NyxCache: Flexible and Efficient Multi-tenant Persistent Memory Caching

Kan Wu, Kaiwei Tu, Yuvraj Patel, Rathijit Sen[†], Kwanghyun Park[†],
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
University of Wisconsin–Madison [†]Microsoft

Abstract. We present NyxCache (Nyx), an access regulation framework for multi-tenant persistent memory (PM) caching that supports light-weight access regulation, per-cache resource usage estimation and inter-cache interference analysis. With these mechanisms and existing admission control and capacity allocation logic, we build important sharing policies such as resource-limiting, QoS-awareness, fair slowdown, and proportional sharing: Nyx resource-limiting can accurately limit PM usage of each cache, providing up to 5× better performance isolation than a bandwidth-limiting method. Nyx QoS can provide QoS guarantees to latency-critical caches while providing higher throughput (up to 6× vs. previous DRAM-based approaches) to best-effort caches that are not interfering. Finally, we show that Nyx is useful for realistic workloads, isolating write spikes, and ensuring that important caches are not slowed down by increased best-effort traffic.

1 Introduction

Memory-based look-aside key-value caches (e.g., memcached [14]) are a critical component of many systems and applications [3, 5, 23, 74]. To improve utilization and simplify management, multiple cache instances are often consolidated onto a single multi-tenant server. For example, Facebook [54] and Twitter [74] each maintain hundreds of dedicated cache servers that host thousands of cache instances. However, multi-tenant servers have the added challenge of ensuring that each client cache meets its performance goals; a range of production and research in-memory multi-tenant caches currently provide different sharing policies, such as enforcing a limit on the used memory capacity and bandwidth [7], guaranteeing a level of quality-of-service (QoS) [18], and allocating resources proportionately [60].

Persistent memory (PM), such as that provided by Intel's Optane DC PMM [10], is emerging as an appealing building block for these caches, due to PM's large capacity, low cost per byte, and comparable performance to DRAM. However, PM performance differs from DRAM and Flash in a number of ways that reduce the effectiveness of current multi-tenant caches for other devices [34, 62]. In particular, unlike DRAM, Optane DC PMM exhibits highly asymmetric read vs. write performance (for a single DC PMM, max read bandwidth is 6.6GB/s whereas max write bandwidth is 2.3GB/s) [45], severe and unfair interference between reads and writes (writing at 1GB/s can cause the same throughput and P99 latency slowdown to a co-running read workload as reading at 8GB/s) [55], and especially efficient access for multiples of 256B [73].

Unfortunately, existing multi-tenant DRAM and storage caching techniques do not readily translate to PM. Some approaches focus exclusively on capacity allocation across clients [34, 60, 62]; capacity allocation is necessary but not sufficient for PM sharing because the rate of requests to PM must also be regulated. Host-level request regulation has been explored extensively for Flash devices using block-layer I/O scheduling [58, 61], but these software overheads are prohibitive given 100ns PM accesses [24]. Device-level request scheduling assumes special hardware that PM lacks [53, 65, 78, 79]. Finally, coarse-grain request throttling underpins the vast majority of DRAM bandwidth allocation techniques; however, these approaches assume both hardware counters and performance characteristics that do not hold for PM (e.g., bandwidth is an accurate estimate of utilization).

In this paper, we introduce NyxCache (Nyx), a standalone lightweight and flexible PM access regulation framework for multi-tenant key-value caches that is optimized for today's PM without special hardware support. Given a PM server and a sharing policy (e.g., QoS), cache instances are admitted and assigned space using existing load admission [36, 37, 52] and capacity allocation [34, 60, 62] techniques. At runtime, Nyx monitors information (e.g., PM resource utilization) of caches, regulates the rate at which each cache is allowed to access PM, and thus enforces the sharing policy's performance goals. Nyx works with any in-memory key-value store that adheres to the memcached interface [14]; the current implementation includes a PM-optimized version of Twitter's Pelikan [17] that can improve single-cache performance by more than 50% for get-heavy workloads and 3× for write-heavy workloads. Nyx's central contribution is a set of software mechanisms designed for PM to extract the information required to flexibly enforce popular sharing policies.

Nyx provides new mechanisms to efficiently i) regulate PM accesses, ii) obtain a client's PM resource usage, iii) analyze inter-client interferences, and has two particularly useful and novel mechanisms for PM. First, Nyx efficiently estimates not only the total PM DIMM utilization (building on pioneering work in this space [55]), but also the PM utilization caused by each cache instance, as is needed for sharing policies; estimating PM utilization is challenging because the number of transferred bytes is not an accurate proxy of PM utilization, unlike on DRAM. Second, Nyx can determine which cache instance most interferes with another cache instance; in PM-based systems, these interactions are difficult to identify because a harmed client may be impacted more by a low-bandwidth client than a high-bandwidth client, unlike DRAM. Both of these mechanisms accurately account for the

	Resource Limit	Quality of Service	Fair Slowdown	Proportional Resource Allocation
Request Regulation	✓	✓	✓	✓
Resource Usage	✓			✓
Interference		✓		*
Application Slowdown			✓	✓

Table 1: **Control and Information Needed.** ✓*indicates control or information is required by the policy. * indicates optional.*

CPU cache prefetching that is essential for high performance on PM. These new mechanisms enable Nyx to easily and efficiently support sharing policies such as resource limiting, QoS, fair slowdown, and proportional sharing.

The sharing policies provided by Nyx are powerful. Nyx can accurately limit the PM utilization of each cache (similar to Google Cloud's memcache [7]), whereas an approach that measures only bandwidth cannot. Nyx can provide QoS guarantees to latency-critical caches while providing higher throughput (up to 6×) to best-effort caches that are not interfering. Nyx can provide proportional resource allocation while redistributing idle PM utilization to clients that will not inadvertently slowdown others. Finally, as shown for real large-scale cache traces from Twitter, Nyx can isolate clients from write spikes and ensure that important caches are not slowed down by increased best-effort traffic.

In the rest of this paper, we evaluate previous multi-tenant caches and their limits for PM (§2); discuss the Nyx design (§3); evaluate overheads of Nyx's mechanisms and the effectiveness of its policies (§4); discuss potential extensions (§5); compare to related work (§6); and conclude (§7).

2 Motivation and Challenges

We provide background on the sharing policies provided by many in-memory multi-tenant key-value caches and the mechanisms needed to implement those policies. We explain why previous approach for providing control and information on DRAM or block I/O do not work well on PM.

2.1 Sharing Policies for Multi-Tenant Caches

In-memory key-value caches such as memcached [14], Redis [66], and Pelikan [17] are an essential part of web infrastructure for many real-time and batch applications [3, 74]. Before accessing data from slow backend-storage or compute nodes, applications first check an in-memory cache server. In production environments, cache servers are usually multi-tenant: many cache instances are consolidated on a single server to improve utilization and simplify management and scaling [54]. In a multi-tenant cache, requests are routed to the cache instance of the corresponding tenant. For example, large companies such as Facebook [54] and Twitter [74] maintain hundreds of large-memory dedicated servers that host thousands of cache instances. Smaller companies use caching-as-a-service providers such as ElastiCache [1], Redis [20] and Memcachier [16]. In this paper, we focus on managing an individual multi-tenant cache server.

Giving competing clients, enforcing performance and sharing goals is critical in multi-tenant caching. Different industrial and research multi-tenant systems have provided different objectives; we focus on the following four.

Resource Limiting: A common objective when clients pay for resources is to guarantee that each client cannot exceed some amount of usage such as bandwidth, ops/sec, or number of resources [2, 7]. For example, Google Cloud memcache limits operations according to a pricing tier, such as "Up to 10k reads or 5k writes (exclusive) per sec per GB" [7]. Multiple resources can be limited simultaneously, e.g., Amazon ElastiCache [2] charges for both memory and vCPUs.

There are two requirements for a multi-tenant cache to enforce per-client resource limits. First, the system must accurately determine the amount of resource each client is using; we refer to this as *resource usage estimation*. Second, the system must reschedule or throttle requests of each client if they exceed this limit, which we call *request regulation*. Below (§2.2), we describe how previous multi-tenant caches have provided request regulation and resource usage estimation, and why these previous approaches are not sufficient for PM.

Quality-of-Service: A multi-tenant system may ensure that each client's performance goals (throughput, latency, or tail latency) are met regardless of other co-located clients, as in Twitter [18] and Microsoft [62]. This objective is useful for latency-critical clients that must meet service-level objectives (SLOs). For example, production caches at Twitter provide a p999 latency of <5 milliseconds [18].

Providing QoS requires knowledge of whether each client is meeting its goals at run-time. When the system observes that one client is not meeting its performance guarantee, interfering clients are identified and limited [31, 39, 51] (e.g., with *request regulation*). Identifying the client causing the most harm is usually straightforward and based on simple bandwidth [39] for DRAM-based caches, but not for PM. A new technique involving *interference estimation* is required on PM to determine how the workloads compose.

In addition to run-time support, guaranteeing QoS requires admission control and space allocation. Admission control must be performed on newly arriving clients to ensure that the system has sufficient resources and that the new client will not interfere with existing clients [36, 37, 52]. Space allocation across cache instances must be performed to provide a specified hit ratio for each client to ensure each can meet its goals. Previous research has focused on this challenge. For example, Microsoft [62] allocates space to meet QoS bandwidth targets, and Robinhood [29] to minimize tail latency. Admission control and space allocation are mostly orthogonal to the new challenges introduced by PM and are not our focus.

Fair Slowdown: Multi-tenant systems in more cooperative environments may ensure that all clients are slowed down by the same amount. Formally, these approaches minimize the ratio of the maximum slowdown to the minimum slow-

down [38, 63]. In web cache settings, application requests may fan out, in which case the cache access with the longest latency determines overall latency [29, 54]; thus, balancing slowdown benefits overall request latency.

Enforcing fair slowdown requires knowledge of each cache's slowdown at runtime. The system must monitor each cache's current performance when sharing the server with others and know its performance if run alone. A technique for *slowdown estimation* is required. Furthermore, to equalize slowdowns of different caches, caches with small slowdown should be further limited and caches with larger slowdowns should be less limited (e.g., with *request regulation*).

Proportional Resource Allocation: Finally, a multi-tenant system may incent clients to share resources by guaranteeing that each of N clients performs within $1/N$-th of its stand-alone performance. This guarantee can be generalized to give each client a different proportional share. Idle resources may be redistributed across clients, such that some obtain more than their guarantee. For example, FairRide [60] ensures proportional cache space allocation.

To guarantee proportional allocation, a multi-tenant cache must meet three requirements. The system must perform *request regulation* and *resource usage estimation* to guarantee that each client does not consume more than its allocation. When assigning idle resources to clients, the system must validate that the additional resource usage does not interfere with others; therefore, the system must track each client's slowdown (i.e., with *slowdown estimation*) and stop idle resource re-allocation before it severely impacts some clients.

In summary, for a multi-tenant cache to provide the above policies, it must control resource usage of each cache instance and obtain information about resources and application performance. Table 1 summarizes the needed control and information for each policy.

2.2 Challenges of PM Cache Sharing

Persistent memory is an appealing building block for key-value caches. After presenting PM background, we describe the challenges of using PM for multi-tenant caching.

2.2.1 Persistent Memory Characteristics

PM is becoming a reality in products and research prototypes. For example, Intel Optane DC PMM [10] is a popularly available device; there are also research prototypes [30,49,70]. In this paper, we use PM to refer to Optane DC PMM. PM performance is similar to DRAM but can deliver extremely large capacity at low cost [10, 11]. PM is significantly faster than NAND Flash and is byte-addressable. PM is directly connected to the memory bus and, when configured in App Direct Mode, can be accessed using loads and stores. Different CPU caching options exist for PM access: loads and stores with CPU caching and prefetching; loads and stores with prefetching disabled (for both PM and DRAM); non-temporal (NT) operations that bypass the CPU cache entirely [73].

Table 2 summarizes the bandwidth and latency of Optane DC PMM for a workload relevant to key-value caches: random 256B and 4KB loads and stores. As shown, for loads, regular loads perform best: CPU cache prefetching is essential for hiding PM latency and increasing throughput. For stores on a random workload, NT-stores that bypass the CPU cache have much better performance. Thus, we use in-PM key-value caches optimized to use regular loads and NT-stores.

	Metric	Load	No-Prefetch	NT-Load	Store	Store+clwb	NT-Store
256B	GB/s	1.59	1.53	0.29	1.12	0.52	3.73
	us	0.49	0.52	0.84	0.38	0.47	0.08
4KB	GB/s	4.08	2.92	2.24	1.03	1.50	3.44
	us	1.22	1.69	1.84	4.14	2.71	1.22

Table 2: **PM Load/Store Performance.** *This table summarizes the throughput/latency of single thread random 256B and 4KB load/store operations (on 2× DC PMMs). No-Prefetch: the CPU's prefetching is turned off (for DRAM/PM); NT: non-temporal operations that bypass the CPU cache.*

PM has unique characteristics that impact multi-tenant caching. For instance, as previously identified, PM exhibits asymmetric read vs. write performance [45], especially efficient access for specific sizes (e.g., 256B) [73], and severe and unfair interference across reads and writes [55]. As we will describe, these characteristics deeply impact the ability to perform request regulation and to estimate resource usage, interference, and application slowdown.

2.2.2 Request Regulation

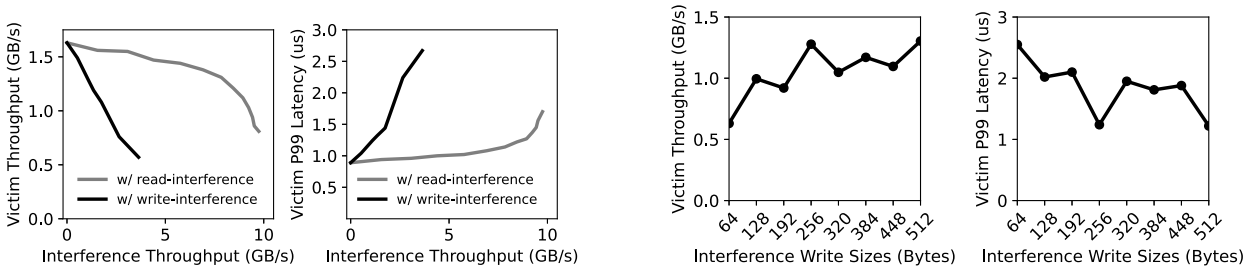
Previous approaches for request regulation have been designed for both DRAM and for block I/O. However, none of these approaches are suitable for PM.

Existing techniques for regulating memory requests have adjusted the number of cores dedicated to an application [39], used clock modulation (DVFS) [57], and Intel Memory Bandwidth Allocation (MBA) [9]. In multi-tenant caching, reducing the number of cores is not suitable because a cache instance is often allotted only a single core [2]. Intel MBA manages last-level cache (LLC) misses from each core to limit memory traffic, but does not distinguish between misses to PM and DRAM [8] and so cannot restrict PM accesses without also slowing down DRAM. Furthermore, Intel MBA does not have access to accurate information about resource usage, interference, and application slowdown, as we will discuss. Likewise, adjusting CPU frequency has an effect on all instructions; Oh et al. [55] demonstrated the ineffectiveness of CPU frequency scaling on regulating PM traffic.

I/O requests have been regulated via software with block-layer I/O scheduling [12], which is not suitable for PM for two reasons. First, the block abstraction would add significant read/write amplification for byte-addressable PM. Second, scheduling requests with merging, reordering, and other synchronization would add unacceptable overhead to otherwise low-latency PM accesses [24].

2.2.3 Resource Usage Estimation

Previous techniques for estimating the memory or I/O usage of clients do not work well for PM. We describe the problems with previous software approaches for tracking I/O



(a) Read vs. Write Interferences

(b) Interferences Related to Access Sizes

Figure 1: **PM Load Performance with Various Interferences.** *We place a victim workload (single thread 256B loads) with various interferences. (a) shows the victim throughput and tail latency when colocated with varying amounts of read and write interferences. (b) shows the victim performance when colocated with 1GB/s store traffic of varying access sizes (range from 64B to 512B with step of 64B).*

usage and with hardware approaches for DRAM.

As discussed above, CPU cache prefetching is required for PM to deliver high bandwidth and low latency. However, when estimating block I/O traffic in software [4, 35, 76], extra PM accesses caused by prefetching are not observed. Running an experiment with 1KB random loads, we found that software-level tracking accounted for only 60% of actual memory traffic, leading to inaccurate resource-usage estimation.

Accounting on DRAM uses hardware counters to track L3 cache line misses to the memory controller per core. While hardware counters accurately measure prefetching, they do not account for the difference between cache line size and PM access granularity, which is needed for PM accounting. Because PM has a 256B minimum access granularity, a 64B load (a single L3 cache line) utilizes the same amount of PM resources as a 256B load (four L3 cache lines). Thus, four cache line accesses can result one to four PMEM accesses. Previous systems for resource estimation have often used bandwidth consumption as a proxy for resource usage [39, 51, 77], but this is not appropriate for PM where operation cost is affected by access size and is different for reads versus writes.

Unfortunately, current hardware counters in PM are also not sufficient; existing PM counters are at the DIMM media-level and do not track per-client or per-core usage [13, 55].

2.2.4 Interference Estimation

In memory-based approaches, interference caused by a particular client was assumed to be related to memory bandwidth. For example, Caladan [39] identifies the client with the highest number of LLC misses, which corresponds directly to the client with the highest memory bandwidth. This simplification does not work for PM, as PM interference depends on both volume and pattern of traffic.

Specifically, on PM, write-intensive clients generate greater interference than read-intensive clients with the same bandwidth, as shown in Figure 1.a. For example, on a read-intensive client, a competing 1GB/s write causes the same throughput and tail latency interference as a competing 8GB/s read. As shown in Figure 1.b, smaller accesses (64B) can cause more interference than larger accesses (256B). Since PM has a minimum granularity of 256B, a 64B access is amplified into 256B on the device; thus, at the same bandwidth, 64B accesses generate significantly more interference than 256B accesses. In short, the bandwidth of a competing client is not a good estimation of interference in PM, unlike DRAM.

2.2.5 Application Slowdown Estimation

Numerous efforts have estimated slowdown for DRAM and Flash-based systems; however, all require specialized device support. For example, FST [38] requires in-DRAM bank conflict counters that are updated with each memory access; MISE [64] and ASM [63] require the DRAM controller to assign priorities to application requests. FLIN [65] changes the Flash controller to track and rearrange each flash transaction. Although application slowdown is not inherently different on PM than DRAM or I/O, previous approaches require special hardware which is not available on PM.

Summary: Multi-tenant PM caching demands new methods for regulating PM accesses and extracting PM resource usage, interference information, and application slowdown.

3 NyxCache Design

Given that existing multi-tenant cache servers cannot handle PM, we introduce NyxCache (Nyx). Nyx provides mechanisms for control (e.g., request throttling) and information estimation on PM (e.g., resource usage, interference, and application slowdown), and supports a range of sharing policies (e.g., resource limiting, quality-of-service, fair slowdown, and proportional resource usage). We describe the overall architecture of Nyx, present our design goals, describe how Nyx provides these mechanisms and policies.

3.1 Architecture

As shown in Figure 2, Nyx provides a multi-tenant in-PM caching framework. Each PM server running Nyx may contain any number of cache instances (e.g., memcached, Pelikan, Redis). Thousands of users may send requests (e.g., set/get) to their associated cache instance. When cache space is exhausted, a cache instance can use any eviction strategy (e.g., FIFO, LRU, and LFU). As in other look-aside caches, users explicitly write desired data into the cache; Nyx does not fetch data from remote storage on a cache miss.

Nyx can be configured with different sharing policies and parameters (e.g., a resource limit, latency target, or propor-

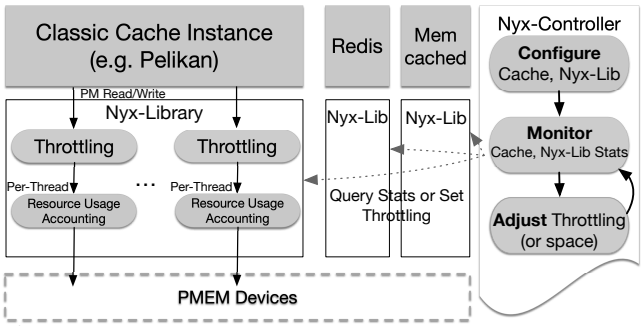


Figure 2: **NyxCache Architecture.** *Nyx implements throttling and resource usage accounting for each cache instance, and enforces sharing policies across cache instances. Nyx contains two major components: 1) a Nyx Library for each instance, and 2) a centralized Nyx Controller.*

tional weight). Administrators can implement new policies using the control and information mechanisms provided by Nyx. At runtime, Nyx enforces the desired sharing policy. Based on information Nyx acquires about per-instance resource usage and performance, the Nyx controller dynamically adjusts the throttling and space allocated to instances.

Nyx has two requirements for cache instances. First, each cache instance must report application-level performance metrics such as throughput and tail latency; most systems have this capability or can be extended [15]. Second, the instances must be integrated with a trusted Nyx-library. When a cache instance reads/writes from/to PM, it must use Nyx library APIs (e.g., read(dest, src), write(dest, src)). For each PM access, the Nyx library throttles access, tracks PM usage, and performs the actual access. The library uses a separate thread to communicate with the Nyx controller. The controller interacts with the library to query statistics and to set configuration, space, and throttling values. Nyx leverages techniques from previous multi-tenant in-memory caches for basic sharing functionality such as admission control and space allocation. As of now, Nyx only manages cache instances on a single NUMA node that share PM (and all PM accesses are local); multiple Nyx can be used to manage multiple NUMA nodes. We leave NUMA-aware management for future work.

3.2 Design goals

Nyx has the following goals. (i) **Lightweight:** Performance is critical for in-PM caching; thus the cost of adding control and acquiring information must be low relative to the cost of accessing PM. (ii) **Flexible Sharing Policies:** Different sharing policies may be required by administrators for different scenarios. Thus, Nyx can be configured with several policies based on a common set of simple mechanisms. (iii) **No Special Hardware:** Previous work has assumed smart resources (e.g. Flash, DRAM) that provide configurable control and information [53, 65, 78, 79]. Nyx handles current devices with existing hardware interfaces. (iv) **Minimal Assumptions:** Storage devices are continuously evolving, with new generations having new performance characteristics. Therefore, Nyx does not assume a particular performance model for all PM devices (e.g., the interference for different operations).

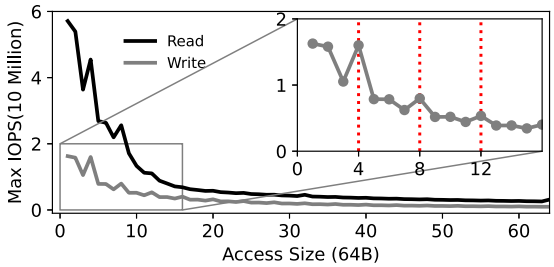


Figure 3: **MaxIOPS Profile.** *MaxIOPS for random reads and writes of different sizes on our 2× Intel Optane DC PMM system.*

3.3 Nyx Mechanisms

Nyx contains low-level mechanisms that enable higher-level sharing policies to be implemented easily. Since request regulation, estimation of resource usage, interference, and application slowdown are changed significantly by PM, we describe these Nyx mechanisms in detail. Access control and space allocation are largely independent of PM and not the focus of this paper; Nyx borrows these techniques from previous systems [29, 34, 52, 60, 62].

PM Access Regulation: To minimize the overhead of regulating requests to PM, Nyx adheres to the basic principle used by previous techniques for DRAM regulation: throttle requests in a coarse-grained manner without reordering or prioritizing. To mimic the behavior of Intel MBA, Nyx implements simple throttling by delaying PM accesses at user-level.

Our current implementation adds delays in units of 10ns with a simple computation-based busy loop. In some cases PM operations may need to be delayed indefinitely (e.g., when a resource limit is reached); in this case, PM operations are stalled until the Nyx controller sets the delay to a finite value.

Resource Usage Estimation: Nyx must determine how much PM resource each cache instance is using. As described in Section 2, for PM the number of transferred bytes is not a good estimate of resource usage; on PM, each operation type (e.g., read or write) and access pattern (e.g., request size) consumes a different amount of the resource and has a different maximum operations per second. Therefore, Nyx determines the utilization of PM as a function of the current IOPS of each operation type relative to the maximum IOPS for that operation type. For example, if the maximum IOPS of pattern A is $MaxIOPS_A$, then the cost of each operation of pattern A is $1/MaxIOPS_A$. If the maximum IOPS of pattern B is $1/N \times MaxIOPS_A$, then each B operation consumes N times more PM than an A operation and has N times the cost. The IOPS cost model accurately captures that writes are more expensive than reads, and the dependency on request size.

Nyx determines the MaxIOPS of each access pattern through profiling, performed once per PM server. The profiler measures IOPS for random read and write operations between 64B and 4KB (in steps of 64B). Because prefetching occurs during profiling, the measured MaxIOPS accurately represents the cost of both the operation itself and any wasted prefetching. Profiling concentrates on random accesses as

multi-tenant key-value caches are mostly random: first, because multiple tenants access PM simultaneously (in different address spaces), their requests are interleaved; second, keys tend to be mapped to arbitrary PM locations based on their time-to-live and size [14, 75]. The profiler stops at request sizes of 4KB which obtain the device's maximum bandwidth.

Figure 3 shows the profiled MaxIOPS for reads and writes as a function of request size. As shown, writes have lower IOPS and thus a higher cost per operation than reads. While larger requests generally have lower IOPS, there is a complex relationship with the minimal PM access size: for example, a 64B random store has a similar maximum IOPS as 256B, the minimum PM access size; accesses that are not aligned to 256B have lower MaxIOPS.

At runtime, Nyx tracks the PM usage of each cache instance. When a cache instance accesses PM, Nyx looks up the MaxIOPS for this operation and size, and increments a cost counter for this cache instance by $\frac{1}{MaxIOPS}$. To reduce synchronization overhead, these counters are maintained per-thread and only lazily combined when needed (e.g., for responding to a resource usage query from Nyx Controller).

While the CPU cache can theoretically introduce errors in PM cost estimation, these errors are negligible for Nyx. First, since CPU prefetching waste depends in part on spatial locality, the profiler mimics the random accesses of cache instances that have little sequentiality. Second, given a cache instance that uses NT-store (as in Nyx-Pelikan), the CPU cache has no effect on stores. Finally, although a PM load could be served in the CPU cache and never access PM, in multi-tenant caches few PM loads hit in the CPU cache: because each instance's working set is typically tens of GBs [28, 74] (and there are many instances), there is little temporal locality in CPU caches of tens of MBs. More intricate cost models for cache instances with spatial (e.g., scan) and temporal locality (e.g., bursty retries) are left for future work.

Interference Analysis: When multiple cache instances are co-located, Nyx determines which instance most impacts another. For example, when an efficient QoS implementation observes that an affected client W is not meeting its guarantee, it will iteratively slow down the one competing client that will produce the greatest benefit for W. In PM-based systems, unlike DRAM, these interactions are difficult to identify because an affected client may be impacted more by a low-bandwidth client than a high-bandwidth client. The amount of interference is due to complex scheduling within the PM device; as future generations of PM devices become available, which clients interfere with which others may change. Therefore, Nyx assumes no prior knowledge of these interactions.

Nyx determines which client is interfering the most with the affected client with a runtime micro-experiment. Given affected client W and several competing clients, Nyx iteratively throttles each competing client by X for some metric of interest while measuring the impact on client W. The throttled client that helps W attain the greatest performance improvement is identified as the client that interferes with W the most. The value of X is configurable, as is the metric (e.g., throughput, average latency, or tail latency). Nyx uses simple pruning techniques to throttle only the clients with the highest resource usage. Optimizations for reducing micro-experiment times (e.g., focus on different client subsets in different trials) are left for future work.

Algorithm 1: Resource Limit The gray area denotes unique functionality used to deal with PM issues

```
EpochLen: ticks in an epoch (e.g. 100), TickLen: (e.g. 10ms)
A.getResCounter(): query A's Nyx-Lib for resource usage
A.setThrottling(t): add t×10ns delay to each access of A
ResAssigned[1..N]: each cache's assigned resource per epoch
while true do
    # Step 1: Begin an epoch and set all cache throttling to 0
    foreach cache A do
        A.setThrottling(0)
        InitResCounter[A] = A.getResCounter()
    # Step 2: Monitor resource utilization and pause clients
      who have used up their allotted resources.
    while Epoch is not completed do
        SleepFor(TickLen)
        foreach cache A do
            ResUsed = A.getResCounter() - InitResCounter[A]
            if ResUsed > ResAssigned[A] then
                A.setThrottling(INFINITE) # Pause
```

SlowDown Estimation: Nyx determines the slowdown that each client experiences at runtime by calculating $\frac{T_{alone}}{T_{share}}$; T_{alone} is the client's performance (for some metric of interest) when it is running alone, and T_{share} is its current performance in the shared environment. As we assume no special hardware, Nyx uses an approach similar to previous work [47].

First, to learn T_{alone}, Nyx briefly pauses all other clients; T_{alone} is updated on a regular basis (e.g., 1s) or whenever a workload change is observed. Second, slowdown is periodically calculated using a runtime measurement of T_{share}. As we will show, at the cost of a small loss of bandwidth and increase in tail latency, this solution adequately approximates slowdown without hardware support. The impact of the pause can be reduced for workloads that do not change frequently.

3.4 Nyx Sharing Policies

Nyx implements four popular sharing policies. We describe how these policies leverage the mechanisms of Nyx for PM.

Resource Limit: Nyx can limit the amount of the PM resource used by each client in multi-tenant caching, isolating the performance of clients from one another. Our policy defines resource limits in terms of standard operations, similar to Google Cloud's memcache [7] (e.g., 1000 1KB random reads per second, or 1MB/s random reads).

As shown in Algorithm 1, Nyx provides resource limits for each client epoch by epoch, extending existing approaches [77]. Each epoch, Nyx monitors the resource utilization of each client; if a client reaches its limit for this epoch, its accesses to PM are delayed until the next epoch. When the epoch ends, the throttling value for each client is reset to zero.

Algorithm 2: QoS The gray area denotes functionality for PM. We omit code to rollback throttling when the action violates any LC task's target.

```
ExperimentStep: a cache's throughput expense pays for an
  interference analysis experiment. (e.g. 500MB/s)
while true do
    # Step 1: Monitor each client's SLO slack
    foreach cache A do
        slack[A] = (A.target - A.latency) / A.target
    S = cache with the smallest slack
    # Step 2: Protect clients violating SLO
    if slack[S] < 0 then
        if S is throttled then
            throttle down S
        else
            # Step 2.1: Pick candidates to throttle
            if there are BE caches then
                candidates = top 3 resource usage BE
            else
                candidates = top 3 res usage LC, slack > 0.2
                if all LCs have little slack then
                    candidates = LC with the most slack
            # Step 2.2: Find the most interfering client
            I = getLargestInterference(S, candidates)
            throttle up I
    else if slack[S] > 0.2 then
        # All caches have slack -> relax throttling
        throttle down every cache

Function getLargestInterference(S, Candidates):
    # Find the tenant who will most improve S at the same
      expense (throughput)
    If there is only one client in Candidates, return the client
    foreach C in Candidates do
        throttle up C by ExperimentStep
        track S latency change after the experiment
        restore all throttle to previous state
    return L who helps S get the largest improvement
```

The implementation allows the administrator to configure the *epoch* and *tick* length to trade-off the overhead of checking counters with reaction time.

Quality-of-Service: Nyx can ensure that latency-critical (LC) tenants meet a service-level-objective while maintaining high PM utilization for best-effort (BE) tenants on the same server. As in earlier work [36, 37], admission control prevents workloads with unachievable QoS targets and space-allocation provides the necessary hit ratio.

As shown in Algorithm 2, Nyx employs an approach similar to Parties [31] and Caladan [39]: for each LC client, the difference between the guaranteed and the current performance is tracked; when the guarantee is violated (i.e., negative slack), a competing tenant is throttled.

Nyx differs in how it identifies the client to be throttled. Caladan always throttles the BE tenant with the maximum bandwidth (LLC misses), whereas Nyx throttles the BE or LC cache that most improves the LC cache, for the same expense across competing tenants. The implementation allows the administrator to configure *ExperimentStep*, allowing a balance between aggressive throttling and faster convergence.

Fair Slow Down: Nyx can achieve fairness in terms of

Algorithm 3: Fair Slow Down

```
A.getSlowDown(): return A's current performance / T_alone
while true do
    if T_alone info is older than P sec then
        foreach cache A do
            refreshTalone(A)
    # Adjust throttling to equalize slowdowns
    foreach cache A do
        SlowDown[A] = A.getSlowDown()
    find cache L and S with the largest and smallest slowdowns
    unfairness = SlowDown[L] / SlowDown[S]
    if unfairness > UnfairnessThreshold then
        throttle down L and throttle up S
        FairIntervals = 0
    else
        # With fair slowdown, try to improve utilization
        FairIntervals ++
        if FairIntervals > FairIntervalThreshold then
            throttle down all caches

Function refreshTalone(A):
    A.setThrottling(0), and pause every other cache
    A.T_alone = measure A throughput
    restore throttle of all caches to previous state
```

equalized slowdown across caches. As in Algorithm 3 [38,63], Nyx minimizes (MaxSlowDown/MinSlowdown) by gradually increasing the throttling of the MinSlowDown cache and decreasing the throttling of the MaxSlowDown cache. The tuning process is terminated when the unfairness metric falls under an UnfairnessThreshold. The implementation periodically (every P seconds) refreshes the estimate of the stand-alone performance (T_{alone}) for each client. Administrators can customize P to balance between lower overhead and faster adjustments for dynamic workloads.

The policy can be generalized to guarantee weighted slowdowns and a hard limit on some cache's slowdown. For the hard limit, Nyx tracks the particular slowdown at runtime and throttles other caches when the hard limit is exceeded.

Proportional Resource Allocation: Nyx implements proportional sharing with actual proportional resource allocation (instead of simple bandwidth allocation) and with interference-aware idle resource redistribution. Nyx ensures that each cache achieves performance equal to or better than accessing PM alone for a given amount of time (time-sharing [67]). For example, if a cache has a weight of 2 out of 3, then it is guaranteed to obtain at least 2/3 of its stand-alone performance.

Nyx first allocates resources (not bandwidth) proportionally to each cache and enforces the resource limit during an epoch (Algorithm 4). We assume cache space has been allocated proportionately. Following an epoch, Nyx forecasts each tenant's desired amount of resources: a tenant that did not use all its given resource may donate idle resources, whereas a tenant that used all assigned resources may consume more (a simple linear model predicts desired resources [77]).

Nyx provides interference-aware resource donation (Option 2 in the Alg.). On PM, idle resource redistribution faces the difficulty that the donated resource may severely interfere with the original donor's performance. For example, as shown

Algorithm 4: Proportional Resource Allocation The slowdown refreshing code is omitted.

```
DonateStep: step to donate idle resources (e.g. 10%)
TotalResource = 1
while true do
    # Step 1: Enforce and track resource usage in an epoch
    Begin a New Epoch
    foreach cache A do
        Enforce A uses resource <= ResourceAssigned[A]
        if A depleted resources, record how long:
          TimeUseUp[A] (e.g. half of the epoch)
        if A left idle resources, record ResourceUsed[A]
    End of the Epoch
    # Step 2: Redistribute Idle resources
    foreach cache A do
        if A has idle resources then
            # Option 1: Donate all extra resources
            DesiredResource[A] = ResourceUsed[A]
            # Option 2: Interference-aware resource donation
            if A.getSlowdown() < TotalWeight / A.weight then
                # Donate a step when within slowdown limit
                DesiredResource[A] =
                Max(ResourceAssigned[A] * (1 - DonateStep),
                ResourceUsed[A])
            else
                # Revoke a step when under slowdown limit
                DesiredResource[A] =
                  Min(ResourceAssigned[A] * (1 +
                  DonateStep), TotalResource * A.weight/
                  TotalWeight)
        if A depleted resources: DesiredResource[A] =
        ResourceAssigned[A] / TimeUseUp[A]
    ResourceAssigned[1..N] = Allocate resources
      proportionally based on weight and desired resource
```

in Section 4.5, if a get-heavy cache A donates idle resources to a write-heavy cache, the new write traffic can dramatically harm A's performance. To prevent this interference, Nyx reallocates resources in increments, stopping when the donating cache's slowdown is near its lower bound; if the slowdown exceeds the lower bound, a portion of the donated resources are returned. Thus, Nyx guarantees the "time-sharing" lower bound while maximizing resource utilization. The implementation allows the administrator to set *DonateStep*, balancing quick idle resource donation and the proportional guarantee.

With Admission Control and Capacity Allocation: In a nutshell, cache instances are 1) admitted, 2) allocated space, and 3) governed by Nyx. A PM free-space check, for example, suffices for resource limiting as admission control for a cache; QoS policy requires logic like [36, 37] to predict SLA compliance given existing caches. The cache size is then determined. For instance, it can be set based on the instance's price tier; to enforce QoS, administrators can profile a client's hit-rate v.s. cache space relationship [62] and allocate enough space to meet SLAs. While running, Nyx assumes the admission logic is correct and is unconcerned about the space allocated.

3.5 Cache Instances: PM-Optimized Pelikan

Nyx has been designed to handle any in-memory key-value store; our current implementation is built upon Pelikan – Twitter's in-memory KV cache [17, 75]. We describe the original Pelikan and optimizations for higher PM performance.

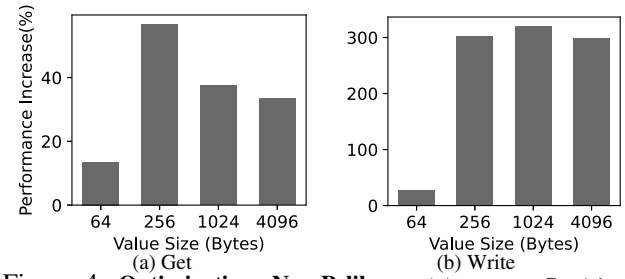


Figure 4: **Optimization: Nyx-Pelikan.** *(a) presents Get (single-thread) throughput improvement due to key-value separation. (b) presents Write (replace, 8 threads) improvement due to changing stores to NT-stores.*

Pelikan (SegCache [75]) maintains a hash table for indexing and segments for storing key-value pairs. Each segment includes items, where each item is a tuple of (key, value, metadata). On a get operation, Pelikan hashes the key to find items. Because of conflicts, multiple keys are likely to be read for a single get. Thus, Pelikan must compare each read item with the key; if the keys match, the value is returned.

When the default version of Pelikan is configured for PM, the hash index is kept in DRAM and the segments in PM. However, this placement is inefficient due to the frequent key accesses in PM: the keys in caching workloads are often much smaller [74] than the granularity of PM access (256B), and small reads perform relatively poorly on PM [73].

Nyx-Pelikan addresses this by separating keys (and metadata) from values into different segments; the keys (and metadata) are placed in DRAM and the values in PM. This design requires DRAM for keys and metadata, which works well because they are typically much smaller than values [73].

As shown previously in Table 2, because non-temporal stores to PM can provide much greater throughput than conventional stores, Nyx-Pelikan uses NT-store. Although non-temporal stores may not benefit from temporal locality in the CPU cache, this loss is negligible on large-scale caching workloads which typically have large working sets. As shown in Figure 4, Nyx-Pelikan improves Pelikan Get performance by up to 55% and set performance by up to 3×.

3.6 Nyx Parameter Values

The values of Nyx's parameters affect its behavior; as previously stated, the appropriate settings depend on the tradeoffs made by administrators. Nyx enables users to configure all of these parameters while also setting defaults.

Nyx follows existing guidelines [38, 63, 77] for policy parameter values' selection. For resource limiting, Nyx uses 10 ms tick and 100 ticks per epoch to limit resource usage offset to 1%. For fair slowdown, Nyx sets the T_{alone} refresh interval to one second to achieve a relatively quick response to workload changes and a within 2% overhead (§4.1).

Nyx provides defaults for newly introduced parameters via sensitivity tests (§4.7). Nyx QoS uses 500MB/s ExperimentStep because it is the smallest step that produces good interference analysis. In interference-aware resource dona-

Trace	Type	Avg.Key/Value Sizes(B)	Operations (Get/Write ratio)
S1	Storage	36/799	0.86/0.13
C1	Computation	67/2439	0.93/0.07
C2	Computation	18/67485	0.52/0.48

Table 3: **Twitter Traces.**

tion, Nyx sets a 10% DonateStep to balance quick donation and steady donator performance. Nyx sets 10ns throttling delay granularity for fine-grained access rate regulation, which is an order of magnitude less than 100ns PM latency. We will discuss potential optimizations like dynamic/adaptive parameters and automatic parameter value selection in §5.

4 Evaluation

We evaluate the overhead of Nyx's mechanisms and how well Nyx provides the sharing policies of resource limit, QoS, fair slowdown, and proportional resource allocation.

Setup: We use a 16-core, single-socket Intel Xeon Gold 5128 CPU @ 2.3GHz server (Ubuntu 18.04), with a 22 MB L3 Cache, 2x16GB DRAM, and 2x128GB Intel Optane DC PMM in app direct mode. We mount an ext4 file system in DAX mode on the PM.

Synthetic Workloads: We begin with synthetic workloads to illustrate key features. Unless specified, the workloads have uniform random accesses to each cache instance, a working set of 10GB per instance, and 4B keys and variable-sized value. To focus on PM accesses, we use get workloads with a high hit ratio (>99 percent). We use in-place replacement for write-heavy workloads; a cache write implies a replace. The cache is warmed to begin.

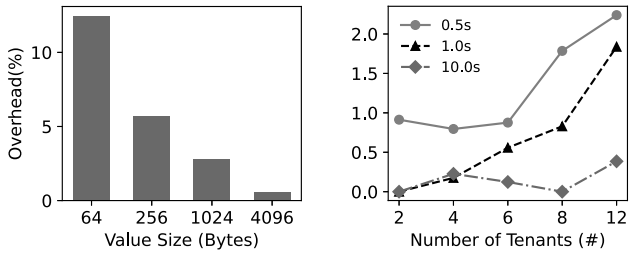
Realistic Workloads: We conclude with three large-scale cache traces from Twitter [74] (Table 3). The traces cover caches with various value sizes (799B to 67845B) and get-percentages (93% to 52%). We pre-load one million operations from the traces and loop through them.

4.1 Mechanisms Overhead

Request Regulation and Resource Usage Estimation: With Nyx, each PM access incurs a call into Nyx-lib, throttling logic, and resource accounting. Figure 5.a shows this can add up to 12% overhead for extremely small value sizes (e.g., a cache line), but less than 6% for access sizes above 256B. Given the benefit of request regulation and resource usage accounting, we believe this overhead is justified.

Interference Analysis: Determining the most interfering client takes longer than simply selecting the client with the greatest bandwidth due to the lag necessary to observe tail latency. In Section 4.3 we will demonstrate the benefit of trading increased analysis time for more precise information.

SlowDown Estimation: The overhead of slowdown estimation is influenced by the time to measure T_{alone} per instance, the frequency of this measurement, and the number of cache instances. We determined that 1ms is a sufficient pause time to accurately determine T_{alone} for a client. Figure 5.b shows that calculating T_{alone} for up to 12 instances adds less than 2.5% overhead, even when performed every 500ms.



(a) Regulation, Accounting Overhead (b) Slowdown Estimation Overhead

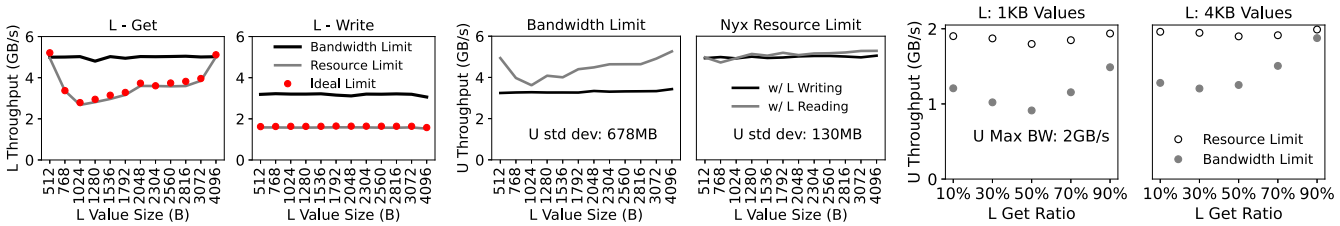
Figure 5: **Mechanisms Overhead.** *(a) shows Nyx request regulation and resource usage accounting overhead (throughput). It is measured with 8-threads get-only caches. A similar percentage of latency overhead was observed. (b) shows Nyx slowdown estimation overhead (throughput). It is measured with 1ms T_{alone} pausing time, different number of clients (x axis) and different frequency (0.5/1/10s) of updating T_{alone} for all caches.*

4.2 Resource Limiting

We demonstrate that Nyx can enforce a true resource limit on PM, in contrast to an approach based only on bandwidth. We begin with a workload containing one unlimited (U) cache and one limited (L) cache. Cache U is a get-heavy cache instance, while Cache L changes: get-only or write-only, with varied value sizes. L has a resource limit of 1.25M 4KB random load OPS, or 42% of the total device resource given that MaxIOPS for 4KB random loads is 3 Million. Defined in terms of bandwidth, this equates to 5GB/s for these 4KB random loads; however, this IOPS limit results in different bandwidths for other workloads.

Figure 6.a shows the bandwidth of L; the target IOPS, in which no more than 42% of the device resource is used, is shown in red. As desired, Nyx always limits L's throughput to the target limit, regardless of L's access pattern (determined by value sizes and read/write). In contrast, a policy based only on bandwidth mistakenly allows L to significantly exceed the target limit, up through the maximum bandwidth of 5GB/s. When L is get-only, this problem is most noticeable when the value size is around 1KB; as previously noted, 1KB accesses result in significant CPU prefetching waste not captured by software-level bandwidth accounting. On the other hand, Nyx's MaxIOPS cost model accurately captures resource usage. Similarly, bandwidth cannot capture PM write cost and fails to properly limit L's throughput.

The impact on the unlimited client (U) is shown in Figure 6.b for the same L workloads. With the bandwidth policy, U's performance depends on L's access pattern. Due to asymmetric read/write cost of PM, whether L performs reads or writes significantly impacts U; similarly, the varied prefetching waste of each access pattern causes up to 45% impact on U. In contrast, Nyx provides U with steady and predictable performance, regardless of L's access pattern: across all of L's workloads, the standard deviation of U's performance is only 130MB/s (bandwidth limit's deviation is 678MB/s). Finally, Figure 6.c shows that when the percentage of gets in L is varied, Nyx provides steady performance for U, whereas a PM-oblivious bandwidth-based approach does not.



(a) L Throughput (b) U Throughput (c) U + Broader L Setups

Figure 6: **Resource Limit: Cache U (unlimited) + Cache L (limited).** *Cache U is get-only. (a) Cache L throughput when resource limit is 5GB/s (1.25M 4KB random load OPS, or 42% of the total device resources). The red dotted line represents L's performance under the "ideal limit", which is calculated as 42% of the current access pattern's MaxIOPS. L is get-only or write-only, and its value sizes varies (x axis). (b) Cache U's performance when colocated with the same L in (a), comparing bandwidth limit and Nyx resource limit. (c) Additional L setups: 1KB/4KB value sizes and 10% - 90% gets. U is a lighter cache than (a) and (b). The label indicates U's max bandwidth when colocated with a 5GB/s cache instance (4KB-value, get-only).*

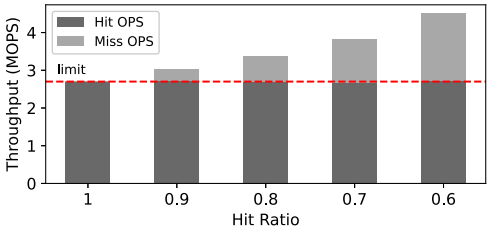


Figure 7: **Resource Limit: Behaviors with a Varying Hit Rate.** *Operations Per Second (OPS) for a Cache with 1KB Get-only workloads when resource limit is 5GB/s. We vary the workloads with different working sets to achieve a different hit rate; note there is no insertion after each miss.*

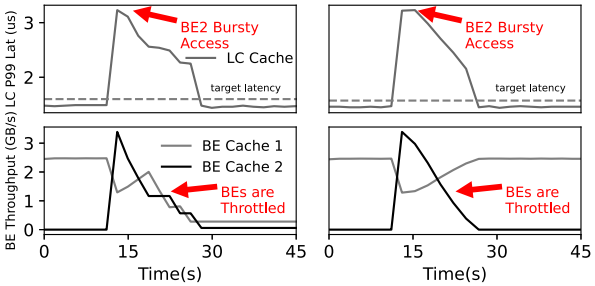


Figure 8: **QoS: Nyx-Caladan vs. Nyx Tuning.** *This figure shows how Nyx and Nyx-Caladan throttle BE caches to ensure LC cache P99 latency. LC cache is colocated with two BE caches; BE1 is get-heavy, B2 is write-heavy (i.e., more interference to LC). BE2 has burst at 12s, breaking LC latency targets. Nyx-Caladan (left) throttles the highest-bw client, whereas Nyx (right) throttles the client with the most interferences to LC. Nyx-Caladan incorrectly throttles BE1, resulting in ×6 less bandwidth for BE1.*

Figure 7 demonstrates Nyx's resource limiting behaviors as the cache hit rate varies. As shown, Nyx restricts PM resource usage from (get) hits. Misses in look-aside caches (e.g., Pelikan) are simply returned after checking the index (in DRAM) and do not use PM resources, so they are not limited.

4.3 QoS-Aware

Nyx can provide QoS guarantees for latency-critical (LC) caches while providing high utilization to best-effort (BE) caches with interference-aware regulation; in contrast, a PM-oblivious approach such as that in Caladan may not be able to deliver the same performance to the BE cache. For comparison, we implemented the Caladan approach in Nyx-Caladan.

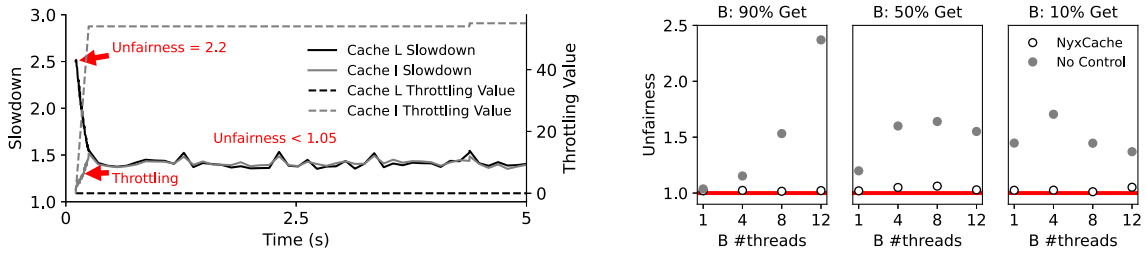
Figure 8 shows an LC cache (P99 latency target of 1.5μs) colocated with two BE caches: BE1 is get-heavy, BE2 is write-heavy. Initially, when BE2 has low throughput and BE1 has moderate throughput of 2.4GB/s, LC meets its P99 objective; however, at 12s, BE2 performs many bursty writes, causing LC's P99 latency to exceed 3μs and violate its target. Both Nyx-Caladan and Nyx resolve the situation by iteratively throttling a BE cache. Nyx-Caladan throttles the cache currently consuming the most bandwidth, shown in the left two subfigures; as a result, Nyx-Caladan throttles both BE1 and BE2, resulting in ×6 less bandwidth for BE1. Nyx, on the other hand, identifies the cache that most interferes with LC as BE2, the write-heavy cache. As a result, Nyx stabilizes to throttling only the correct interference source; after 28 seconds, only BE2 is throttled, and BE1 returns to its original throughput. To summarize, Nyx provides high utilization for multiple caches while guaranteeing each target.

Nyx's convergence time of tens of seconds is similar to prior work such as Parties [31]: the majority of the converging time is spent monitoring tail latencies. As in Parties, Nyx measures tail latency for 500ms because shorter intervals can result in noisy measurements. We leave faster tail latency measurement at network packet queues (as utilized in the original Caladan [39]) for future investigation.

Our experiments reveal that Nyx has an intriguing effect on convergence time: as shown in the Figure, Nyx can bring the LC cache to its target performance in a comparable amount of time to just selecting the cache with the highest bandwidth (which does not require any micro-experiment time). The implication of these results is that, rather than simply acting quickly and throttling any competing instance, Nyx acts correctly and throttles the source of the interference.

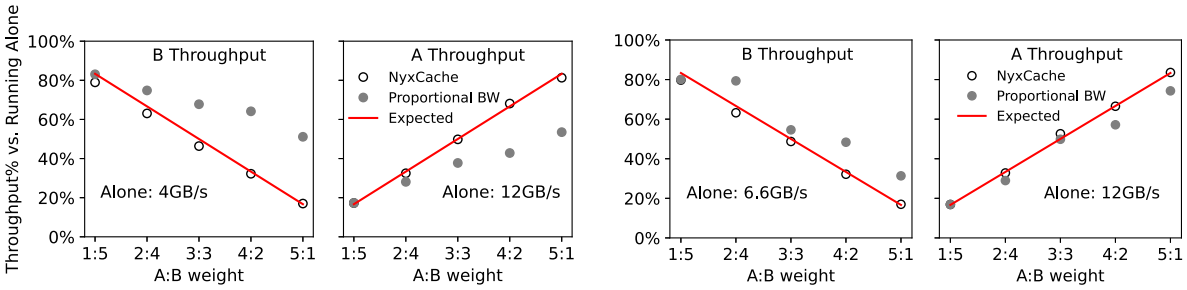
4.4 Fair Slowdown

Nyx implements fair slowdown by iteratively regulating requests according to the measured slowdown of each client (i.e., $\frac{T_{alone}}{T_{share}}$). Figure 9.a shows Nyx's tuning given colocated light and intensive get-heavy caches. Initially, the slowdown of the light cache is 2.2 times higher than that of the intensive cache. Over time, Nyx dynamically increases the throttling of the cache with the minimum slowdown and decreases throttling for the cache with maximum slowdown. Relatively



(a) Tuning Process (Light + Intensive) (b) L + various B

Figure 9: **Fair Slowdown.** *(a) shows how Nyx equalizes slowdown over time for two cache instances (a light one (L) and an intensive one (I)). Both cache instances are get-heavy. (b) shows the unfairness metric when colocating L (a light get-heavy cache) with different B instances (get-heavy -> write-heavy, and light -> intensive). Unfairness = MaxSlowDown / MinSlowDown, the more close to 1, the more fair.*



(a)A: Get-heavy, B: Write-heavy (b) A: Efficient-Get, B: Inefficient-Get

Figure 10: **Proportional Sharing.** *(a) shows A (get-heavy cache) and B (write-heavy cache)'s throughput with different weight configuration. The labels indicate running alone throughput of A and B. With bandwidth allocation, B surpasses its allotted proportional performance. (b) shows A (efficient get-intensive cache, 4KB value sizes) and B (inefficient get-intensive cache, 1KB value sizes).*

quickly, both caches converge to a slowdown near 1.5 and the unfairness metric of $\frac{MaxSlowdown}{MinSlowdown}$ settles near 1.05.

Figure 9.b shows Nyx's fair slowdown policy on a range of caches. Cache L remains a light get-heavy cache; Cache B varies the number of threads and can be get-heavy, 50% mixed, or write-heavy. Without Nyx, L can experience dramatically unfair slowdown (due to PM's complex performance); for example, colocating A with a multi-threaded get-heavy cache B gives unfairness near 2.4. In contrast, Nyx achieves fair slowdown (< 1.05 unfairness) for all 12 cases.

4.5 Proportional Resource Allocation

Nyx achieves proportional resource allocation and guarantees a time-sharing lower bound while performing idle resource re-distribution. We begin with simple scenarios in which two caches that use all their assigned resources are colocated. The scenarios in Figure 10 vary the desired proportional share for A and B along the x-axis; the red line indicates the ideal proportional throughput given their throughput when run alone. Figure 10.a shows that a PM-oblivious bandwidth approach cannot guarantee a proportional share; in particular, the write-intensive B cache obtains up to 3× more throughput than desired and the get-intensive cache A suffers significantly (∼40%). However, by correctly estimating resource usage, Nyx delivers the desired allocation to each cache. Figure 10.b shows a similar effect occurs when efficient-get (value: 4KB) and inefficient-get (value: 1KB) caches are colocated.

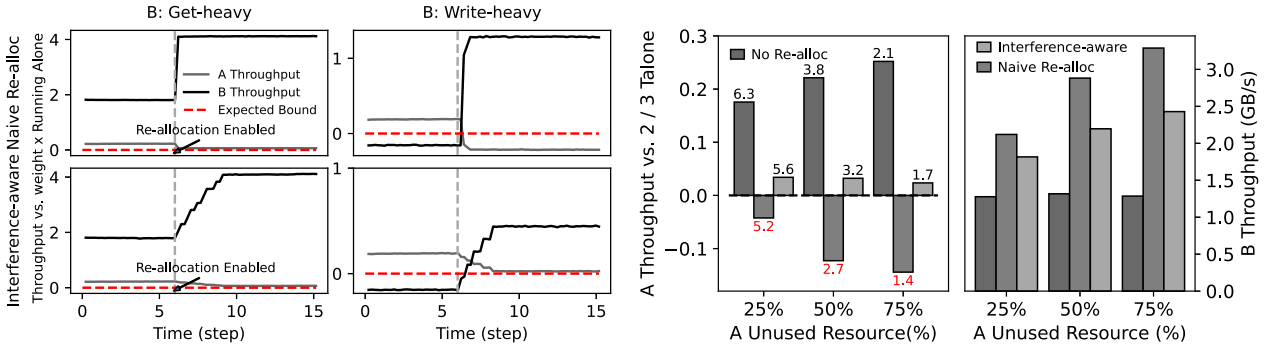
Proportional allocation is more challenging when there are idle resources to be redistributed. Figure 11.a shows two caches A and B, where A uses only 25% of its share. When B is get-heavy (left-top subfigure), A can donate all its idle resources to B; A's performance is slightly degraded, but B receives substantially higher throughput. However, when B is write-heavy (right-top subfigure), if A donates all its idle resources, the higher throughput of B substantially interfere with A, breaching A's time-sharing lower bound (2/3 of A's stand-alone throughput). Therefore, Nyx does not perform naive donation; instead, Nyx donates idle resources in increments while monitoring each cache's slowdown. As shown in the bottom two graphs, Nyx guarantees the time-sharing lower bound for each cache while improving utilization.

We next examine workloads varying the percentage of idle resources in Cache A. When cache B is get-heavy, all of A's idle resources can be safely redistributed to B, and Nyx achieves the same performance for cache B as simple donation (Figure not shown due to space limit). However, when cache B is write-heavy, simple donation of A's idle resources to B violates A's time-sharing bound (Figure 11.b); Nyx accurately protects cache A's performance while still improving the performance of cache B relative to no donation.

4.6 Realistic Traces

Nyx provides isolation for realistic workloads. We demonstrate use cases for resource limiting and slowdown limiting.

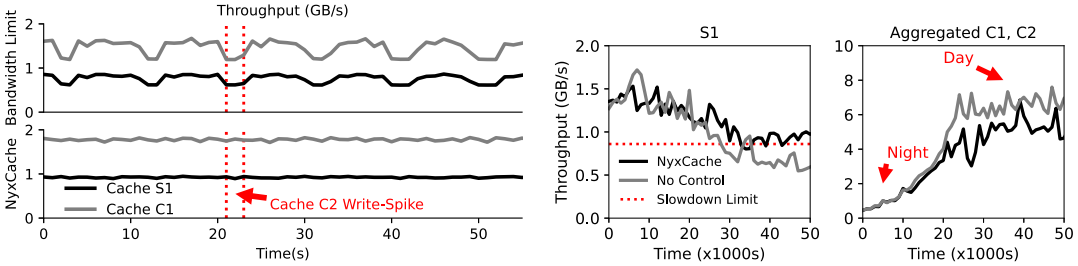
In production workloads, write spikes are common; for example, when a cache is used for ML models, write spikes occur with model parameters are regularly refreshed [74]. Figure 12.a shows how Nyx can isolate caches S1 and C1



(a) Problem of Naive "Donate All" Strategy

(b) Re-distribution between Get and Write-heavy Caches

Figure 11: **Proportional Share: Extra Resource Re-distribution.** *Cache weight A:B is 2:1. (a) shows cache A (light, get-heavy) throughput before and after it donates its extra allocated resource. A has a 75 percent idle resource. Y axis is the normalized difference. When cache B is get-heavy (the top-left figure), A gets nominal performance drop due to donation. However, when cache B is write-heavy (top-right figure), donating cause severe slowdown for A. Unlike naive extra re-distribution, Nyx (two bottom figures) ensures that tenant A's performance is always more than two-thirds of its running alone performance. TimeStep = 2ms. (b) shows A's slowdown (left figure) and B's throughput (right figure) before and after A donating extra resource. A is get-heavy and B is write-heavy. The label indicates absolute throughput number. Naive extra resource allocation can easily break isolation guarantee, while Nyx always ensures it.*



(a) Protecting Caches from Write Spikes

(b) Limiting S1 Slowdown During Day and Night

Figure 12: **Realistic Traces.** *(a) shows the performance of Cache S1 and C1 when colocated with Cache C2. Cache C2 has write spikes. Nyx (bottom figure) can isolate write spikes, whereas bandwidth limits cannot (top figure). (b) shows the performance of Cache S1 (the cache we guarantee its slowdown is always smaller than 1.5×). S1 is colocated with C1 and C2; both C1 and C2 have a strong diurnal pattern (light during the night, and intensive during the day). Without Nyx, S1 performance plummets during the day (because the impact from C1 and C2), discouraging sharing. However, Nyx can always offer reasonable performance (e.g. within x1.5 slowdown vs. running alone). The red line represents S1's performance guarantee.*

from (added) write spikes in cache C2. If resource limiting is based only on the bandwidth of C2, S1 and C1 suffer when C2 experiences write spikes. However, Nyx's resource-limit policy can cap C2's resource usage (at 4GB/s, defined as 1M 4KB random load OPS) to keep S1 and C1 steady.

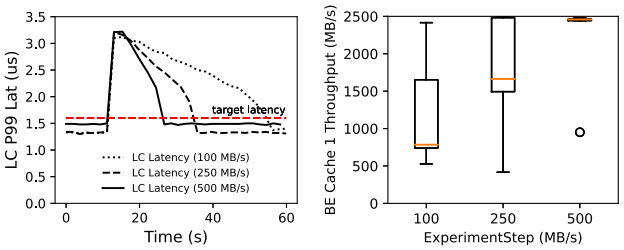
Nyx can also protect the performance of critical caches. To encourage tenants to use multi-tenant PM environments, some caches must be guaranteed performance similar to exclusive use of the PM device. In the experiment shown in Figure 12.b, S1 (the critical cache) is colocated with C1 and C2 which have diurnal patterns [74]. With no control (gray lines), the performance of S1 drops below its target during the day due to the heavy accesses of C1 and C2. However, Nyx can establish a hard limit of slowdown (e.g., 1.5) for S1. As observed, Nyx keeps S1 performance loss within a fair range.

4.7 Parameters Sensitivity Analysis

Here, we present the sensitivity analysis of Nyx behaviors with different ExperimentStep and DonateStep values.

The ExperimentStep affects the Nyx interference analysis's accuracy. As shown in Figure 13.b, using the same configuration as Figure 8, a smaller ExperimentStep is more likely to result in a lower BE 1 final throughput. When ExperimentStep is small, the tail latency change is more likely to be due to measurement noise rather than interference, leading to a less accurate interference analysis. Our experiments suggest an ExperimentStep of at least 500MB/s. ExperimentStep also influences how quickly the Nyx QoS can ensure LC tail latency. As shown in Figure 13.a, a larger ExperimentStep indicates faster convergence. However, it increases the risk of over-throttling BE caches and lowering system utilization. ExperimentStep in Nyx QoS defaults to 500MB/s for good interference analysis and high system utilization while maintaining a reasonable convergence time.

Figure 14 shows how DonateStep affects Nyx proportional resource allocation. A larger DonateStep causes faster idle resource donation, but also potentially large performance fluctuations (e.g., 60% DonateStep, 12s and 18s in the figure). At runtime, cache throughput always varies slightly, causing donation adjustments. These adjustments are subtle with small DonateSteps but significant with large ones. The fluctuation harms donors by slowing them down at times (exceeding the limit). Nyx uses a 10% DonateStep, which balances between quick resource donation and steady donor performance.



(a) Convergence Time (b) BE 1 Final Throughput

Figure 13: **QoS: ExperimentStep Sensitivity Analysis.** *Same as in Figure 8. (a) shows how fast Nyx QoS can ensure LC P99 latency varying ExperimentSteps. (b) shows BE 1 final throughput (boxplot, five runs) varying ExperimentSteps; BE 2 has near zero final throughput in all cases.*

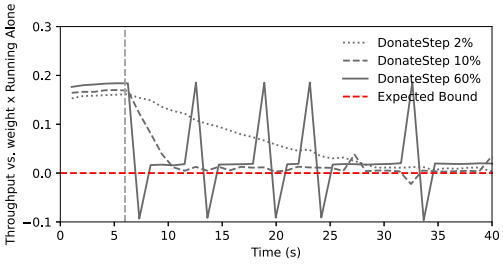


Figure 14: **Proportional Share: DonateStep Sensitivity Analysis.** *We use the same setup as in Figure 11 when B is write-heavy. This figure shows A, the donator's throughput over time with different DonateStep.*

5 Discussion

Beyond Basic Policies: Nyx can be extended to more sophisticated policies for more complex setups. For instance, a proportional sharing policy can be applied across groups of caches. Then, within a group, another sharing policy (e.g., QoS) can be enforced. We leave a full study as a future work.
Multi-tenant Caching Alternatives: Nyx manages caches, each with its own space. There are alternatives to shared caching; for instance, a single large instance can be shared by multiple users [60]. This model can make use of the Nyx resource usage accounting and interference analysis techniques. However, it may create new problems like: how should users be charged for PM writes to commonly cached objects?
Smarter Parameter Value Selection: i) Adaptive parameters can be beneficial, e.g., the DonateStep can be larger when it is far from the threshold (for quick donation) and smaller when it is close (to avoid performance fluctuations). ii) Autotuning [46, 68] may ease the load for choosing parameter values. We leave these optimizations as future work.
Security: Nyx policies can be attackable, e.g., in resource limiting, an adversary client may limit its access in the first ticks while putting significant load in the last. A solution would be to use randomized measuring points rather than fixed ones. We leave Nyx security studies as future work.

6 Related Work

Multi-tenant in-mem key-value caching: Our work builds on past research in multi-tenant in-memory key-value cache systems. These efforts include techniques for allocating space across tenants [29, 32, 34, 60, 62] as well as optimization of individual cache instances [25, 27, 28, 33, 42, 54, 75]. Our work instead focuses on the challenges of access regulation and information extraction when many caches share PM.
PM Caching: There have been efforts to integrate PM with individual caching systems. Previous work covers databases [50, 72, 81], file systems [22, 48, 80], in-memory key-value caches [6, 19, 21], and general policies [26, 27]. However, to the best of our knowledge, we are the first to address PM issues in multi-tenant caching settings.
PM Interference: Several efforts have characterized PM devices [45, 69, 71, 73]. However, only a few have investigated the interference effect in PM. To our knowledge, Dicio [55] is the first work in this space. Both Dicio and our work observe the different read-write interference effect in PM. However, the goals of Dicio and Nyx differ. Dicio's purpose is to identify when PM DIMM bandwidth is saturated. Dicio approximates this by using the write pending queue (WPQ) delay as a heuristic. We, on the other hand, aim to provide mechanisms for per-client (not per-DIMM) resource usage accounting, slowdown estimation, and cross-client interference analysis. Dicio protects a single LC task from a single BE task, while our QoS policy applies to multiple clients. Dicio acknowledges that deciding which best-effort task to throttle, with PM media-level statistics, was challenging (and hence not done); we address this issue with a run-time method for interference analysis. Finally, Dicio extends Caladan [39] to use CPU scheduling to regulate PM accesses. This approach is applicable to all applications, including cache, but requires application modifications to use Caladan's unique runtime system (not fully Linux compatible). We leave CPU scheduling approaches for PM regulation to future work.
Sharing Other Resources: Efforts have been made to manage and share other resources such as network, CPU, LLC, storage devices, and locks [31, 39–41, 43, 44, 51, 56, 57, 59, 65]. They are essentially orthogonal to our work; we plan to integrate PM management into these systems in the future.

7 Conclusion

We demonstrated that prior DRAM or storage device-intended approaches for access regulation, resource-usage estimation, and interference analysis fail to work on PM due to its unique properties. We introduced Nyx, which enables these mechanisms in a lightweight manner without hardware support. We showed that Nyx can support a variety of multi-tenant cache sharing policies, meeting performance or sharing goals better than earlier DRAM or storage approaches.
Acknowledgments. We thank Ali R. Butt (our shepherd), the anonymous reviewers, and ADSL members for their valuable input. This material was supported by funding from NSF CNS-1838733, CNS-1763810, Google, VMware, Intel, Seagate, Samsung, and Microsoft. The authors' opinions and findings may not reflect those of NSF or other institutions.

References

[1] Amazon elasticache. https://aws.amazon.com/elasticache/.

[2] Amazon elasticache pricing. https://aws.amazon.com/elasticache/pricing/.

[3] Aws elasticache. https://aws.amazon.com/elasticache/redis/customers/.

[4] Budget fair queueing i/o scheduler. http://algo.ing.unimo.it/people/paolo/disk_sched/.

[5] Caching at reddit. https://redditblog.com/2017/1/17/caching-at-reddit/.

[6] Caching on pmem: an iterative approach. yue yao. https://www.snia.org/educational-library/caching-pmem-iterative-approach-2020.

[7] Google memcache resource limit. https://cloud.google.com/appengine/docs/standard/python/memcache.

[8] Intel mba issue with pm. https://github.com/intel/intel-cmt-cat/issues/170.

[9] Intel memory bandwidth allocation (mba). https://software.intel.com/content/www/cn/zh/develop/articles/introduction-to-memory-bandwidth-allocation.html.

[10] Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[11] Intel Optane DIMM Pricing. https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html.

[12] I/o scheduling. https://en.wikipedia.org/wiki/i/o_scheduling.

[13] ipmctl mediareads, mediawrites. https://docs.pmem.io/ipmctl-user-guide/instrumentation/show-device-performance.

[14] Memcached. https://memcached.org/.

[15] Memcached stats. https://docs.oracle.com/cd/E17952_01/mysql-5.6-en/ha-memcached-stats.html.

[16] Memcachier. https://www.memcachier.com/.

[17] Pelikan - twitter. https://twitter.github.io/pelikan/.

[18] Pelikan cache - taming tail latency and achieving predictability. https://twitter.github.io/pelikan/2020/benchmark-adq.html.

[19] Pmem redis. https://github.com/pmem/pmem-redis.

[20] Redis enterprise cloud. https://redis.com/redis-enterprise-cloud/overview/.

[21] The volatile benefit of persistent memory - memcached. https://memcached.org/blog/persistent-memory/.

[22] Thomas E Anderson, Marco Canini, Jongyul Kim, Dejan Kostić, Youngjin Kwon, Simon Peter, Waleed Reda, Henry N Schuh, and Emmett Witchel. Assise: Performance and Availability via Client-local NVM in a Distributed File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1011–1027, 2020.

[23] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, pages 53–64, 2012.

[24] Luiz Barroso, Mike Marty, David Patterson, and Parthasarathy Ranganathan. Attack of the Killer Microseconds. *Communications of the ACM*, 60(4):48–54, 2017.

[25] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, pages 389–403, 2018.

[26] Nathan Beckmann, Phillip B Gibbons, Bernhard Haeupler, and Charles McGuffey. Writeback-aware caching. In *Symposium on Algorithmic Principles of Computer Systems*, pages 1–15. SIAM, 2020.

[27] Nathan Beckmann, Phillip B Gibbons, and Charles McGuffey. Block-granularity-aware caching. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*, pages 414–416, 2021.

[28] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The cachelib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020.

[29] Daniel S Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. Robinhood: Tail latency aware caching–dynamic reallocation from cache-rich to cache-poor. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 195–212, 2018.

[30] E Chen, D Apalkov, Z Diao, A Driskill-Smith, D Druist, D Lottis, V Nikitin, X Tang, S Watts, S Wang, et al. Advances and future prospects of spin-transfer torque random access memory. *IEEE Transactions on Magnetics*, 46(6):1873–1878, 2010.

[31] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[32] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[33] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 379–392, 2016.

[34] Asaf Cidon, Daniel Rushton, Stephen M Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 321–334, 2017.

[35] Craciunas, Silviu S and Kirsch, Christoph M and Röck, Harald. I/o resource management through system call scheduling. *ACM SIGOPS Operating Systems Review*, 42(5):44–54, 2008.

[36] Christina Delimitrou, Nick Bambos, and Christos Kozyrakis. Qos-aware admission control in heterogeneous datacenters. In *10th International Conference on Autonomic Computing (ICAC 13)*, pages 291–296, 2013.

[37] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

[38] Eiman Ebrahimi, Chang Joo Lee, Onur Mutlu, and Yale N Patt. Fairness via source throttling: a configurable and high-performance fairness substrate for multi-core memory systems. *ACM Sigplan Notices*, 45(3):335–346, 2010.

[39] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 281–297, 2020.

[40] Ajay Gulati, Irfan Ahmad, Carl A Waldspurger, et al. Parda: Proportional allocation of resources for distributed storage access. In *FAST*, volume 9, pages 85–98, 2009.

[41] Ajay Gulati, Arif Merchant, and Peter J Varman. mclock: Handling throughput variability for hypervisor io scheduling. In *OSDI*, volume 10, pages 437–450, 2010.

[42] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. Lama: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 57–69, 2015.

[43] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K Qureshi. Flashblox: Achieving both performance isolation and uniform lifetime for virtualized ssds. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 375–390, 2017.

[44] Călin Iorgulescu, Reza Azimi, Youngjin Kwon, Sameh Elnikety, Manoj Syamala, Vivek Narasayya, Herodotos Herodotou, Paulo Tomita, Alex Chen, Jack Zhang, et al. Perfiso: Performance isolation for commercial latency-sensitive services. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 519–532, 2018.

[45] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. Basic performance measurements of the intel optane dc persistent memory module. *arXiv preprint arXiv:1903.05714*, 2019.

[46] Yichen Jia and Feng Chen. Kill two birds with one stone: Auto-tuning rocksdb for high bandwidth and low latency. In *2020 IEEE 40th International Conference on Distributed Computing Systems (ICDCS)*, pages 652–664. IEEE, 2020.

[47] Ram Srivatsa Kannan, Michael Laurenzano, Jeongseob Ahn, Jason Mars, and Lingjia Tang. Caliper: Interference estimator for multi-tenant environments sharing architectural resources. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(3):1–25, 2019.

[48] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[49] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable dram alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*, pages 2–13, 2009.

[50] Gang Liu, Leying Chen, and Shimin Chen. Zen: a high-throughput log-free oltp engine for non-volatile main memory. *Proceedings of the VLDB Endowment*, 14(5):835–848, 2021.

[51] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 450–462, 2015.

[52] Jason Mars, Lingjia Tang, Robert Hundt, Kevin Skadron, and Mary Lou Soffa. Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations. In *Proceedings of the 44th annual IEEE/ACM International Symposium on Microarchitecture*, pages 248–259, 2011.

[53] Kyle J Nesbit, Nidhi Aggarwal, James Laudon, and James E Smith. Fair queuing memory systems. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*, pages 208–222. IEEE, 2006.

[54] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. Scaling memcache at facebook. In *Proceedings of the 10th Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 385–398, Lombard, Illinois, April 2013.

[55] Jinyoung Oh and Youngjin Kwon. Persistent Memory Aware Performance Isolation with Dicio. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems*, pages 97–105, 2021.

[56] Jinsu Park, Seongbeom Park, and Woongki Baek. Copart: Coordinated partitioning of last-level cache and memory bandwidth for fairness-aware workload consolidation on commodity servers. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–16, 2019.

[57] Jinsu Park, Seongbeom Park, Myeonggyun Han, Jihoon Hyun, and Woongki Baek. Hypart: a hybrid technique for practical memory bandwidth partitioning on commodity servers. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*, pages 1–14, 2018.

[58] Stan Park and Kai Shen. Fios: a fair, efficient flash i/o scheduler. In *FAST*, volume 12, pages 13–13, 2012.

[59] Yuvraj Patel, Leon Yang, Leo Arulraj, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Michael M Swift. Avoiding scheduler subversion using scheduler-cooperative locks. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–17, 2020.

[60] Qifan Pu, Haoyuan Li, Matei Zaharia, Ali Ghodsi, and Ion Stoica. Fairride: Near-optimal, fair cache sharing. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 393–406, 2016.

[61] Kai Shen and Stan Park. Flashfq: A fair queueing i/o scheduler for flash-based ssds. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 67–78, 2013.

[62] Ioan Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony Rowstron, and Tom Talpey. Software-defined caching: Managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*, pages 174–181. ACM, 2015.

[63] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 62–75. IEEE, 2015.

[64] Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu. Mise: Providing performance predictability and improving fairness in shared main memory systems. In *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, pages 639–650. IEEE, 2013.

[65] Arash Tavakkol, Mohammad Sadrosadati, Saugata Ghose, Jeremie Kim, Yixin Luo, Yaohua Wang, Nika Mansouri Ghiasi, Lois Orosa, Juan Gómez-Luna, and Onur Mutlu. Flin: Enabling fairness and enhancing performance in modern nvme solid state drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 397–410. IEEE, 2018.

[66] Team at Redis. Redis. https://redis.io/, 2021.

[67] Matthew Wachs, Michael Abd-El-Malek, Eno Thereska, and Gregory R Ganger. Argon: Performance Insulation for Shared Storage Servers. In *FAST*, volume 7, pages 5–5, 2007.

[68] Benjamin Wagner, André Kohn, and Thomas Neumann. Self-tuning query scheduling for analytical workloads. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1879–1891, 2021.

[69] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. Characterizing and modeling non-volatile memory systems. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 496–508. IEEE, 2020.

[70] H-S Philip Wong, Heng-Yuan Lee, Shimeng Yu, Yu-Sheng Chen, Yi Wu, Pang-Shiu Chen, Byoungil Lee, Frederick T Chen, and Ming-Jinn Tsai. Metal–oxide rram. *Proceedings of the IEEE*, 100(6):1951–1970, 2012.

[71] Kan Wu, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. Towards an unwritten contract of intel optane ssd. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19). USENIX Association, Renton, WA*, 2019.

[72] Kan Wu, Zhihan Guo, Guanzhou Hu, Kaiwei Tu, Ramnatthan Alagappan, Rathijit Sen, Kwanghyun Park, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 307–323, 2021.

[73] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. *arXiv preprint arXiv:1908.03583*, 2019.

[74] Juncheng Yang, Yao Yue, and KV Rashmi. A large scale analysis of hundreds of in-memory cache clusters at twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 191–208, 2020.

[75] Juncheng Yang, Yao Yue, and Rashmi Vinayak. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *NSDI*, pages 503–518, 2021.

[76] Suli Yang, Tyler Harter, Nishant Agrawal, Salini Selvaraj Kowsalya, Anand Krishnamurthy, Samer Al-Kiswany, Rini T Kaushik, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Split-level i/o scheduling. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 474–489, 2015.

[77] Heechul Yun, Gang Yao, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 55–64. IEEE, 2013.

[78] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Jihong Kim, et al. Flashshare: punching through server storage stack from kernel to firmware for ultra-low latency ssds. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 477–492, 2018.

[79] Jishen Zhao, Onur Mutlu, and Yuan Xie. Firm: Fair and high-performance memory control for persistent memory systems. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 153–165. IEEE, 2014.

[80] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. Ziggurat: a tiered file system for non-volatile main memories and disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207–219, 2019.

[81] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. Spitfire: A three-tier buffer manager for volatile and non-volatile memory. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2195–2207, 2021.