View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by DSpace@MIT

Ubik: Efficient Cache Sharing with Strict QoS for Latency-Critical Workloads

Harshad Kasture   Daniel Sanchez

Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology
{harshad, sanchez}@csail.mit.edu

Abstract

Chip-multiprocessors (CMPs) must often execute workload mixes with different performance requirements. On one hand, user-facing, latency-critical applications (e.g., web search) need low tail (i.e., worst-case) latencies, often in the millisecond range, and have inherently low utilization. On the other hand, compute-intensive batch applications (e.g., MapReduce) only need high long-term average performance. In current CMPs, latency-critical and batch applications cannot run concurrently due to interference on shared resources. Unfortunately, prior work on quality of service (QoS) in CMPs has focused on guaranteeing average performance, not tail latency.

In this work, we analyze several latency-critical workloads, and show that guaranteeing average performance is insufficient to maintain low tail latency, because microarchitectural resources with state, such as caches or cores, exert *inertia* on instantaneous workload performance. Last-level caches impart the highest inertia, as workloads take tens of milliseconds to warm them up. When left unmanaged, or when managed with conventional QoS frameworks, shared last-level caches degrade tail latency significantly. Instead, we propose Ubik, a dynamic partitioning technique that predicts and exploits the transient behavior of latency-critical workloads to maintain their tail latency while maximizing the cache space available to batch applications. Using extensive simulations, we show that, while conventional QoS frameworks degrade tail latency by up to 2.3×, Ubik simultaneously maintains the tail latency of latency-critical workloads and significantly improves the performance of batch applications.

Categories and Subject Descriptors B.3.2 [*Memory structures*]: Design styles—Cache memories; C.1.4 [*Processor architectures*]: Parallel architectures

Keywords multicore, interference, isolation, quality of service, tail latency, resource management, cache partitioning

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASPLOS '14, March 1–4, 2014, Salt Lake City, Utah, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2305-5/14/03...$15.00.
http://dx.doi.org/10.1145/2541940.2541944

1. Introduction

General-purpose systems are designed to maximize *long-term average performance*. They focus on optimizing the common case and rely on best-effort techniques. While this is a good match to batch applications, many applications are latency-critical, and *short-term worst-case response times* determine their performance. This disconnect between how systems are designed and used causes large inefficiencies.

While latency-critical workloads are common in embedded, mobile, and datacenter computing, in this work we focus on datacenters. Datacenter servers often execute user-facing applications that need low tail latencies. For example, web search nodes need to provide 99th percentile latencies of a few milliseconds at worst [11, 43]. Moreover, to avoid being dominated by queuing delays, latency-critical workloads are typically executed at low loads. These stringent requirements limit the utilization of servers dedicated to a single application. For example, Meisner et al. report average server utilization of 14% for applications with medium QoS needs, and 7% for real-time services [35]. Barroso and Hölzle report an average utilization below 30% on the Google fleet [3], which includes both real-time services (e.g., search), and batch applications (e.g., MapReduce). This poor utilization wastes billions of dollars in equipment and terawatt-hours of energy yearly [3].

Fortunately, datacenters also support a large amount of batch applications that only require high average performance. Ideally, systems should be designed to execute mixes of latency-critical and batch applications, providing strict performance guarantees to latency-critical applications and achieving high utilization at the same time. However, this is not possible in current chip-multiprocessors (CMPs), because concurrent applications contend on shared resources, such as cache capacity, on-chip network bandwidth, and memory bandwidth, causing unpredictable interference.

To sidestep this issue, prior work has proposed several techniques to provide quality of service (QoS) in CMPs and datacenters. On one hand, several software-only solutions [12, 33, 54, 60] rely on empirically detecting interference, and throttling, migrating, or avoiding to run applications that induce too much degradation. These schemes often avoid interference, but do not prevent it. On the other hand, prior work has introduced hardware to explicitly par-

tition shared resources [9, 16, 21, 25, 38, 45], and frameworks that manage these resources dynamically to provide QoS [5, 13, 17, 29, 31, 37, 42, 49, 61]. However, all these techniques only provide guarantees on long-term average performance (e.g., IPC or queries per second over billions of instructions), instead of on worst-case, short-term behavior.

The fundamental issue with conventional QoS frameworks is that, despite having hardware that enforces explicit partitioning, they reconfigure at coarse-grained intervals and assume instant transitions between steady states, *ignoring transient behavior*. For example, cache partitioning policies [39, 42, 52] periodically reallocate space across applications. When an application gets additional space, it can take tens of milliseconds to fill it with its own cache lines and reap the benefits of a larger allocation. We call this effect *performance inertia*. Conventional QoS schemes avoid the effects of inertia by reconfiguring at long enough intervals (making transients rare), and adapt across intervals (e.g., giving more resources to an application that is below its target performance [60]). This long-term adaptation works for batch workloads, but not for latency-critical workloads, as long low-performance periods dominate tail latency. Alternatively, we can implement fixed, conservative resource allocations. For example, if a latency-critical application needs 2 MB of cache to meet its target tail latency, we can statically assign it 2 MB of cache. This eliminates transients, but it is too conservative. Instead, we advocate a more efficient approach: by understanding and leveraging transient behavior, we can still perform dynamic, aggressive resource sharing without sacrificing short-term performance.

In this paper, we develop adaptive techniques that provide *strict QoS guarantees on tail latency* for latency-critical applications. We focus on last-level caches as they have the largest amount of microarchitectural, hardware-managed state and therefore induce significant inertia. Although we expect our findings to apply to other resources with inertia, we leave such evaluation to future work. Specifically, our contributions are:

- We develop a suite of representative latency-critical workloads, and define a methodology to accurately measure the interaction between tail latency and microarchitecture using simulation. We then show that most workloads exhibit significant inertia, which impacts tail latency considerably.
- We extend conventional partitioning policies that achieve a single objective (e.g., maximizing cache utilization) to achieve the dual objectives of maximizing cache utilization while providing strict QoS guarantees for latency-critical workloads (Section 4). We show that, without accounting for inertia, these solutions either preserve tail latency but are too inefficient, or degrade tail latency significantly.
- We propose *Ubik*, a cache partitioning policy that characterizes and leverages inertia to share space dynamically while maintaining tail latency (Section 5). Ubik safely takes space away from latency-critical applications when they are idle, and, if needed, temporarily boosts their space when they are active to maintain their target tail latency. In designing Ubik, we show that transient behavior can be accurately predicted and bounded. Ubik is mostly implemented in software and needs simple hardware extensions beyond partitioning.
- We evaluate Ubik against conventional and proposed management schemes (Section 7). While conventional adaptive partitioning policies [42] degrade tail latency by up to 2.3×, Ubik strictly maintains tail latency and simultaneously achieves high cache space utilization, improving batch application performance significantly. Moreover, by enabling latency-critical and batch applications to coexist efficiently, Ubik considerably improves server utilization.

2. Background and Related Work

Lack of quality of service fundamentally limits datacenter utilization, due to the combination of three factors. First, low-latency datacenter-wide networks [22] and faster storage (e.g., Flash) have turned many I/O-bound workloads into compute-bound, often making compute the major source of latency [40, 43]. Second, single-node latencies must be small and tightly distributed, since servicing each user request involves hundreds to thousands of nodes, and the slowest nodes often determine end-to-end latency. As a result, responsive online services (100-1000 ms) often require single-node tail latencies in the millisecond range [11]. Third, frequency scaling, which drove compute latency down for free, has stopped. Instead, we now rely on CMPs with more cores and shared resources, which suffer from interference and degrade performance. Therefore, in this work we focus on QoS on CMPs.

2.1. Quality of Service in CMPs

Prior work has proposed two main approaches to provide QoS in CMPs: software-only schemes that detect and disallow colocations that cause excessive interference, and software-hardware schemes where software explicitly manages shared hardware resources. Both approaches only guarantee average long-term performance, instead of worst-case, short-term performance, and ignore transients and performance inertia.

Software-only QoS management: Prior work has proposed profiling, coscheduling, and admission control techniques to perform QoS-aware scheduling in datacenters [12, 33, 60] (e.g., coscheduling latency-sensitive applications only with workloads that are not memory-intensive). When used across the datacenter, these techniques are complementary with hardware QoS schemes, since they can select workloads that use resources in a complementary way, improving system utilization. However, while usable in current best-effort hardware, these schemes do not disallow short-term interference, and must be conservative, only colocating workloads that have low memory utilization, which leaves shared memory resources underutilized. With QoS-enforcing hardware, these techniques could coschedule workloads much more aggressively, relying on hardware to prevent interference.

Hardware-based QoS enforcement: Prior work has proposed hardware schemes to partition shared resources [9, 16, 21, 25, 38, 45], as well as software QoS frameworks that

leverage these mechanisms to implement fairness, maximize throughput, or provide differentiated service [5, 13, 17, 20, 29, 31, 37, 42, 49, 61]. However, these techniques have been developed with the goal of meeting *long-term performance goals* (e.g., IPC or queries per second over billions of instructions), instead of providing strict guarantees on worst-case, short-term behavior (e.g., bounding the performance degradation of a 1 ms request). These systems reconfigure at coarse-grained intervals and assume instant transitions between steady states. However, for many resources, reconfigurations cause relatively long transients, inducing *performance inertia*. While inertia can be safely ignored for batch workloads by reconfiguring at long enough intervals, it *cannot be ignored for latency-critical workloads*.

Different microarchitectural resources induce different degrees of performance inertia. On one hand, bandwidth has minimal inertia: reallocating it can be done in tens of cycles [21], and applications instantly start benefiting from it. On the other hand, components with significant state (such as cores or caches) take time to warm up, causing significant transients. While core state is relatively small, the memory wall keeps driving the amount of cache per core up [7, 58], making cache-induced inertia a growing concern. Therefore, we focus on shared cache management.

2.2. Cache Partitioning in CMPs

We first discuss hardware schemes to enforce partition sizes, then review relevant policies to manage them.

Partitioning schemes: A partitioning scheme should support a large number of partitions with fine-grained sizes, disallow interference among partitions, avoid hurting cache associativity or replacement policy performance, support changing partition sizes efficiently, and require small overheads. For our purposes (Section 4), fine-grained partition sizes, strict isolation, and fast resizing are especially important. Achieving all these properties simultaneously is not trivial. For example, way-partitioning [9], the most common technique, restricts insertions from each partition to its assigned subset of ways. It is simple, but it supports a small number of coarsely-sized partitions (in multiples of way size); partition associativity is proportional to its way count, so partitioning degrades performance; and more importantly, reconfigurations are slow and unpredictable (Section 7). Alternatively, Vantage [45] leverages the statistical properties of skew-associative caches [48] and zcaches [44] to implement partitioning efficiently. Vantage supports fine-grained partitions (defined in cache lines), provides strict guarantees on partition sizes and isolation, can resize partitions without moves or invalidations, reconfiguration transients are much faster than in way-partitioning [45], and it is cheap to implement (requiring $\approx$1% extra state and negligible logic). For these reasons, we use Vantage to enforce partition sizes, although the policies we develop are not strictly tied to Vantage.

Partitioning policies: Partitioning policies consist of a monitoring mechanism, typically in hardware, that profiles partitions, and a controller, in software or hardware, that uses this information to set partition sizes.

Utility-based cache partitioning (UCP) is a frequently used policy [42]. UCP introduces a *utility monitor* (UMON) per core, which samples the address stream and measures the partition's *miss curve*, i.e., the number of misses that the partition would have incurred with each possible number of allocated ways. System software periodically reads these miss curves and repartitions the cache to maximize cache utility (i.e., the expected number of cache hits). Although UCP was designed to work with way-partitioning, it can be used with other schemes [45, 59]. Ubik uses UMONs to capture miss curves, and extends them to enable quick adaptation for latency-critical workloads.

Partitioning policies can have goals other than maximizing utility. In particular, CoQoS [29] and METE [49] combine cache and bandwidth partitioning mechanisms to provide end-to-end performance guarantees. As discussed before, these guarantees are only on long-term performance. For example, METE uses robust control theory to efficiently meet target IPCs for each application, but adding new applications and dynamic application behavior cause variability on short-term IPCs [49, Figure 10]. Prior work has proposed policies that optimize for responsiveness. PACORA [5] uses convex optimization to find resource partitionings that minimize penalty functions. These functions can encode both average performance and latency requirements. Cook et al. [10] propose a software-only gradient descent algorithm that limits the average degradation that one or more background applications exert on a foreground application. While both techniques can be used to reduce latency and responsiveness, they still use periodic reconfiguration and ignore transients, so they only work when the desired latency is much larger than the transient length, which makes them inapplicable to latency-critical applications (Section 3).

2.3. Cache Management in Real-Time Systems

Latency-critical datacenter workloads are one instance of an application with soft real-time requirements running on commodity systems. Prior work on soft real-time systems has used static cache partitioning to provide strict QoS [8, 28]. In contrast, Ubik manages partition sizes dynamically and at fine granularity to provide QoS to latency-critical workloads, and, at the same time, maximize the space available to batch applications. Ubik should be directly applicable to other domains with mixes of soft real-time and batch applications (e.g., mobile and embedded computing).

Caches are problematic for hard-real time applications, which must not violate deadlines [47]. As a result, hard real-time systems often favor scratchpads or locked caches [41, 57]. We do not target hard-real time workloads in this work.

3. Understanding Latency-Critical Applications

Latency-critical applications are significantly different from the batch workloads typically used in architectural studies. In

Workload	Configuration	Requests
xapian	English Wikipedia, zipfian query popularity	6000
masstree	mycsb-a (50% GETs, 50% PUTs), 1.1GB table	9000
moses	opensubtitles.org corpora, phrase-based mode	900
shore	TPC-C, 10 warehouses	7500
specjbb	1 warehouse	37500

Table 1: Parameters of the latency-critical workloads studied.

this section, we analyze several representative latency-critical workloads, with a focus on understanding these differences.

3.1. Applications

We have selected five latency-critical workloads, intended to capture a broad variety of domains and behaviors:

- `xapian`, a web search engine, typically a latency-critical, compute-intensive workload [11, 43], configured to represent a leaf node, executing search queries over Wikipedia.
- `masstree`, a high-performance in-memory key-value store that achieves speeds similar to memcached [32].
- `moses`, a statistical machine translation system [26], represents a real-time translation service (e.g., Google Translate).
- `shore-mt`, a modern DBMS, running TPC-C, an OLTP workload [23].
- `specjbb`, representative of the middle-tier portion of 3-tier business logic systems. We run it using the HotSpot JVM v1.5, and configure the warmup period and garbage collector to ensure that the measured portion is not dominated by JIT overheads and is free of garbage collection periods. This avoids having the JVM dominate tail latencies; commercial real-time JVMs [55] achieve a similar effect.

Table 1 details their input sets and simulated requests.

3.2. Methodology

Prior architectural studies consider similar workloads [6, 15, 18], but they execute them at full load and measure their average throughput. Instead, we are interested in their *tail latency* characteristics, which requires a different methodology.

Simulation: We simulate these workloads using zsim [46], and model a CMP with six OOO cores validated against a real Westmere system (Section 6 details the simulated system). We focus on application-level characteristics, not including OS and I/O overheads. We have verified that this is reasonable by profiling OS time, I/O time, and request distributions in a real system. For `shore-mt`, this is only accurate if the database and log are stored in a ramdisk (0.8 ms 95th pct service time in real system vs 0.9 ms simulated), or on an SSD (1.4 ms 95th pct service time in real). `shore-mt` becomes I/O-bound on a conventional HDD (86 ms 95th pct service time in real). For the other workloads, these simplifications are accurate.

Client-server setup: We integrate client and server under a single process, and measure the server running the realistic stream of requests generated by the client. A common harness throttles these requests to achieve exponential interarrival times at a configurable rate (i.e., a Markov input process, common in datacenter workloads [34]). This harness adds negligible server work (155 ns/request). Finally, this approach does not include network stack overheads. These can be relatively high and unpredictable in conventional architectures, but recent work has shown that user-level networking can reduce them to microseconds even in network-intensive workloads [24]. However, the networking stack introduces delays due to interrupt coalescing. Therefore, we model interrupt coalescing with a 50 µs timeout [36].

For the characterization experiments, we pin the application to one core, and simulate a 2 MB LLC except when noted (in line with the per-core LLC capacity of current CMPs [27]). All applications run a single server worker thread, although some applications have additional threads (e.g., `shore-mt` uses log writer threads to reduce I/O stalls). Section 3.3 qualitatively discusses multithreaded latency-critical workloads.

Metrics: While percentiles are often used to quantify tail latency [11] (e.g., 95th percentile latency), we report tail latency as the *mean latency of all requests beyond a certain percentile*. Our goal is to design adaptive systems, so it would be easy to design schemes that game percentile metrics by degrading requests beyond the percentile being measured. Instead, tail means include the whole tail, avoiding this problem. Additionally, while 99th percentile latencies are commonly used [11], we use 95th tail metrics, as gathering statistically significant 99th tail metrics would require much longer runs. Our analysis is equally applicable to higher-percentile metrics.

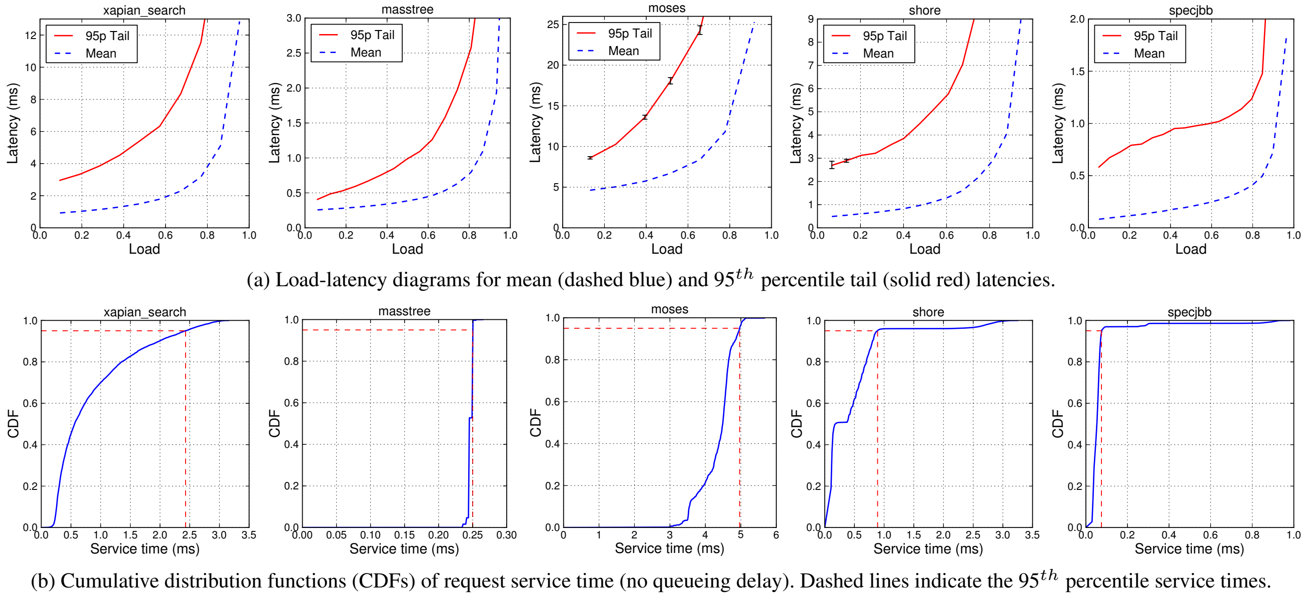
Statistical significance: Despite using 95th instead of 99th tail metrics, measuring tail latency accurately is much more expensive than measuring average performance. Intuitively, only 5% of the work influences tail latency, requiring many repeated simulations to reduce measurement error. Indeed, together, the results we present in this paper required simulating over one quadrillion (10^{15}) instructions, several orders of magnitude higher than typical architectural studies.

Each sample run executes the same requests (fixed work), but interarrival times are randomized, introducing variability and ensuring that we model a representative distribution instead of just one instance. We run enough simulations to obtain tight confidence intervals for the results we present (e.g., mean and tail latencies). However, some tail latencies are so variable that making their confidence intervals negligible would require an unreasonable number of simulations. Therefore, we omit the 95% confidence intervals when they are within $\pm 1\%$, and plot them (as error bars) otherwise.

3.3. Load-Latency Analysis

Figure 1a shows the load-latency plots for each application. Each graph shows the mean latency (blue) and tail latency (red) as a function of offered load ($\rho = \lambda/\mu$, which matches processor utilization). We make three observations that have significant implications for adaptive systems:

Observation 1: Tail $\neq$ mean. Tail latencies are significantly higher than mean latencies, and the difference between both is highly dependent on the application and load. In general, tail latencies are caused by the complex interplay of arrival



(a) Load-latency diagrams for mean (dashed blue) and 95th percentile tail (solid red) latencies.

(b) Cumulative distribution functions (CDFs) of request service time (no queueing delay). Dashed lines indicate the 95th percentile service times.

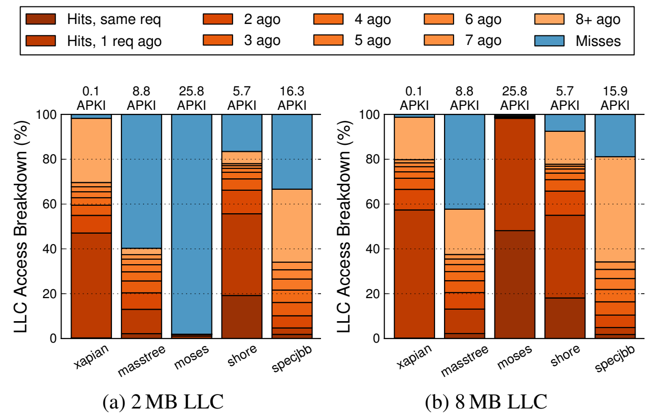
Figure 1: Load-latency and service time distributions of the latency-critical applications studied.

and service times for each request. Figure 1b shows the cumulative distribution functions (CDFs) of request service times for each workload. Some workloads (e.g., `masstree`, `moses`) have near-constant service times, but others (e.g., `xapian`, `shore-mt`, `specjbb`) are multi-modal or long-tailed. While applications with more variable service times have a larger difference between tail and mean latencies, there is no general way to determine the relationship between both. Therefore, QoS guarantees on mean performance are insufficient to guarantee tail latency efficiently. Instead, adaptive systems should be designed to guarantee tail latency directly.

Observation 2: Latency-critical workloads have limited utilization. Figure 1a shows that tail latency quickly increases with load, even in applications with regular service times. Most applications must run at 10-20% utilization if reducing latency is crucial, and cannot run beyond 60-70% load. Even if tail latency is secondary, traffic spikes and interference impose large guardbands that reduce utilization [11]. Due to this inherently low utilization, dedicating systems to latency-critical applications is fundamentally inefficient. This motivates building systems that achieve high utilization by colocating mixes of latency-critical and batch workloads.

Observation 3: Tail latency degrades superlinearly. When processor performance degrades, tail latency suffers from two compounding effects. First, requests take longer (in Figure 1a, all the curves scale up). Second, load increases (in Figure 1a, the operating point moves right). For example, `masstree` has a 1.2 ms tail latency at 60% load. If processor performance drops 25%, at the same absolute request rate (queries per second), load increases to $1.25 \cdot 60$ =75%, and tail latency grows to $1.25 \cdot TailLat(75\%) = 2.9$ ms, 2.3× higher. In adaptive systems, this places very stringent requirements on performance variability.

Note that, as configured, these applications service one request at a time in FIFO order. Servers with multiple worker threads have more nuanced tradeoffs. On one hand, being able to service multiple requests simultaneously reduces queuing delay, especially at high load. On the other hand, we have observed that server threads often interfere among themselves, block on critical sections, and in some workloads (e.g., OLTP) concurrent requests cause occasional aborts, degrading tail latency. Since providing QoS for multithreaded workloads is far more complex, and single-threaded latency-critical workloads are common in datacenters [11, 34], we defer studying multi-threaded latency-critical workloads to future work.



(a) 2 MB LLC (b) 8 MB LLC

Figure 2: Breakdown of LLC accesses for the applications studied, with 2 and 8 MB LLCs. To quantify inertia, hits are classified by how many requests ago they were last accessed.

3.4. Performance Inertia

In latency-critical applications, the performance of each request largely depends on the microarchitectural state at the start of the request (e.g., caches, TLBs, branch predictors, etc). On long requests, ignoring this dependence may be a good

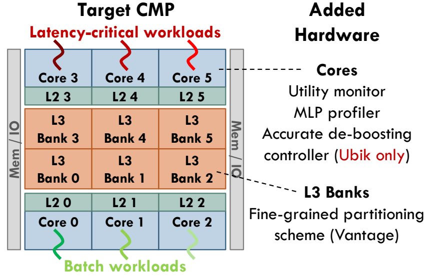


Figure 3: Target CMP and added hardware structures.

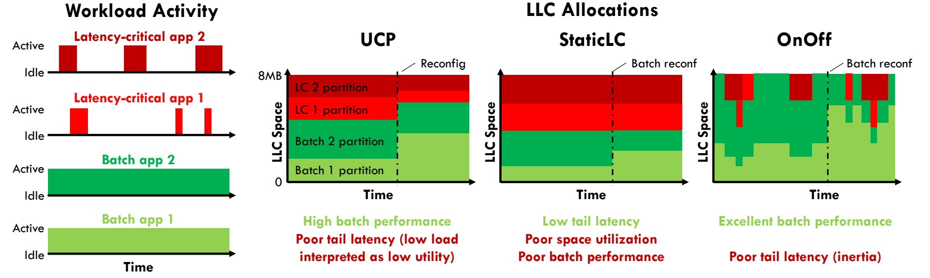


Figure 4: Example operation of UCP, StaticLC, and OnOff on a mix of two latency-critical and two batch workloads, and main properties of each partitioning policy.

approximation, but this is not the case for short requests. We call this dependence *performance inertia*. We focus on the last-level cache, the microarchitectural component with the most state. Figure 2a classifies LLC accesses into hits and misses for each application, using a 2 MB LLC. Furthermore, hits are classified according to how many requests ago they were last accessed, and shown in different shades of red (with the lowest one being during the current request, and the highest one being 8 or more requests ago). Classifying hits this way is sensible because, in these workloads, most gains can be achieved by taking space away between servicing requests, when the application is idle. To quantify intensity, each bar also shows the LLC accesses per thousand instructions (APKI).

As we can see, all applications, and especially those with shorter response times, have significant reuse across requests. In all cases, more than half of the hits come from lines that were brought in or last touched by previous requests. Moreover, this inertia largely depends on cache size. Figure 2b shows LLC access distributions for an 8 MB LLC. Compared to the 2 MB LLC, all workloads exhibit (a) significantly lower miss rates, and (b) higher cross-request reuse, often going back many requests. Thus, larger eDRAM and 3D-stacked caches are making performance inertia a growing issue for latency-critical workloads (e.g., POWER7+ has 10MB of LLC per core [58], and Haswell has up to 32 MB of L4 per core [7]).

4. Simple Cache Partitioning Policies for Mixes of Latency-Critical and Batch Workloads

Although latency-critical workloads have limited utilization, we can achieve high utilization by coscheduling mixes of latency-critical and batch applications in the same machine. Figure 3 illustrates this scenario, showing a six-core CMP with a three-level cache hierarchy (private L2s and a shared, banked L3), where three cores are executing latency-critical applications, while the other three execute batch applications.

Achieving safe operation and efficient utilization with latency-critical and batch mixes requires a *dual-goal* partitioning policy. First, it must not degrade the tail latency of latency-critical applications beyond a given bound. Second, it should maximize the average performance of batch workloads. With this approach, each latency-critical application must be given a target tail latency. In this work, we set this target to the tail latency achieved when running the workload alone with a smaller, fixed-size LLC, as in Figure 1a. This ensures that the target tail latency is achievable, and is a reasonable strategy for multi-machine deployments: operators can easily find the achievable tail latency bound, and the system will guarantee that it is not violated. In contrast, to allow high utilization, we do not set performance targets for batch workloads: since they are not latency-sensitive, they can be migrated to another machine if they are not meeting their performance target [12, 60].

Previously proposed policies focus on improving or guaranteeing long-term performance only, so they are not directly applicable to our purposes. In this section, we develop simple extensions to these policies, and show that they have serious shortcomings because they ignore transient behavior and performance inertia. Figure 4 illustrates these schemes through a simple example, and summarizes their characteristics.

UCP baseline: Figure 3 shows our baseline system and the hardware structures added to support cache partitioning. Cores can be in-order or out-of-order. As discussed in Section 2, we use Vantage [45] to perform fine-grained partitioning. We also add some monitoring structures to the core: UMONs [42] to gather miss curves efficiently (Section 2), and the simple long-miss memory-level parallelism (MLP) profiling scheme from Eyerman et al. [14].

We choose utility-based cache partitioning (UCP) enhanced with MLP information to illustrate the behavior of conventional partitioning policies. UCP maximizes overall throughput, and improves fairness as well [42]. UCP repartitions periodically (e.g., every 50 ms) by reading the per-core UMONs and MLP profilers; using the UMON data to construct miss curves, combining them with the MLP information to construct *miss-per-cycle* curves (i.e., misses per cycle the core would incur for each possible partition size) and using the Lookahead algorithm (Section 2) to find the partition sizes that minimize the expected misses per cycle in the next interval[1]. Figure 4 shows the behavior of UCP in a mix of two latency-critical and two batch workloads. UCP suffers from two problems. First, UCP maximizes cache utilization with no concern for performance bounds, and will happily degrade tail latency if doing so improves overall throughput. Second,

[1] Using MLP instead of miss curves as in UCP has been shown to provide modest performance improvements [30, 37], because UCP already accounts for miss intensity indirectly. We include this mechanism in fairness to Ubik, which uses MLP profiling to derive transient behavior.

UCP interprets the low average utilization of latency-critical workloads as low utility, and assigns them smaller partitions. Other partitioning techniques, even those that strive to maintain average IPCs, also suffer from this issue [37, 42, 49, 61].

StaticLC — Safe but inefficient: An easy solution to these problems is to give a fixed-size partition to each latency-critical application, and perform UCP on the batch applications' partitions only. We call this policy *StaticLC*. Figure 4 shows its behavior when both latency-critical applications have target sizes of 2 MB (25% of the cache) each. Unlike prior policies (Section 2), StaticLC safely maintains tail latencies and improves batch application performance somewhat, but it wastes a lot of space, as latency-critical applications hold half of the cache even when they are not active (i.e., most of the time).

OnOff — Efficient but unsafe: To improve space utilization, we develop the *OnOff* policy, which operates as follows: whenever a latency-critical application is active (i.e., *on*), it gets its full target allocation; when it goes idle (i.e., *off*), its space is reassigned to the batch partitions, which share the space using UCP. Using UCP's reconfiguration algorithm every time a latency-critical application shifts between idle and active would be expensive, so at periodic intervals (as in the UCP baseline), we precompute the batch partition sizes for all possible allocations ($N+1$ cases with N latency-critical workloads). Figure 4 shows OnOff's behavior. Note how the latency-critical applications only get allocated space when active, as shown on the left. OnOff gives space to batch apps aggressively, solving StaticLC's shortcoming. Unfortunately, Section 3 showed that latency-critical apps have significant *cross-request reuse*, so taking away their allocations when they are idle degrades their tail latency.

In summary, both StaticLC and OnOff have significant drawbacks, which Section 7 quantifies. As we will see now, we can do much better by understanding and leveraging transients instead of avoiding or ignoring them.

5. Ubik: Inertia-Based Cache Management

Ubik leverages the insight that the transient behavior caused by resizing a partition can be derived analytically, and uses it to manage the partition sizes of latency-critical apps more aggressively than StaticLC. When a latency-critical app goes idle, Ubik downsizes its partition below its *target size* while providing a high-level guarantee: when the app goes from idle to active, after a configurable time, which we call the *deadline*, both its overall progress and performance will be the same as if the partition size had been kept at the target instead of downsized. By setting the deadline to the 95th percentile latency at the target size, Ubik does not degrade tail behavior (it can, however, make short requests slower, which reduces variability). Ubik achieves this by determining how much to downsize safely, and by temporarily boosting the partition beyond its target size when the app is active to make up for the lost performance. Idle periods are common, so Ubik frees a significant fraction of cache space, and uses it to improve batch app throughput. We have developed two variants of Ubik: *strict Ubik*, which makes conservative assumptions to provide strict guarantees, and *Ubik with slack*, which relaxes these assumptions to manage space more aggressively with a graceful, controllable degradation in tail latency.

5.1. Strict Ubik

Transient behavior: When we increase a partition's target size, its application takes some time to fill the extra space. During this time, the partition's hit rate is lower than when it reaches its target size. Ubik relies on this transient behavior being analyzable. In general, this is not true for all partitioning schemes. For example, in way-partitioning, when an application is given an extra way, it has to have misses in all the sets to claim the way. This heavily depends on the application's access pattern, and although prior work in analytical modeling of set-associativity has approximated them [1, 53], transients cannot be accurately derived online.

To avoid these issues, Ubik uses Vantage [45]. In Vantage, transients happen *as fast as possible* and are *independent of the application's access pattern*: when a partition is upsized, nothing is evicted from that partition until it reaches its target size. Every miss in that partition grows its size by one line, evicting lines from partitions that are being downsized (through Vantage's two-stage demotion-eviction process [45]). Vantage leverages the statistical properties of zcaches [44] to guarantee that a growing partition has a negligible probability of suffering an eviction (about once in a million accesses) independently of the access pattern, so we can safely assume that no line is evicted until the partition reaches its target[2].

Under these conditions, transients are easy to predict. Specifically, assume that we upsize a partition from s_1 to s_2 cache lines ($s_2 > s_1$). Figure 5 illustrates this process. We are interested in two quantities: how long does the transient last ($T_{transient}$), and how many cycles do we lose compared to not having the transient, i.e., if we started at s_2 lines (L).

First, consider what happens at a specific size s. If the probability of experiencing a miss is p_s, the average time between accesses is $T_{access} = c + p_s \cdot M$, where M is the average number of cycles that the processor stalls due to a cache miss, and c would be the cycles between cache accesses if all accesses were hits. We can find all these quantities from basic performance counters and the MLP profiler. For example, suppose the core has $IPC = 1.5$, issues 5 L3 accesses per thousand instructions, and 10% of them miss. The MLP profiler computes M directly [14] (e.g., $M = 100$ cycles); $p_s = 0.1$, and $T_{access} = c + 0.1 \cdot 100 = 1000/(5 \cdot 1.5) = 133$ cycles, so $c = 123$ cycles. Moreover, the UMON produces a miss curve, which we can express as a *miss probability curve*, getting p_s for all sizes. Using these, we can find the transient's length.

Since only a fraction p_s of accesses miss, the average time between two consecutive misses is $T_{miss,s} = T_{access,s}/p_s = c/p_s + M$. To find the duration of a transient, we simply need

[2] In general, Ubik can be used with any analyzable partitioning scheme; Section 7 evaluates Ubik under different partitioning schemes and arrays.

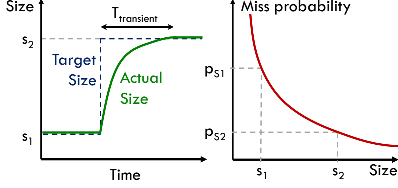


Figure 5: Transient length can be lower-bounded using miss curves and simple performance counters.

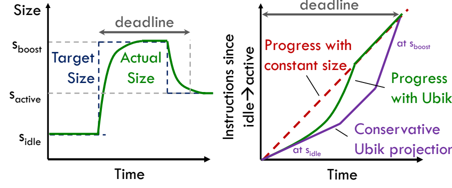


Figure 6: Example transient in Ubik: target and actual sizes, and execution progress compared to a constant partition size.

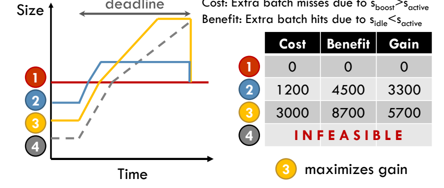


Figure 7: Sizing a latency-critical partition: determining s_{boost} for feasible values of s_{idle}, and cost-benefit analysis.

to add these intervals up:

$$T_{transient} = \sum_{s=s_1}^{s_2-1} T_{miss,s} = \sum_{s=s_1}^{s_2-1} \frac{c}{p_s} + M$$

Note that p_s is a variable: it decreases as the partition grows, as shown in Figure 5. Though this summation is complex, we can easily compute a conservative upper bound:

$$T_{transient} \leq \sum_{s=s_1}^{s_2-1} \frac{c}{p_{s_2}} + M = (s_2 - s_1)\left(\frac{c}{p_{s_2}} + M\right)$$

For the example above, if a transient starts at $s_1 = 1\,\text{MB}$ and ends at $s_2 = 2\,\text{MB}$ ($s_2 - s_1 = 16384$ 64-byte cache lines), $p_{s_1} = 0.2$, and $p_{s_2} = 0.1$, then the transient will take at most $16384 \cdot (123/0.1 + 100) = 21.8$ million cycles.

During this transient, we incur $\sum_{s=s_1}^{s_2-1} (1 - p_{s_2}/p_s)$ more misses than if we started at size s_2, and each miss costs M cycles on average. Therefore, we can compute the number of lost cycles, L, as well as a conservative upper bound:

$$L = M \sum_{s=s_1}^{s_2-1} 1 - \frac{p_{s_2}}{p_s} \leq M\,(s_2 - s_1)\left(1 - \frac{p_{s_2}}{p_{s_1}}\right)$$

This upper bound assumes that the application does not enjoy the extra reuse from a larger partition until it finishes the transient. For the example above, we lose at most $100 \cdot 16384 \cdot (1 - 0.5) = 819$ thousand cycles.

As we can see, the transient duration mainly depends on the miss rate, while the cycles lost depend on the difference between miss rates. Thus, Ubik works best with cache-intensive workloads that are mildly sensitive to partition size.

5.1.1. Managing latency-critical applications

Boosting: Ubik's goal is to give the same performance as having a constant partition size, which we call the *active* size, s_{active}. In strict Ubik, s_{active} is the desired target size. When the app goes idle, Ubik downsizes its partition to its *idle* size, s_{idle}. When the app becomes active, to make up for the lost performance, Ubik upsizes its partition to the *boost* size, $s_{boost} > s_{active}$. Once the application has recovered the cycles it lost in the transient, Ubik downsizes it to s_{active} until the app goes idle again. Figure 6 shows this behavior.

Ubik periodically computes the best s_{idle} and s_{boost} for each partition. Figure 7 illustrates this process. Ubik evaluates N options for the idle size: $s_{idle} = s_{active}$, $s_{idle} = s_{active}(N-1)/N$, ..., $s_{idle} = 0$ (we set $N = 16$ in our experiments; Figure 7 shows $N = 4$). For each option, Ubik computes the s_{boost} needed so that, by the deadline, the number of cycles gained by being at s_{boost} instead of at s_{active} equals the upper bound on lost cycles at the transient. We limit s_{boost} to $s_{boost,max}$ = total lines/latency-critical apps. This way, latency-critical apps never interfere with each other (even when all are boosted). The lower s_{idle} is, the higher s_{boost}. Ubik stops evaluating options when either the transient is too long, or the needed s_{boost} is too high. For example, in Figure 7 Ubik stops at option 4, which is too aggressive. Then, among the feasible options, Ubik performs a simple cost-benefit analysis to determine the best one. The benefit is the extra hits that batch apps would gain by having $s_{active} - s_{idle}$ more lines available when the app is idle, and the cost is the extra misses they incur by having $s_{boost} - s_{active}$ fewer lines for *deadline* cycles. These are computed using the batch apps' miss curves. For example, in Figure 7 Ubik chooses option 3 ($s_{idle} = s_{active}/2$), which yields the maximum gain.

Accurate de-boosting: By using upper bounds on both the transient length and the lost cycles, Ubik downsizes conservatively. Therefore, most requests have faster transients, and regain the performance lost by downsizing much earlier than the deadline. Waiting until the deadline to downsize from s_{boost} to s_{active} would improve the latency-critical application's performance unnecessarily while hurting batch throughput.

Fortunately, we can leverage the UMON to accurately determine when an application has made up for its lost performance. UMON tags are not flushed when the app goes idle, so UMONs can already track (through hit counters [42]) how many misses the *current request* would have incurred if we had not downsized its partition. Therefore, we simply add a counter that uses UMON events to track this; when the count exceeds the current number of misses (plus a guard to account for the small UMON sampling error [42]), the application has made up for its losses. This triggers an interrupt, and the Ubik runtime deboosts the app and gives the space back to batch apps.

5.1.2. Managing batch applications

Ubik manages batch app partitions at two granularities. First, as in UCP, Ubik repartitions the cache space available to batch apps (but not latency-critical apps) at periodic, coarse-grained intervals (e.g., 50 ms) using the Lookahead algorithm. Second, whenever a latency-critical app partition is resized (e.g., on an *idle→active* transition), Ubik takes space from or gives it to batch apps. Using the Lookahead algorithm on these frequent resizings would be too expensive. Additionally, it is infeasible to precompute batch partition sizes for all cases, as it is done in OnOff, because there are many more possibilities.

Instead, on each coarse-grained reconfiguration interval,

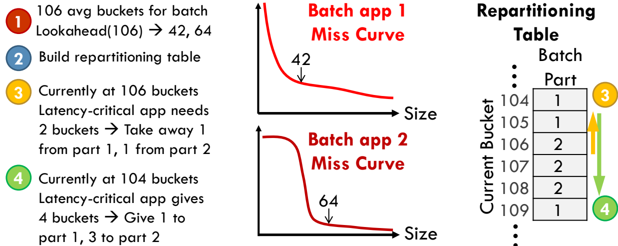


Figure 8: Building and using the repartitioning table, which provides fast incremental reallocations for batch workloads.

the Ubik runtime computes the average cache space that was available to batch apps in the previous interval, and performs Lookahead on this average size to determine the baseline batch allocations. Then, it builds a *repartitioning table* that stores what partitions to give space to or take space from when the space available to batch apps changes. We quantize partition sizes to "buckets", $1/B^{th}$s of cache size ($B = 256$ in our evaluation), so the table has B entries. The table is built greedily: for each possible batch budget, growing a latency-critical partition takes space away from the batch app that gets the lowest marginal utility from its allocation. Figure 8 shows how this table is built and used. When the runtime needs to resize a latency-critical partition, it walks this table from the current to the target buckets to find what batch partitions to take space from or give it to. This makes repartitions fast, and, despite the shortcomings of greedy partitioning with non-convex miss curves [4, 42], it works well in practice because the space available to batch apps is often close to the average.

5.1.3. Putting it all together

Ubik is mostly implemented as a software runtime, requiring minimal hardware extensions over conventional partitioning schemes, namely the MLP profilers and the accurate de-boosting mechanism. Periodically, the runtime reads the UMONs and performance counters, computes the best idle and boost sizes for latency-critical apps (Section 5.1.1), finds the batch app sizes using Lookahead, and builds the repartitioning table (Section 5.1.2). Overall, this process takes a few tens of thousands of cycles. Since this happens at a coarse granularity (every 50 ms), it incurs minimal overheads. Latency-critical apps call into the Ubik runtime when they become idle and active (this does not happen after every request; an app only idles when it runs out of requests to process). On each such event, Ubik resizes the latency-critical partition (to s_{idle} or s_{boost}) and uses the repartitioning table to find the batch apps to give or take space from. On an idle-active transition, the runtime also arms the accurate de-boosting circuit, which triggers an interrupt when the cost of the transient has been recovered. Ubik then downsizes the app to s_{active}, and gives the space to batch apps using the repartitioning table. These resizings are fast (hundreds of cycles), imposing negligible overheads.

Overall, hardware costs from partitioning are small. For the six-core CMP in Figure 3, Vantage adds a 3-bit partition ID to each LLC tag (0.52% state overhead), 256 bits of state per partition and bank (0.009% state overhead for 2 MB banks), and minimal, off-the-critical-path logic for the replacement process [45]. Each UMON adds 2 KB (256 tags) of state, and is rarely accessed (only one in 768 accesses is inserted into the UMON). Ubik's accurate de-boosting mechanism requires minimal changes to UMONs: a comparator, an MSR to control it, and an input to the interrupt controller.

5.2. Ubik with Slack

Ubik as described so far takes conservative decisions to avoid degrading tail latency. Some applications have little to gain from sizes larger than the target size, and in this case, strict Ubik is conservative, often not downsizing at all. However, often these applications also have little to lose from having less space. For example, Figure 2a shows that, at 2 MB, `moses` barely gets any LLC hits, despite being memory-intensive. For these cases, by slightly relaxing the tail latency requirement, we can free a large amount of space. This enables Ubik to smoothly trade off tail latency degradation for higher batch app throughput.

We relax the tail latency requirements by allowing a configurable amount of tail latency degradation, expressed as a fraction of the strict deadline, which we call the slack. Ubik uses this slack on tail latency to determine a *miss slack*, i.e., a bound on the additional misses that can be sustained in the course of a request while maintaining the tail latency within the desired slack. The miss slack is computed adaptively using a simple proportional feedback controller that accepts individual request latencies as input.

Given a miss slack, Ubik uses the miss curve to set s_{active} to a value lower than the given target size such that the additional misses incurred at this size are within the miss slack. s_{boost} and s_{idle} are then determined relative to this new s_{active} as in Section 5.1.1. Since the sizing of s_{active} relative to the target size is contrained only by the relative difference in miss rates but not by any transient lengths, s_{active} can be much lower than the target for apps that are not cache-sensitive, even if they are not very cache-intensive.

Unfortunately, with slack, rare requests that might not impact the miss curve much can suffer significantly, especially since s_{active} may be much smaller than in the no-slack case. To detect these requests, we add a low watermark to the de-boosting circuit: if, after reaching s_{boost}, the current misses outgrow the UMON-measured misses by a factor of $(1 + miss\ slack)$, we trigger an interrupt and conservatively use the no-slack s_{boost} and s_{active}. This avoids catastrophic degradation for these cases. Further, the adaptive miss slack mechanism mentioned earlier allows Ubik to quickly adapt s_{active} in response to changing application behavior.

6. Methodology

Modeled system: As in Section 3, we use zsim [46] to model CMPs with 6 cores and a 3-level cache hierarchy, shown in Figure 3, with parameters given in Table 2. This configuration closely models a Westmere-EP processor. We use out-of-order

Cores	6 x86-64 cores, Westmere-like OOO [46], 3.2 GHz
L1 caches	32 KB, 4-way set-associative, split D/I, 1-cycle latency
L2 caches	256 KB private per-core, 16-way set-associative, inclusive, 7-cycle latency
L3 cache	6 banks, 2 MB/bank (12 MBs total), 4-way 52-candidate zcache, 20 cycles, inclusive
Coherence protocol	MESI protocol, 64B lines, in-cache directory, no silent drops; TSO
Memory	200-cycle latency

Table 2: Configuration of the simulated 6-core CMP.

core models by default (validated against a real Westmere system [46]), and also include results with simple core models.

Since we focus on cache capacity partitioning and dealing with the effects of performance inertia, we model a fixed-latency LLC and main memory, which do not include bandwidth contention. Bandwidth has no inertia, so Ubik should be easy to combine with bandwidth partitioning techniques for real-time systems [21]. We leave such an evaluation to future work. The memory latency matches a Xeon W3670 running at 3.2 GHz with 3 DDR3-1066 channels at minimal load.

LLC configurations: We use private per-core 2 MB LLCs (which provide perfect isolation) as a baseline, and evaluate a shared 12 MB LLC with different schemes: LRU (without partitioning), UCP, StaticLC, OnOff (Section 4), and Ubik. All partitioning policies use Vantage configured as in [45], reconfigure every 50 ms, use 32-way, 256-line UMONs (2 KB each), and linearly interpolate the 32-point UMON miss curves to 256 points for finer granularity.

Workload mixes: We simulate mixes of latency-critical and batch apps. Each six-core mix has three latency-critical and three batch apps. Apps are pinned to cores. We use the five latency-critical apps described in Section 3. First, we run each app alone with a 2 MB LLC, and find the request rates that produce 20% and 60% loads. We then use each app in two configurations: with the 20% request rate to simulate a low-load case, and the 60% rate to simulate a high-load case. We test each case with a wide and representative set of batch apps. Each mix has three instances of the same latency-critical workload, with each instance serving a different set of requests. Both requests and arrival times are different for each instance.

To produce a wide variety of batch app mixes, we use a methodology similar to prior partitioning work [42, 45]. We classify all 29 SPEC CPU2006 workloads into four types according to their cache behavior: insensitive (`n`), cache-friendly (`f`), cache-fitting (`t`), and streaming (`s`) as in [45, Table 2], and build random mixes of all the 20 possible combinations of three workload types (e.g., `nnn`, `nnf`, `nft`, and so on). We generate two mixes per possible combination, for a total of 40 mixes. We then simulate all combinations of the 10 three-application latency-critical mixes and the 40 three-application batch mixes to produce 10×40=400 six-application mixes.

To run each mix, we fast-forward each application to the beginning of its *region of interest* (ROI). For each latency-critical app, the ROI starts after processing a fixed number of warmup requests, and covers the number of requests in Table 1. This number has been chosen so that, at 20% load, the ROI takes around 5 billion cycles. For batch apps, we first run each app in isolation with a 2 MB LLC, and measure the number of instructions I_i executed in 5 billion cycles. The ROI starts after fast-forwarding 10 billion instructions, and covers I_i instructions. In each experiment we simulate the full mix until all apps have executed their ROIs, and keep all apps running. We only consider each app's ROI when reporting aggregate metrics. Though involved, this multiprogrammed methodology is *fixed-work* and *minimizes sample imbalance*, allowing unbiased comparisons across schemes [19].

Metrics: For each mix, we report two metrics. For latency-critical apps, we report *tail latency degradation*, defined as the 95th percentile tail latency (Section 3.2) across all three app instances, normalized to the latency of the same instances running in isolation. For batch apps, we report weighted speedup, $(\sum_i IPC_i/IPC_{i,alone\,on\,2MB\,LLC})/N_{apps}$ [42, 51], which represents the multiprogrammed speedup that batch applications achieve vs having private LLCs. To achieve statistically significant results, we introduce small amounts of non-determinism [2], randomize request arrival times, and perform enough runs to achieve 95% confidence intervals $\leq$3% on all individual runs. Most individual runs and all aggregate metrics have confidence intervals within 1%.

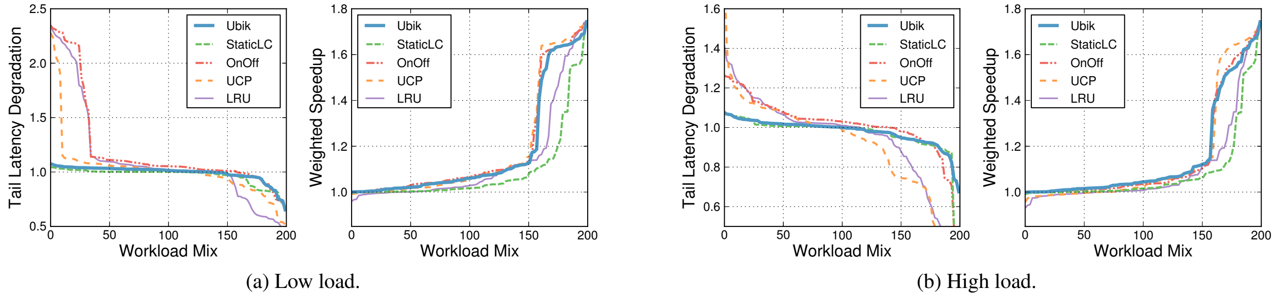
As discussed in Section 3, measuring tail latencies accurately requires many runs, making our evaluation compute-intensive. For example, Figure 9 summarizes 145 trillion simulated instructions. For this reason, we evaluate Ubik on small CMPs. Ubik should apply to large-scale CMPs with tens to hundreds of cores, but we leave that evaluation to future work.

7. Evaluation

7.1. Comparison of LLC Management Schemes

Figure 9 shows the distributions of both tail latency degradation for the latency-critical apps (lower is better) and weighted speedup for the batch apps (higher is better) in each mix. Figure 9a contains the mixes with low-load latency-critical apps, and Figure 9b contains the high-load mixes. On each graph, each line represents a single scheme. For each scheme, mixes are sorted from worst to best (ascending weighted speedups, descending tail latency degradations). Mixes are sorted independently for each line, so these graphs concisely summarize each scheme, but should not be used for mix-by-mix comparisons. In this section, Ubik runs with a slack of 5% (Section 7.2 explores other slack values).

Figure 9a shows that LRU, UCP, and OnOff severely degrade tail latency on a large fraction of the mixes: LRU and OnOff suffer significant degradation for about 20% of the mixes, with many of these being in excess of 2×. While UCP maintains tail latencies within acceptable bounds for a larger fraction of mixes, the worst case degradations for it are just as bad as in LRU and OnOff, about 2.2×. These severe deadline violations make these policies unsuitable for



(a) Low load.

(b) High load.

Figure 9: Distribution of tail latency degradation (latency-critical apps, lower is better) and weighted speedup (batch apps, higher is better) for all schemes vs a private LLC baseline, for the 400 mixes simulated, split into low and high loads (200 mixes each).

	LRU	UCP	OnOff	StaticLC	Ubik
Low load	13.1%	18.3%	18.3%	8.9%	17.1%
High load	9.8%	14.7%	14.5%	8.3%	14.8%

Table 3: Average weighted speedups (%) for all schemes.

latency-critical workloads. In contrast, our proposed StaticLC and Ubik schemes have negligible latency degradation. Most mixes match the tail latency of private LLCs, and some are slightly worse [3]. Moreover, as expected, Ubik's tail latencies are within 3% of StaticLC's (consistent with Ubik's 5% slack).

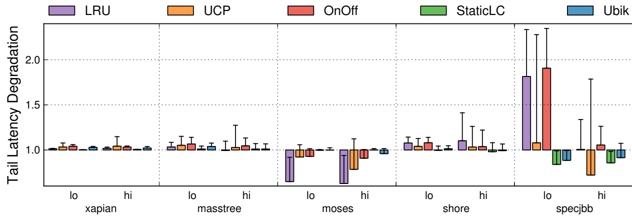
Tail latencies for the high-load mixes look similar, with Ubik and StaticLC maintaining a tight upper bound on tail latency degradation, while LRU, UCP, and OnOff experience significant degradation: up to 60% for UCP, and 40% for both LRU and OnOff. Though still unacceptably high, these degradations are *better* than in the low-load mixes. Two competing effects are at play here: on one hand, tail latency is more sensitive to performance at high load (Section 3); on the other hand, at higher utilization, latency-critical apps "defend" their cache working sets better in LRU and OnOff, and UCP sees higher average utility, giving them larger partitions. Overall, the second effect dominates, and running latency-critical apps at higher loads gives more stable (albeit higher) tail latencies with these best-effort schemes. As we will see later, which effect dominates depends on the specific latency-critical app.

Figure 9 also shows the distributions of weighted speedups for batch apps, and Table 3 reports their averages. OnOff and UCP perform best for batch apps. LRU enjoys the extra capacity stolen from latency-critical apps, but cannot use it as effectively due to the lack of partitioning among batch apps, performing sensibly worse than OnOff and UCP. StaticLC achieves the smallest weighted speedups, as it always limits batch apps to half of the cache (though, by using UCP on batch apps, StaticLC improves throughput over private LLCs). Finally, Ubik has weighted speedups competitive with OnOff and UCP, and outperforms LRU, *while still meeting deadlines*. Overall, weighted speedups are modest because many batch apps in our mixes are not memory-intensive. As Figure 9 shows, about 23% of the mixes enjoy weighted speedups of over 30%, with improvements of up to 74%.

Per-application results: To gain more insights into these results, Figure 10 shows the tail latency degradation and average weighted speedup separately for each latency-critical workload in their low and high load scenarios. In the tail latency graph, each bar shows the tail latency degradation across all 40 batch mixes for a specific latency-critical app and load, and the whisker shows the mix with the worst tail degradation. Intuitively, these graphs can be interpreted as follows. Suppose we have a 40-machine cluster. We run three instances of the same latency-critical app in every machine, and spread the 40 batch app mixes across machines. We then issue requests to all the latency-critical apps uniformly, and measure the tail latency of all response times from all the machines considered together. Each bar shows how much worse the *global* tail latency is vs having private LLCs, and the whisker shows the tail latency of the worst-performing machine. The weighted speedup plots show the additional batch throughput vs having private LLCs, over the whole cluster.

Figure 10 shows large differences across latency-critical apps. First, `xapian` has very low LLC intensity (Section 3), so in the low-load case, all techniques achieve similar tail latency. UCP and Ubik achieve the highest weighted speedups by aggressively downsizing `xapian` partitions. StaticLC is excessively conservative, but even OnOff is wasteful, as it gives `xapian` its full allocation when it's on. However, at high loads `xapian` becomes very sensitive to performance changes (Figure 1a shows its tail latency has a steep slope beyond 60%), and UCP degrades `xapian`'s tail latency by up to 20% in some cases. `masstree` is mildly memory-intensive and has cross-request reuse, but has high MLP as well, which somewhat mitigates its tail latency degradation under LRU, UCP, and OnOff. The three best-effort schemes suffer deadline violations of up to 20% in the low-load case and up to 35% in the high-load case. `moses` is very memory-intensive and has no reuse at 2 MB, but starts to have significant reuse at around 4 MB. Interestingly, UCP detects this and gives `moses` a large fraction of the cache, and with LRU, `moses` naturally grabs the space from batch apps. Both schemes achieve tail laten-

[3] These differences are an artifact of our baseline being private caches instead of a statically sized partition. Because a Vantage partition has very high associativity [45], these differences are overall positive.



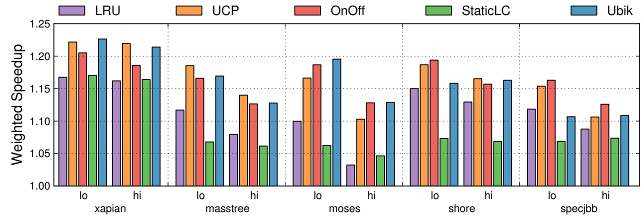
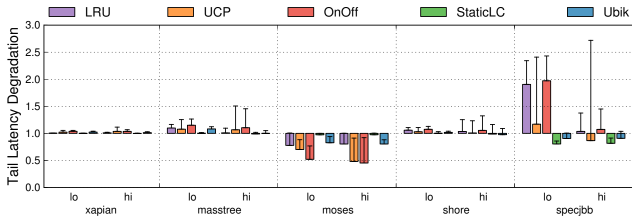


Figure 10: Tail latency degradation (lower is better) and weighted speedup (higher is better) of all schemes vs a private LLC baseline, for the 400 mixes simulated, split by latency-critical app and load, using OOO cores. The left graph shows both overall (bar) and worst-mix (whisker) tail latencies; the right graph shows average weighted speedups.



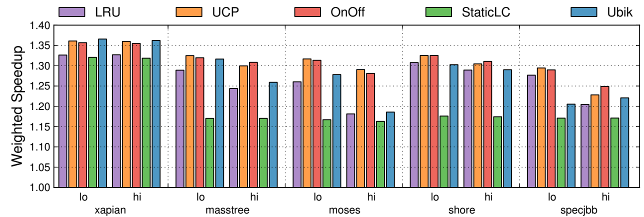


Figure 11: Results with simple in-order (IPC=1) cores: tail latency degradation (lower is better) and weighted speedup (higher is better) of all schemes vs a private LLC baseline, for the 400 mixes simulated, split by latency-critical app and load.

cies well below our target, which is unnecessary and degrades batch throughput. Additionally, UCP also suffers occasional deadline violations of up to 20%. Ubik, on the other hand, maintains `moses`'s target tail latency, and achieves the highest batch throughput of all schemes. Finally, `shore-mt` and `specjbb` have significant cross-request reuse (Figure 2) and are memory-intensive. LRU, OnOff, and UCP experience severe deadline violations for these two apps. On the other hand, StaticLC and Ubik maintain their tail latencies by protecting their working sets when they go idle. Both UCP and OnOff achieve high weighted speedups, but these come at the cost of significant tail latency degradation. Even in these sensitive cases, Ubik achieves significant throughput improvements over StaticLC by exploiting transients and boosting.

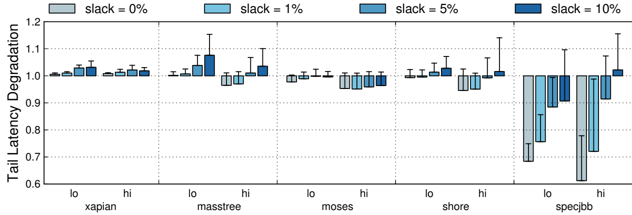
In summary, Ubik achieves the best balance between maintaining tail latency and improving batch throughput of all the schemes. Ubik achieves the highest batch throughput of all schemes on `xapian` and `moses`, and achieves a somewhat smaller throughput than UCP and OnOff on `masstree`, `shore-mt`, and `specjbb`, but unlike UCP, OnOff, and LRU, it maintains their tail latency.

Utilization: Our end goal is to improve datacenter utilization. With a few conservative assumptions, we can approximate the utilization gains of StaticLC and Ubik. First, since low tail latencies are often critical [11], assume the datacenter executes latency-critical apps at 20% (low) load. Current CMPs implement LRU, so to avoid high tails, the conventional approach is to not coschedule any batch applications. Assume we can use half of the cores without degrading the tail much (which is somewhat optimistic, since in many cases even two latency-critical applications will interfere with each other). In this case, we achieve 10% utilization, which matches reported numbers from industry [35]. In contrast, StaticLC and Ubik can safely schedule mixes across all the machines, achieving 60% utilization, 6× higher than LRU. Beyond utilization, both schemes significantly improve throughput for batch applications.

In-order cores: Many datacenter workloads have vast request-level parallelism, so using simpler cores can be a sensible way to improve datacenter efficiency [43, 50, 56]. To see if our proposed techniques apply to simpler cores, Figure 11 shows tail latencies and weighted speedups when using simple in-order core models (IPC=1 except on L1 misses) instead of OOO cores. In-order cores are more sensitive to memory access latency, as they cannot hide stalls. Therefore, LRU, UCP, and OnOff degrade tail latency more severely, being up to 2.7× worse. In contrast, StaticLC and Ubik both avoid significant tail latency degradation despite the higher sensitivity. The added sensitivity to cache misses also translates into higher weighted speedups across all schemes: StaticLC achieves an average weighted speedup of 20%, while Ubik significantly outperforms it with a weighted speedup of 28%. LRU, UCP, and OnOff achieve weighted speedups of 27%, 30%, and 30%, respectively. All schemes also achieve higher maximum weighted speedups (2.44×).

7.2. Sensitivity to Slack

As discussed in Section 5, by introducing slack, Ubik can smoothly trade off tail latency for batch throughput. Figure 12 shows tail latencies and weighted speedups when Ubik uses slacks of 0%, 1%, 5%, and 10% (so far, we have discussed results using a 5% slack). Note that the scale on the y-axis for the tail latency degradation chart in Figure 12 is much smaller than in Figure 11 and Figure 10. With no slack, Ubik strictly maintains tail latency and achieves an average weighted speedup of 9.9%. Larger slacks improve batch throughput while maintaining tail latencies within the specified bounds. With slack



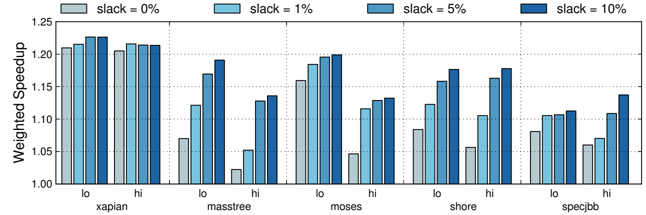
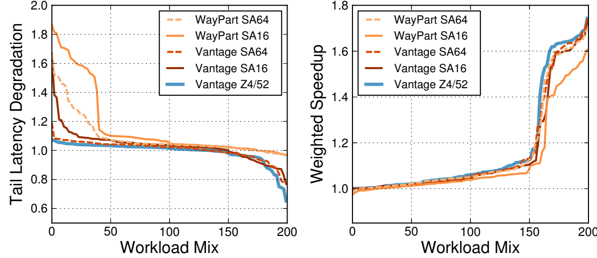
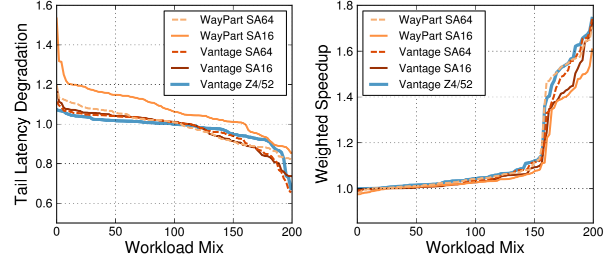


Figure 12: Results for Ubik with 0%, 1%, 5% and 10% slack: tail latency degradation (lower is better) and weighted speedup (higher is better) vs a private LLC baseline, for the 400 mixes simulated, split by latency-critical app and load, using OOO cores.



(a) Low load.



(b) High load.

Figure 13: Distribution of tail latency degradation (latency-critical apps, lower is better) and weighted speedup (batch apps, higher is better) vs a private LLC baseline for Ubik (5% slack) with different partitioning schemes and arrays: way-partitioning and Vantage on set-associative caches of 16 (SA16) and 64 (SA64) ways, and Vantage on the default 4-way 52-candidate zcache.

values of 1%, 5% and 10%, weighted speedup improves to 13.1%, 16.0% and 17.0%, respectively. Therefore, with an adjustable slack, Ubik dominates all the other schemes evaluated. We find that, in practice, a small value of slack (e.g., 5%) achieves a good trade-off between maintaining tail latencies and improving batch performance.

7.3. Sensitivity to Partitioning Scheme

Ubik relies on fine-grained partitioning with fast, analyzable transients. So far, we have presented results using Vantage on a 4-way, 52-candidate zcache, which satisfies both properties. Figure 13 characterizes Ubik using way-partitioning and Vantage on 16 and 64-way set-associative caches (SA16/SA64).

Way-partitioning suffers from three problems: lower associativity, coarse-grained partitions, and longer transients. With the way-partitioned 16-way cache, Ubik has limited choice in partition sizes and observes much longer transients, so tail latencies and weighted speedup are significantly degraded. The 64-way cache ameliorates the associativity and granularity problems, resulting in comparable throughput to Vantage on zcaches. However, transients are still longer and unpredictable, so tail latencies are still worse by up to 60%.

Ubik works better with Vantage on set-associative caches. Vantage works well in set-associative arrays, but it loses its analytical guarantees [45] and becomes a soft partitioning scheme. On the 16-way cache, frequent forced evictions hurt both tail latency (by up to 45%) and batch throughput. However, with 64 ways, Vantage has enough associativity to achieve almost the same throughputs and tail latencies as with a zcache. Though rare, the 64-way cache still suffers from occasional tail latency degradation of up to 20%.

In conclusion, while Ubik misses deadlines with way-partitioning due to its lack of predictability, Ubik is usable with set-associative caches partitioned with Vantage, although at the cost of high associativity and occasional deadline violations.

8. Conclusions

We have identified the problem of performance inertia, and characterized its impact in latency-critical applications. We have shown that, in systems that execute mixes of batch and latency-critical applications, prior cache partitioning techniques can be adapted to provide capacity isolation and improve cache utilization, but ignoring inertia sacrifices significant batch performance to guarantee isolation of latency-critical workloads. We have used these insights to design Ubik, a partitioning technique that leverages transient behavior to dynamically manage the allocations of latency-critical workloads without degrading their tail latencies, providing both high cache utilization and strict QoS guarantees on mixes of latency-critical and batch workloads.

Acknowledgments

We sincerely thank Nathan Beckmann, Christopher Fletcher, and the anonymous reviewers for their helpful feedback on prior versions of this manuscript. This work was supported in part by DARPA PERFECT under contract HR0011-13-2-0005 and by NSF grant CCF-1318384.

References

[1] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Transactions on Computer Systems*, 7(2), 1989.

[2] A. Alameldeen and D. Wood. IPC considered harmful for multiprocessor workloads. *IEEE Micro*, 26(4), 2006.
[3] L. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40(12):33–37, 2007.
[4] N. Beckmann and D. Sanchez. Jigsaw: Scalable Software-Defined Caches. In *Proc. PACT-22*, 2013.
[5] S. Bird and B. Smith. PACORA: Performance aware convex optimization for resource allocation. In *Proc. HotPar-3*, 2011.
[6] E. Blem, J. Menon, and K. Sankaralingam. Power Struggles: Revisiting the RISC vs CISC Debate on Contemporary ARM and x86 Architectures. In *Proc. HPCA-16*, 2013.
[7] R. Brain, A. Baran, N. Bisnik, et al. A 22nm High Performance Embedded DRAM SoC Technology Featuring Tri-Gate Transistors and MIMCAP COB. In *Proc. of the Symposium on VLSI Technology*, 2013.
[8] B. D. Bui, M. Caccamo, L. Sha, and J. Martinez. Impact of cache partitioning on multi-tasking real time embedded systems. In *Proc. RTCSA-14*, 2008.
[9] D. Chiou, P. Jain, L. Rudolph, and S. Devadas. Application-specific memory management for embedded systems using software-controlled caches. In *Proc. DAC-37*, 2000.
[10] H. Cook, M. Moreto, S. Bird, et al. A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness. In *Proc. ISCA-40*, 2013.
[11] J. Dean and L. A. Barroso. The tail at scale. *Communications of the ACM*, 56(2):74–80, 2013.
[12] C. Delimitrou and C. Kozyrakis. Paragon: QoS-Aware Scheduling for Heterogeneous Datacenters. In *Proc. ASPLOS-18*, 2013.
[13] E. Ebrahimi, C. J. Lee, O. Mutlu, and Y. N. Patt. Fairness via source throttling: A configurable and high-performance fairness substrate for multi-core memory systems. In *Proc. ASPLOS-15*, 2010.
[14] S. Eyerman, L. Eeckhout, T. Karkhanis, and J. E. Smith. A performance counter architecture for computing accurate CPI components. In *Proc. ASPLOS-12*, 2006.
[15] M. Ferdman, A. Adileh, O. Kocberber, et al. Clearing the clouds: a study of emerging scale-out workloads on modern hardware. In *Proc. ASPLOS-17*, 2012.
[16] B. Grot, J. Hestness, S. W. Keckler, and O. Mutlu. Kilo-NOC: a heterogeneous network-on-chip architecture for scalability and service guarantees. In *Proc. ISCA-38*, 2011.
[17] F. Guo, Y. Solihin, L. Zhao, and R. Iyer. A framework for providing quality of service in chip multi-processors. In *Proc. MICRO-40*, 2007.
[18] N. Hardavellas, M. Ferdman, B. Falsafi, and A. Ailamaki. Reactive NUCA: near-optimal block placement and replication in distributed caches. In *Proc. ISCA-36*, 2009.
[19] A. Hilton, N. Eswaran, and A. Roth. FIESTA: A sample-balanced multi-program workload methodology. In *MoBS*, 2009.
[20] R. Iyer, L. Zhao, F. Guo, et al. QoS policies and architecture for cache/memory in CMP platforms. In *Proc. SIGMETRICS*, 2007.
[21] M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver. A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC. In *Proc. DAC-49*, 2012.
[22] N. Jiang, D. Becker, G. Michelogiannakis, and W. Dally. Network congestion avoidance through speculative reservation. In *Proc. HPCA-18*, 2012.
[23] R. Johnson, I. Pandis, N. Hardavellas, et al. Shore-MT: A scalable storage manager for the multicore era. In *Proc. EDBT-12*, 2009.
[24] R. Kapoor, G. Porter, M. Tewari, et al. Chronos: predictable low latency for data center applications. In *Proc. SoCC-3*, 2012.
[25] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proc. MICRO-43*, 2010.
[26] P. Koehn, H. Hoang, A. Birch, et al. Moses: Open source toolkit for statistical machine translation. In *Proc. ACL-45*, 2007.
[27] N. Kurd, S. Bhamidipati, C. Mozak, et al. Westmere: A family of 32nm IA processors. In *Proc. ISSCC*, 2010.
[28] B. Lesage, I. Puaut, and A. Seznec. PRETI: Partitioned REal-TIme shared cache for mixed-criticality real-time systems. In *Proc. ICRTNS-20*, 2012.
[29] B. Li, L. Zhao, R. Iyer, et al. CoQoS: Coordinating QoS-aware shared resources in NoC-based SoCs. *Journal of Parallel and Distributed Computing*, 71(5), 2011.
[30] X. Lin and R. Balasubramonian. Refining the utility metric for utility-based cache partitioning. In *Proc. WDDD*, 2011.
[31] R. Liu, K. Klues, S. Bird, et al. Tessellation: Space-time partitioning in a manycore client OS. In *Proc. HotPar-1*, 2009.
[32] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. EuroSys-7*, 2012.
[33] J. Mars, L. Tang, R. Hundt, et al. Bubble-Up: Increasing Utilization in Modern Warehouse Scale Computers via Sensible Co-locations. In *Proc. MICRO-44*, 2011.
[34] D. Meisner and T. F. Wenisch. Stochastic queuing simulation for data center workloads. *EXERT*, 2010.
[35] D. Meisner, B. Gold, and T. Wenisch. PowerNap: Eliminating server idle power. *Proc. ASPLOS-14*, 2009.
[36] J. Mogul and K. Ramakrishnan. Eliminating receive livelock in an interrupt-driven kernel. In *Proc. USENIX ATC*, 1996.
[37] M. Moreto, F. J. Cazorla, A. Ramirez, et al. FlexDCP: A QoS framework for CMP architectures. *SIGOPS Operating Systems Review*, 43(2), 2009.
[38] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith. Fair queuing memory systems. In *Proc. MICRO-39*, 2006.
[39] K. J. Nesbit, J. Laudon, and J. E. Smith. Virtual private caches. In *Proc. ISCA-34*, 2007.
[40] J. Ousterhout, P. Agrawal, D. Erickson, et al. The case for RAMClouds: scalable high-performance storage entirely in DRAM. *SIGOPS Operating Systems Review*, 43(4), 2010.
[41] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proc. DATE*, 2007.
[42] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proc. MICRO-39*, 2006.
[43] V. Reddi, B. Lee, T. Chilimbi, and K. Vaid. Web search using mobile cores: quantifying and mitigating the price of efficiency. In *Proc. ISCA-37*, 2010.
[44] D. Sanchez and C. Kozyrakis. The ZCache: Decoupling Ways and Associativity. In *Proc. MICRO-43*, 2010.
[45] D. Sanchez and C. Kozyrakis. Vantage: Scalable and Efficient Fine-Grain Cache Partitioning. In *Proc. ISCA-38*, 2011.
[46] D. Sanchez and C. Kozyrakis. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *Proc. ISCA-40*, 2013.
[47] M. Schoeberl. Time-predictable computer architecture. *EURASIP Journal on Embedded Systems*, 2009.
[48] A. Seznec. A case for two-way skewed-associative caches. In *Proc. ISCA-20*, 1993.
[49] A. Sharifi, S. Srikantaiah, A. Mishra, et al. METE: meeting end-to-end QoS in multicores through system-wide resource management. In *Proc. SIGMETRICS*, 2011.
[50] J. Shin, K. Tam, D. Huang, et al. A 40nm 16-core 128-thread CMT SPARC SoC processor. In *ISSCC*, 2010.
[51] A. Snavely and D. M. Tullsen. Symbiotic jobscheduling for a simultaneous multithreading processor. In *Proc. ASPLOS-8*, 2000.
[52] S. Srikantaiah, M. Kandemir, and Q. Wang. SHARP control: Controlled shared cache management in chip multiprocessors. In *MICRO-42*, 2009.
[53] W. D. Strecker. Transient behavior of cache memories. *ACM Transactions on Computer Systems*, 1(4), 1983.
[54] L. Tang, J. Mars, W. Wang, et al. ReQoS: Reactive Static/Dynamic Compilation for QoS in Warehouse Scale Computers. In *Proc. ASPLOS-18*, 2013.
[55] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. In *Proc. ISMM*, 2011.
[56] Tilera. TILE-Gx 3000 Series Overview. Technical report, 2011.
[57] X. Vera, B. Lisper, and J. Xue. Data caches in multitasking hard real-time systems. In *Proc. RTSS-24*, 2003.
[58] D. Wendel, R. Kalla, R. Cargoni, et al. The implementation of POWER7: A highly parallel and scalable multi-core high-end server processor. In *ISSCC*, 2010.
[59] Y. Xie and G. H. Loh. PIPP: promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proc. ISCA-36*, 2009.
[60] H. Yang, A. Breslow, J. Mars, and L. Tang. Bubble-Flux: Precise Online QoS Management for Increased Utilization in Warehouse Scale Computers. In *Proc. ISCA-40*, 2013.
[61] X. Zhang, S. Dwarkadas, and K. Shen. Hardware execution throttling for multi-core resource management. In *Proc. of USENIX ATC*, 2009.