Ubik: efficient cache sharing with strict qos for latency-critical workloads

The MIT Faculty has made this article openly available. Please share how this access benefits you. Your story matters.

| As Published | http://dx.doi.org/10.1145/2541940.2541944 |
| Publisher | Association for Computing Machinery (ACM) |
| Version | Author's final manuscript |
| Accessed | Sun Jan 06 04:57:18 EST 2019 |
| Citable Link | http://hdl.handle.net/1721.1/90846 |
| Terms of Use | Creative Commons Attribution-Noncommercial-Share Alike |
| Detailed Terms | http://creativecommons.org/licenses/by-nc-sa/4.0/ |
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 $2.3 \times$, 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

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 $99^{th}$ 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 re-configuring 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 ≈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 parti-
We have selected five latency-critical workloads, with a focus on understanding these differences.

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

<table>
<thead>
<tr>
<th>Workload</th>
<th>Configuration</th>
<th>Requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>xapian</td>
<td>English Wikipedia, zipfian query popularity</td>
<td>6000</td>
</tr>
<tr>
<td>maastree</td>
<td>mcsb-a (50% GETs, 50% PUTs), 1.1GB table</td>
<td>9000</td>
</tr>
<tr>
<td>moses</td>
<td>opensubtitles.org corpora, phrase-based mode</td>
<td>900</td>
</tr>
<tr>
<td>shore</td>
<td>TPC-C, 10 warehouses</td>
<td>7500</td>
</tr>
<tr>
<td>specjbb</td>
<td>1 warehouse</td>
<td>37500</td>
</tr>
</tbody>
</table>

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.
- **maastree**, 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.

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 95\textsuperscript{th} pct service time in real system vs 0.9 ms simulated), or on an SSD (1.4 ms 95\textsuperscript{th} pct service time in real). **shore-mt** becomes I/O-bound on a conventional HDD (86 ms 95\textsuperscript{th} 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\textmu s 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., 95\textsuperscript{th} 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 99\textsuperscript{th} percentile latencies are commonly used [11], we use 95\textsuperscript{th} tail metrics, as gathering statistically significant 99\textsuperscript{th} tail metrics would require much longer runs. Our analysis is equally applicable to higher-percentile metrics.

**Statistical significance:** Despite using 95\textsuperscript{th} instead of 99\textsuperscript{th} 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\textsuperscript{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 ±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 (ρ = λ/μ, which matches processor utilization). We make three observations that have significant implications for adaptive systems:

**Observation 1:** Tail ≠ 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
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-nt, 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 \times 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.

### 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
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 fix a size of 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 use 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 95$^{th}$ 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 size.2

Under these conditions, transients are easy to predict. Specifically, assume that we upsized 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 ($T_{\text{transient}}$), 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 $T_{\text{access}} = c + p_s \cdot M$, 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 $T_{\text{access}} = c + 0.1 \cdot 100 = 1000/(5 \cdot 1.5) = 133$ 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 $T_{\text{miss,s}} = T_{\text{access,s}}/p_s = c/p_s + M$. 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.
When the app goes idle, Ubik downsizes its partition to its workloads that are mildly sensitive to partition size. \( N \) equals the upper bound on lost cycles at the transient. We limit the number of cycles gained by being at a size, \( s \), to add these intervals up:

\[
T_{\text{transient}} = \sum_{s=1}^{s_2-1} T_{\text{miss}, s} = \sum_{s=1}^{s_2-1} \frac{c}{p_2} + M
\]

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:

\[
T_{\text{transient}} \leq \sum_{s=1}^{s_2-1} \frac{c}{p_2} + M = (s_2 - s_1) \left( \frac{c}{p_2} + M \right)
\]

For the example above, if a transient starts at \( s_1 = 1 \) MB and ends at \( s_2 = 2 \) MB (\( s_2 - s_1 = 16384 \) 64-byte cache lines), \( p_1 = 0.2 \), and \( p_2 = 0.1 \), then the transient will take at most \( 16384 \times (123/0.1 + 100) = 21.8 \) million cycles.

During this transient, we incur \( \sum_{s=1}^{s_2-1} (1 - p_2/p_s) \) 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:

\[
L = M \sum_{s=1}^{s_2-1} (1 - \frac{p_2}{p_s}) \leq M (s_2 - s_1) \left( 1 - \frac{p_2}{p_1} \right)
\]

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 \( 100 \times 16384 \times (1 - 0.5) = 819 \) 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 highly 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, \( s_{\text{active}} \). In strict Ubik, \( s_{\text{active}} \) is the desired target size. When the app goes idle, Ubik downsizes its partition to its idle size, \( s_{\text{idle}} \). When the app becomes active, to make up for the lost performance, Ubik upsizes its partition to the boost size, \( s_{\text{boost}} > s_{\text{active}} \). Once the application has recovered the cycles it lost in the transient, Ubik downsizes it to \( s_{\text{active}} \) until the app goes idle again. Figure 6 shows this behavior.

Ubik periodically computes the best \( s_{\text{idle}} \) and \( s_{\text{boost}} \) for each partition. Figure 7 illustrates this process. Ubik evaluates \( N \) options for the idle size: \( s_{\text{idle}} = s_{\text{active}}, s_{\text{idle}} = s_{\text{active}}(N-1)/N, \ldots, s_{\text{idle}} = 0 \) (we set \( N = 16 \) in our experiments; Figure 7 shows \( N = 4 \)). For each option, Ubik computes the \( s_{\text{boost}} \) needed so that, by the deadline, the number of cycles gained by being at \( s_{\text{boost}} \) instead of at \( s_{\text{active}} \) equals the upper bound on lost cycles at the transient. We limit \( s_{\text{boost}} \) to \( s_{\text{boost}} \leq s_{\text{active}} \) to \( s_{\text{boost}} \rightarrow s_{\text{active}} \) total lines/latency-critical apps. This way, latency-critical apps never interfere with each other (even when all are boosted). The lower \( s_{\text{idle}} \), the higher \( s_{\text{boost}} \). Ubik stops evaluating options when either the transient is too long, or the needed \( s_{\text{boost}} \) 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 \( s_{\text{active}} - s_{\text{idle}} \) more lines available when the app is idle, and the cost is the extra misses they incur by having \( s_{\text{boost}} - s_{\text{active}} \) fewer lines for deadline cycles. These are computed using the batch apps’ miss curves. For example, in Figure 7 Ubik chooses option 3 (\( s_{\text{idle}} = s_{\text{active}}/2 \)), 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 \( s_{\text{boost}} \) to \( s_{\text{active}} \) would improve the latency-critical application’s performance unnecessarily while hurting batch throughput.

Fortunately, we can leverage the UMOM to accurately determine when an application has made up for its lost performance. UMOM tags are not flushed when the app goes idle, so UMOMs 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 UMOM events to track this; when the count exceeds the current number of misses (plus a guard to account for the small UMOM sampling error [42]), the application has made up for its losses. This triggers an interrupt, and the Ubik runtime de-boosts 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 re-partitions 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,
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 (every 50 ms), it incurs minimal overheads. Latency-critical stores what partitions to give space to or take space from apps call into the Ubik runtime when they become idle and time also arms the accurate de-boosting circuit, which triggers an interrupt when the cost of the transient has been recovered. Ubik resizes the latency-critical partition (to fast (hundreds of cycles), imposing negligible overheads. 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 resizing 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 UMOSNs: 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, most rarely 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 constrained 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 \((1 + \text{miss slack})\), 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
We simulate mixes of latency-critical and workload mixes:

work. The memory latency matches a Xeon W3670 running workload, with each instance serving a different set of requests. (which provide perfect isolation) as a baseline, and evaluate beginning of its configurations: with the 20% request rate to simulate a low-load case, and the 60% rate to simulate a high-load case. We use the five apps. Each mix has three instances of the same latency-critical app alone with a 2 MB LLC, and find the request rates that test each case with a wide and representative set of batch loads; TSO.

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 UMOS (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, nff, 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, \( \left( \sum I_{PC_i}/I_{PC_i,alone on 2MB LLC} \right)/N_{apps} \) [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 \( 2 \times \). 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 \( 2.2 \times \). These severe deadline violations make these policies unsuitable for

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

Table 2: Configuration of the simulated 6-core CMP.
With these best-effort schemes. As we will see later, which Ubik and StaticLC maintaining a tight upper bound on tail associativity \[ \text{of a statically sized partition. Because a Vantage partition has very high associativity} \] because a baseline being private caches instead of a statically sized partition. Because a Vantage partition has very high associativity [45], these differences are overall positive.

Overall, weighted speedups are modest because many batch apps, and UCP, and outperforms LRU, 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. Noises 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 noises a large fraction of the cache, and with LRU, noises naturally grabs the space from batch apps. Both schemes achieve tail latency for batch apps, and some are slightly worse. 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 speeds 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 speeds, 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 speeds competitive with OnOff and UCP, and outperforms LRU, while still meeting deadlines. Overall, weighted speeds are modest because many batch apps in our mixes are not memory-intensive. As Figure 9 shows, about 23% of the mixes enjoy weighted speeds 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 speeds 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. maestree 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. Noises 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 noises a large fraction of the cache, and with LRU, noises naturally grabs the space from batch apps. Both schemes achieve tail latency...

<table>
<thead>
<tr>
<th>Workload Mix</th>
<th>LRU</th>
<th>UCP</th>
<th>OnOff</th>
<th>StaticLC</th>
<th>Ubik</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low load</td>
<td>13.1%</td>
<td>18.3%</td>
<td>18.3%</td>
<td>8.9%</td>
<td>17.1%</td>
</tr>
<tr>
<td>High load</td>
<td>9.8%</td>
<td>14.7%</td>
<td>14.5%</td>
<td>8.3%</td>
<td>14.8%</td>
</tr>
</tbody>
</table>

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