way is unable to hold the process's working set.

At the other end of the graph, when MICA-big has most of the cache and TensorFlow is limited to just one way, TensorFlow's throughput drops by 13.3%. This drop in contender throughput, however, *does not* improve MICA-big's tail latency, which is still over 500 µs, indicating that this point in the tradeoff space is suboptimal for both MICA-big and TensorFlow. We explore this issue in the next subsection.

We chose 4 cache ways for CAT in the experiments using MICA in Section 4 based on the above observation. Using 8 cache ways provides negligible mite tail latency reduction compared to 3 cache ways, and preferring cache way allocations at the lower end of this range minimizes the impact on the contender. Since static cache allocations based on offline measurement may run the risk of misrepresenting the online workload, we add a slack at the bottom of this range, leading us to use 4 cache ways.

Through the same process, we came to the conclusion that 7 cache ways was sufficient for the echo server. We performed a similar experiment using the echo server as a mite. In this case, the tail latency had significantly more variance, and there were two key differences from the MICA case: (1) The echo server did not experience the increase in tail latency when contender was starved for cache space. We believe that this is because the echo server makes fewer memory accesses to serve requests than MICA does and thus is less susceptible to the increased memory bandwidth use by the contender (see Section 5.2). (2) The echo server benefited from having more cache ways, up to 6 or 7, compared to MICA. As a result, we used 7 cache ways in the experiments using the echo server in Section 4.

5.2 More Cache Isn't Necessarily Beneficial

Intuitively, giving additional cache ways to the mite should progressively reduce its tail latency until it becomes equal to the NoContention latency. Figure 5a shows, however, that tail latency gets worse when the mite has exclusive access to almost all of the cache. This degradation is not the result of imperfect LLC isolation: MICA-big's LLC miss rate decreases monotonically as it is allocated more cache ways (Figure 5a, bottom).

We observe that memory bandwidth contention is a major contributor to the reduction in contender throughput. Figure 5b shows that as TensorFlow is restricted to fewer cache ways, it experiences more frequent LLC misses. These misses are caused by a combination of lower total cache capacity and lower cache associativity. The additional misses translate directly into new memory accesses to retrieve the data previously fulfilled by cache.

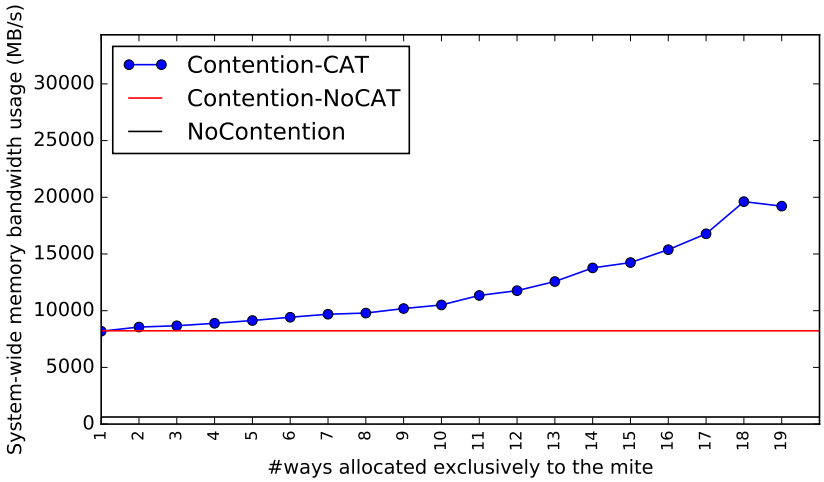


Figure 6: System-wide memory bandwidth use when running MICA-big and TensorFlow, varying cache allocation.

Figure 6 measures this effect more directly by plotting system-wide memory bandwidth consumed during these experiments. Recall that MICA-big's working set size is larger than LLC. Thus, although MICA-big experiences fewer cache misses under generous allocations, some accesses will still go to memory. When that happens, those access must compete with TensorFlow for memory bandwidth on the shared bus and are thus *more expensive*, resulting in a higher tail latency for MICA-big.

Note that memory bandwidth contention is most significant when the mite has a working set too large for LLC and the contender is memory access intensive. In our experiment, we do not observe this tail latency degradation when using MICA-small or the Lepton contender provided the mite has a sufficiently large cache allocation.

5.3 Limiting Contender Core Count

As an alternative technique for performance isolation, contenders may use a small number of cores to limit their performance interference with the mite.

Figure 7 depicts the tail latency of MICA-big when varying the number of cores dedicated to TensorFlow. The results show that reducing the contender core count without CAT (Contention-NoCAT) can indeed provide performance isolation. However, when using more than 50% of the cores, CAT-enabled consolidation (Contention-CAT), on average, allows the contender to use two to three additional cores and still meet the same mite latency SLO as Contention-CAT. Depending on the SLO, this improves the per-node contender throughput by 14% to 20%.

5.4 Varying Working Set

Unlike previous sections where we focus on two representative dataset sizes that model the lock server and object store, here we vary the working set of the key-value store to understand its effect

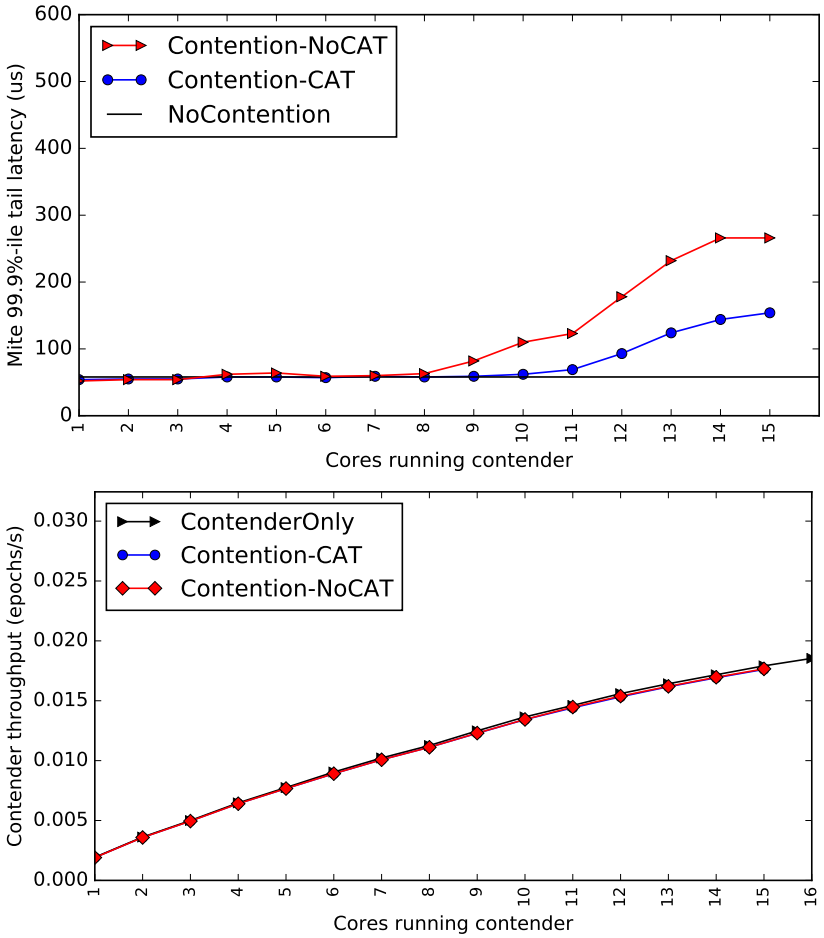


Figure 7: Tail latency of MICA-big and machine learning throughput when varying the number of cores used by TensorFlow.

on performance isolation. Because it takes a long time for the remote client to populate the key-value store using a skewed key distribution, we locally prepopulate the server with a full key-value dataset before the usual warmup period for these experiments.

Figure 8 shows that using CAT can also provide performance isolation benefits over naive consolidation when mites have large working set sizes. Even with only 4 cache ways (8 MiB), we get 45% lower tail latency compared to Contention-NoCAT for 10 GiB working set.

6 Conclusion

Workload consolidation in datacenters enables high resource utilization and cost-effectiveness. However, this high resource utilization is traditionally accompanied by performance interference between throughput-oriented tasks ("contenders") and low-latency networked applications ("mites"). This problem is exacerbated by recent advancements in underlying networking technology, result-

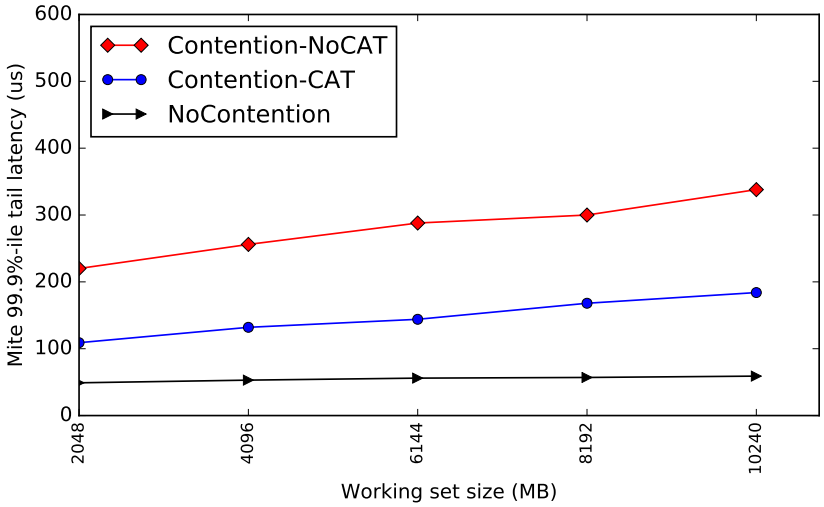


Figure 8: Tail latency of the key-value store with TensorFlow for varied working set.

ing in tight tail latency SLOs of tens to hundreds of microseconds. This creates a situation where latency-critical datacenter applications are increasingly sensitive and intolerant to local interference, making effective performance isolation critical to making workload consolidation possible. We evaluate the effectiveness of statically configured hardware-assisted cache partitioning provided by Intel Cache Allocation Technology (CAT) in addressing this performance isolation problem. We show that these static configurations can be effective even across mite and contender workloads with different properties, reducing the necessity of complex online controllers or schedulers. The code we used in our evaluation is available at `https://github.com/efficient/catbench`, and the data files which can be used to replicate our results are available at `https://github.com/efficient/catbench/releases`.

Acknowledgments

We would like to thank Brandon Bohrer for his early contributions to the project.

References

[1] Apache Hadoop. `http://hadoop.apache.org/`, 2017.

[2] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the SIGMETRICS'12*, June 2012.

[3] Luiz André Barroso and Urs Hölzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines*. Morgan & Claypool, 2009.

[4] Edouard Bugnion, Jennifer M. Anderson, Todd C. Mowry, Mendel Rosenblum, and Monica S. Lam. Compiler-directed page coloring for multiprocessors. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 1996.

[5] Michael Burrows. The Chubby lock service for loosely-coupled distributed systems. In *Proc. 7th USENIX OSDI*, Seattle, WA, November 2006.

[6] Derek Chiou, Prabhat Jain, Srinivas Devadas, and Larry Rudolph. Dynamic cache partitioning via columnization. In *Proceedings of Design Automation Conference (DAC)*, 2000.

[7] Henry Cook, Miquel Moreto, Sarah Bird, Khanh Dao, David A. Patterson, and Krste Asanovic. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[8] Brian Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, Indianapolis, IN, June 2010.

[9] Zehan Cui, Licheng Chen, Yungang Bao, and Mingyu Chen. A swap-based cache set index scheme to leverage both superpage and page coloring optimizations. In *Proceedings of the 51st Annual Design Automation Conference (DAC)*, 2014.

[10] Jeffrey Dean and Luiz André Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, February 2013.

[11] Data Plane Development Kit (DPDK). `http://dpdk.org/`, 2017.

[12] Alexandra Fedorova, Margo Seltzer, and Michael D. Smith. Improving performance isolation on chip multiprocessors via an operating system scheduler. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, 2007.

[13] Liran Funaro, Orna Agmon Ben-Yehuda, and Assaf Schuster. Ginseng: Market-driven LLC allocation. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, June 2016.

[14] gRPC, a high performance, open-source universal RPC framework. `http://www.grpc.io/`, 2017.

[15] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. Cache QoS: From concept to reality in the Intel Xeon processor E5-2600 v3 product family. In *Proc. HPCA*, 2016.

[16] Cache allocation technology improves real-time performance. `https://www-ssl.intel.com/content/www/us/en/communications/cache-allocation-technology-white-paper.html`, 2015.

[17] Intel 64 and IA-32 architectures developer's manual: Vol. 3A. http://www.intel.com/content/www/us/en/architecture-and-technology/, 2011.

[18] Intel resource director technology patch. https://patchwork.kernel.org/patch/9226287/, 2016.

[19] Virajith Jalaparti, Peter Bodik, Srikanth Kandula, Ishai Menache, Mikhail Rybalkin, and Chenyu Yan. Speeding up distributed request-response workflows. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[20] Myeongjae Jeon, Yuxiong He, Hwanju Kim, Sameh Elnikety, Scott Rixner, and Alan L. Cox. TPC: Target-driven parallelism combining prediction and correction to reduce tail latency in interactive services. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[21] Harshad Kasture and Daniel Sanchez. Ubik: Efficient cache sharing with strict QoS for latency-critical workloads. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2014.

[22] Harshad Kasture and Daniel Sanchez. TailBench: A benchmark suite and evaluation methodology for latency-critical applications. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, September 2016.

[23] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, November 1998.

[24] Lepton. https://github.com/dropbox/lepton, 2017.

[25] Jialin Li, Naveen Kr. Sharma, Dan R. K. Ports, and Steven D. Gribble. Tales of the tail: Hardware, OS, and application-level sources of tail latency. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)*, 2014.

[26] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A holistic approach to fast in-memory key-value storage. In *Proc. 11th USENIX NSDI*, Seattle, WA, April 2014.

[27] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Improving resource efficiency at scale with Heracles. *ACM Transactions on Computer Systems (TOCS)*, 34(2), May 2016.

[28] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-Up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[29] Memcached: A distributed memory object caching system. http://memcached.org/, 2011.

[30] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C. Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, David Stafford, Tony Tung, and Venkateshwaran Venkataramani. Scaling Memcache at Facebook. In *Proc. 10th USENIX NSDI*, Lombard, IL, April 2013.

[31] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. Open versus closed: A cautionary tale. In *Proc. 3rd Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, CA, May 2006.

[32] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *Journal of Supercomputing*, 28(1), April 2004.

[33] TensorFlow. `https://www.tensorflow.org/`, 2017.

[34] Yunjing Xu, Zachary Musgrave, Brian Noble, and Michael Bailey. Bobtail: Avoiding long tails in the cloud. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (NSDI)*, 2013.

[35] Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. Elfen scheduling: Fine-grain principled borrowing from latency-critical workloads using simultaneous multithreading. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, June 2016.

[36] Xiao Zhang, Sandhya Dwarkadas, and Kai Shen. Towards practical page coloring-based multicore cache management. In *Proceedings of the 4th ACM European Conference on Computer Systems (EuroSys)*, 2009.

[37] Yunqi Zhang, David Meisner, Jason Mars, and Lingjia Tang. Treadmill: Attributing the source of tail latency through precise load testing and statistical inference. In *ISCA*, 2016.

[38] Haishan Zhu and Mattan Erez. Dirigent: Enforcing QoS for latency-critical tasks on shared multicore systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

A Hot Set Size

For a Zipf distribution with skew of 0.99, its CDF is a ratio of generalized harmonic numbers:

$$F(k) = \frac{\sum_{i=1}^{k} \frac{1}{i^{0.99}}}{\sum_{i=1}^{N} \frac{1}{i^{0.99}}} \tag{1}$$

This equation converts between the total working set size and the hot set size by fixing the ratio of hot-set requests to cold-set requests. $F(k)$ is ratio of requests that land at keys at or above rank k. N is the total number of keys in the system, which is effectively the total working set size of the mite (ignoring constant overhead). k is the hot set size.