# Holpaca: Holistic and Adaptable Cache Management for Shared Environments

José Pedro Peixoto
INESC TEC & University of Minho
Braga, Portugal
jose.p.peixoto@inesctec.pt

Alexis Gonzalez
Florida International University
Miami, FL, USA
agonz825@fiu.edu

Janki Bhimani
Florida International University
Miami, FL, USA
jbhimani@fiu.edu

Raju Rangaswami
Florida International University
Miami, FL, USA
raju@cs.fiu.edu

Cláudia Brito
INESC TEC & University of Minho
Braga, Portugal
claudia.v.brito@inesctec.pt

João Paulo
INESC TEC & University of Minho
Braga, Portugal
joao.t.paulo@inesctec.pt

Ricardo Macedo
INESC TEC & University of Minho
Braga, Portugal
ricardo.g.macedo@inesctec.pt

## Abstract

Modern data-intensive systems rely on in-memory caching to achieve high throughput and low latency. CacheLib, Meta's general-purpose caching engine, provides high performance and flexibility for building specialized caches for a variety of applications. However, despite its wide adoption in large-scale infrastructures, CacheLib's data management mechanisms exhibit inefficiencies in shared environments. Particularly, its static and uncoordinated memory allocation leads to fragmented resource usage, unfair memory distribution, and degraded performance across tenants and instances.

We present Holpaca, a general-purpose caching middleware that enables holistic and adaptable orchestration of shared caching environments. Holpaca introduces a shim data layer co-located with each cache instance and a centralized orchestrator with system-wide visibility, enabling global memory management and per-tenant QoS policies. Using production traces from Twitter, results show that, by continuously readjusting memory allocations based on workload dynamics, Holpaca achieves up to 3× higher throughput in multi-tenant and 2.2× improvement in multi-instance settings over CacheLib's rigid built-in mechanisms.

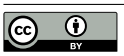
## CCS Concepts

• **Computer systems organization** → *Client-server architectures*; **Cloud computing**; • **Software and its engineering** → Middleware; **Memory management**.

## Keywords

Caching; Memory Systems; Software-Defined Storage; Miss-Ratio Curves; Quality-of-Service

**ACM Reference Format:**
José Pedro Peixoto, Alexis Gonzalez, Janki Bhimani, Raju Rangaswami, Cláudia Brito, João Paulo, and Ricardo Macedo. 2026. Holpaca: Holistic and Adaptable Cache Management for Shared Environments. In *Proceedings of the 17th ACM/SPEC International Conference on Performance Engineering (ICPE '26), May 04–08, 2026, Florence, Italy.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3777884.3797013




*ICPE '26, Florence, Italy*

ACM ISBN 979-8-4007-2325-4/2026/05
https://doi.org/10.1145/3777884.3797013

## 1 Introduction

Data-intensive systems, such as databases, key-value stores, content delivery networks (CDNs), and machine learning engines, are fundamental to modern I/O infrastructures [22]. To process large data volumes efficiently, these systems rely on in-memory caching to increase throughput and reduce latency of costly accesses to persistent storage. Typically, each system employs a cache fine-tuned for its specific design and workload characteristics, such as read-write ratio, access pattern, I/O granularity (*e.g.,* block, file, object), and concurrency model (*i.e.,* single- *vs.* multi-tenant) [12, 36].

Despite these differences, caching systems share common design goals and challenges (*e.g.,* serialization, memory allocation and placement, eviction policies, concurrency control). However, the absence of a unified cache abstraction has led to fragmented features and redundant engineering efforts, resulting in significant development and maintenance overheads. To address these issues, Meta introduced CacheLib [12], a general-purpose caching library that provides building blocks for designing high-performance caches. Through a flexible and extensible API covering cache indexing, thread-safe mechanisms, configurable eviction policies, customizable memory management strategies, and multi-tenancy isolation, CacheLib enables system designers to build specialized cache instances tailored to their use cases without sacrificing performance. Its success stems from configurability and performance portability, allowing applications to inherit caching improvements transparently. Today, CacheLib powers several large-scale systems, including CDNs, recommendation engines, and databases at Meta [12, 20], caching engines at Twitter [4, 44], key-value stores at Pinterest [6], and has been adopted in various research projects [3, 12, 35, 37, 41].

**Challenges.** Given its widespread adoption, we ask a fundamental question: *To what extent are CacheLib's memory management mechanisms suited for the dynamic and resource-constrained nature of shared environments, that dominate modern data-intensive infrastructures?* To address this question, and as our first contribution, we conducted a comprehensive study characterizing the performance

and resource efficiency of CacheLib's memory management mechanisms under both multi-tenant and multi-instance[1] environments, where we report the following key findings (§3). In multi-tenant scenarios, enforcing uniform memory allocation strategies across heterogeneous workloads (*i.e.,* distinct load intensity and skewness) leads to severe performance imbalance and inefficient memory usage, degrading throughput by up to 40% relative to workload-aware configurations (§3.1). This stems from CacheLib's non-adaptable nature, as even though it exposes many tunable knobs (*e.g.*, eviction policies, memory allocation, rebalancing schemes), these parameters are typically defined at initialization. To address this, CacheLib introduced a self-tuning mechanism that enables dynamic memory reallocation across tenants, improving the default static behavior used in most production environments. However, we observe that while some tenants attain performance benefits, others experience resource starvation, degrading their performance when compared to static allocations (§3.1). Such behavior may be acceptable in scenarios that prioritize global throughput, but is otherwise problematic when per-tenant quality-of-service (QoS) guarantees or performance isolation are required.

Further, when multiple services run co-located in the same server, it is common practice to have independent cache instances for each of them [9, 12, 14, 23, 25, 40]. This may be required to ensure workload isolation or to prevent a shared cache engine from becoming a performance bottleneck. As shown in §5.5, multi-tenant configurations generally scale worse, performance-wise, than multi-instance setups. Currently, in such multi-instance deployments, each CacheLib instance operates in isolation, without coordination or shared visibility into global memory usage. As a result, some instances become underprovisioned while others remain overprovisioned, resulting in inefficient and fragmented memory resources. For instance, under our testing scenario, this lack of coordination results in performance losses of up to 30% compared to coordinated configurations (§3.2). Notably, multi-instance memory management has not been addressed by previous work (even for cache engines other than CacheLib). Addressing this challenge requires finding the right design and mechanisms to efficiently coordinate isolated instances, without sacrificing the scalability of such deployments.

**This work.** Motivated by these findings, as our second contribution, we propose Holpaca, a general-purpose caching middleware that improves the performance and resource efficiency of shared caching environments. Holpaca extends CacheLib's design by adopting ideas from the software-defined paradigm [27, 29], following a decoupled design that separates the caching mechanisms (*e.g.,* memory pools) from the policies that govern them (*e.g.,* memory allocation strategy). Specifically, it introduces two complementary components. The *data layer*, co-located with each CacheLib instance (forming an *agent*), acts as a shim layer between the application and the host caching system, transparently intercepting cache requests, collecting runtime metrics and statistics (*e.g.,* throughput and latency, hit ratio, miss ratio curves, workload skew) and continuously adjusting the underlying building blocks. Agents are then collectively controlled by a centralized *orchestrator* with system-wide visibility, which continuously monitors performance characteristics among the different tenants/instances and dynamically reallocates memory according to a configurable optimization goal (*e.g.,* maximizing throughput, or ensuring performance isolation). This design enables coordinated memory management decisions across tenants and instances while adding capabilities absent in the current caching engine (*e.g.,* multi-instance coordination and QoS support), without compromising the scalability of such scenarios.

[1]We define a "*tenant*" as each individual workload directed to a CacheLib instance, which may serve multiple tenants. *"Multi-instance"* refers to multiple CacheLib instances co-located on the same node.

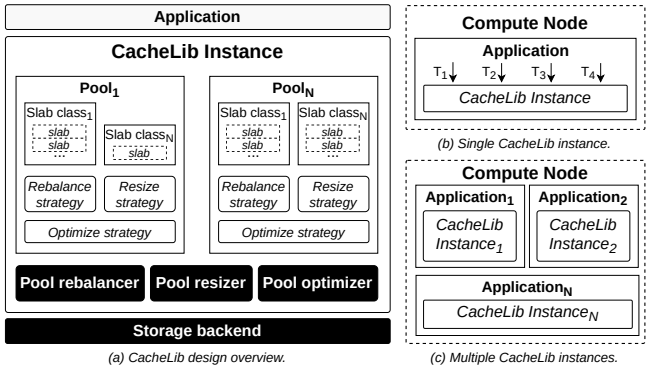



**Figure 1:** CacheLib design overview (a) and typical deployment models observed in production settings (b-c).

To demonstrate Holpaca's flexibility, we implement two resource management strategies targeting distinct optimization goals (§4.3). The first, a *global throughput maximization* algorithm, dynamically reclaims memory from tenants/instances with minimal impact on global throughput and reallocates it to those that have a higher impact while preventing starvation. The second, a *QoS-aware throughput maximization* strategy, allows user-defined performance prioritization (*e.g.,* minimum throughput), ensuring that critical workloads receive adequate resources even under contention.

**Evaluation.** We validate the performance and resource efficiency of Holpaca through a comprehensive experimental evaluation, conducted under both multi-tenant and multi-instance scenarios, using production traces from Twitter [43] (§5). We compare Holpaca against CacheLib's default configuration (*baseline*) and its dynamic auto-tuning mode (*optimizer*). Under multi-tenant scenarios, Holpaca achieves up to 3× higher throughput than the *baseline* and up to 1.6× over the *optimizer*, while preventing starvation in tenants with low-skew workload distributions. Under multi-instance settings, Holpaca outperforms both configurations up to 2.2×, highlighting the benefits of holistic memory management. Finally, we demonstrate that Holpaca can prioritize critical workloads and enforce per-instance QoS objectives. Holpaca is publicly available at **https://github.com/dsrhaslab/Holpaca**.

## 2 Background

This section provides background on the CacheLib engine, outlining its design, core functionalities, and applicability.

### 2.1 CacheLib overview

CacheLib [12] is an embeddable caching library developed by Meta, designed for high performance in concurrent environments. Its architecture emphasizes memory efficiency, workload isolation, and

runtime adaptability. It exposes a thread-safe interface that allows users to store and retrieve data using a key-value pair abstraction. At its core, CacheLib comprises four main components, as depicted in Fig. 1 *(a)*. The *memory pool* provides isolated memory regions to store and service data items, each dedicated to different tenants and configured with independent allocation strategies, eviction queues, and memory limits. The *pool rebalancer* and *pool resizer* minimize memory waste and fragmentation by redistributing memory within and across pools, respectively. The *pool optimizer* improves overall memory usage by dynamically resizing each pool.

**Memory allocation and fragmentation management.** CacheLib employs a slab-based memory allocation scheme to minimize external memory fragmentation. Rather than allocating variable-sized chunks, it partitions memory into fixed-size slabs, each associated with a (slab) class that stores items of similar size. Each slab class manages its own memory region, and a single *pool* may contain multiple slab classes. Once memory has been allocated to slab classes, the system becomes static, which can lead to *slab calcification* [24]. This phenomenon occurs when slabs remain assigned to classes of item sizes no longer accessed (*e.g.,* due to changes in workload popularity or distribution), resulting in wasted memory and allocation failures in other slab classes. To mitigate this, CacheLib integrates a *pool rebalancer*, a background worker that periodically transfers slabs from underutilized classes to those with higher demand for each pool. Since a single slab can store multiple key-value pairs, both hot and cold, the cache performance is sensitive to which slabs are moved. As such, each pool employs a configurable *rebalance strategy* that selects victim slabs based on a given optimization goal (*e.g.,* maximize hit ratio, ensure fairness).

**Multi-tenancy support and memory reclamation.** To support concurrent workloads, CacheLib enables the creation of multiple memory pools within the same instance, each isolating a specific workload issued by a tenant. By default, each pool is assigned a static memory limit on creation. While users can adjust this limit at runtime, increasing a pool's memory allocation does not immediately guarantee additional memory, as allocations may remain held by other pools until explicitly released. Eviction is also insufficient, as it is triggered independently within each pool and only releases memory for incoming items for that specific pool. To ensure new memory limits are respected, CacheLib implements a *pool resizer*, a background worker that continuously monitors all memory pools and reclaims slabs from those exceeding their capacity. Similar to the pool rebalancer, the pool resizer must determine which slabs to release first, which depends on the optimization goal. For this, each memory pool implements a *resize strategy* (*e.g.,* maximize hit ratio, fair eviction age) to guide the resizing process.

**Adaptive memory management.** While the pool resizer automatically enforces memory limits, it still requires users to manually adjust pool sizes. As workloads evolve over time, manually tracking and predicting per-pool memory needs is challenging. To reduce manual tuning, CacheLib implements the *pool optimizer*, a background worker that continuously monitors memory pools and automatically adjusts their memory limits [3]. Similar to the rebalancer and resizer, the *pool optimizer* relies on a configurable *optimization strategy* to rank pools and determine a *victim* (to shrink) and a *receiver* (to grow), redistributing capacity between them. Currently, CacheLib only provides a single optimization strategy that analyzes access patterns at the tail of the eviction queue, only eligible for LRU-2Q, to estimate workload pressure and moves slabs from overprovisioned to underprovisioned pools [3].

## 2.2 CacheLib operation workflow

CacheLib separates application handling and system maintenance into foreground and background workflows, respectively.

**Foreground flow.** Applications interact with the cache by inserting and retrieving data objects through a key-value pair abstraction. On writes, the application first requests CacheLib to allocate memory within a specific memory pool and copies the data into the allocated region. If successful, the application can then commit the write to the cache, making the item accessible by its key. Furthermore, since CacheLib is commonly deployed as an in-memory cache, the application is responsible for storing data in persistent storage. On reads, given a key, CacheLib searches its internal structures, returning the item if found or a cache miss otherwise, leaving data retrieval from persistent storage to the application.

**Background flow.** The background flow maintains the cache's internal structure and adapts it to workload changes through background workers, including the *pool rebalancer*, *resizer*, and *optimizer* (§2.1). These tasks run periodically, asynchronously, and independently of the foreground flow, allowing CacheLib to adapt to workload changes without interrupting application service.

## 2.3 Usage in production environments

CacheLib supports multiple deployment scenarios, depending on the number of applications and tenants involved. This section highlights two common production setups that serve as the foundation for the discussion in the following sections.

**Single instance, multiple tenants.** As depicted in Fig. 1 *(b)*, a single application or service, such as CDNs, key-value stores, social-graph systems, and databases, can use a single CacheLib instance to support multiple tenants [12]. Tenants are typically represented by distinct threads, processes, or components with different request patterns and memory needs. Since these services often include multiple workflows accessing the same cache, each workflow can instead operate over a separate memory pool within the same CacheLib instance, ensuring workload isolation and minimizing interference across tenants.

**Multiple instances.** As depicted in Fig. 1 *(c)*, multiple CacheLib instances are deployed on the same compute node, each serving a different application or service. This is a common setup in large and complex data-intensive software stacks, such as those used by Meta, Hadoop, Uber, Amazon, and more, where isolated services running co-located in the same server require dedicated cache instances with individual configurations, allocation policies, and fixed memory limits [9, 12, 14, 23, 25, 40].

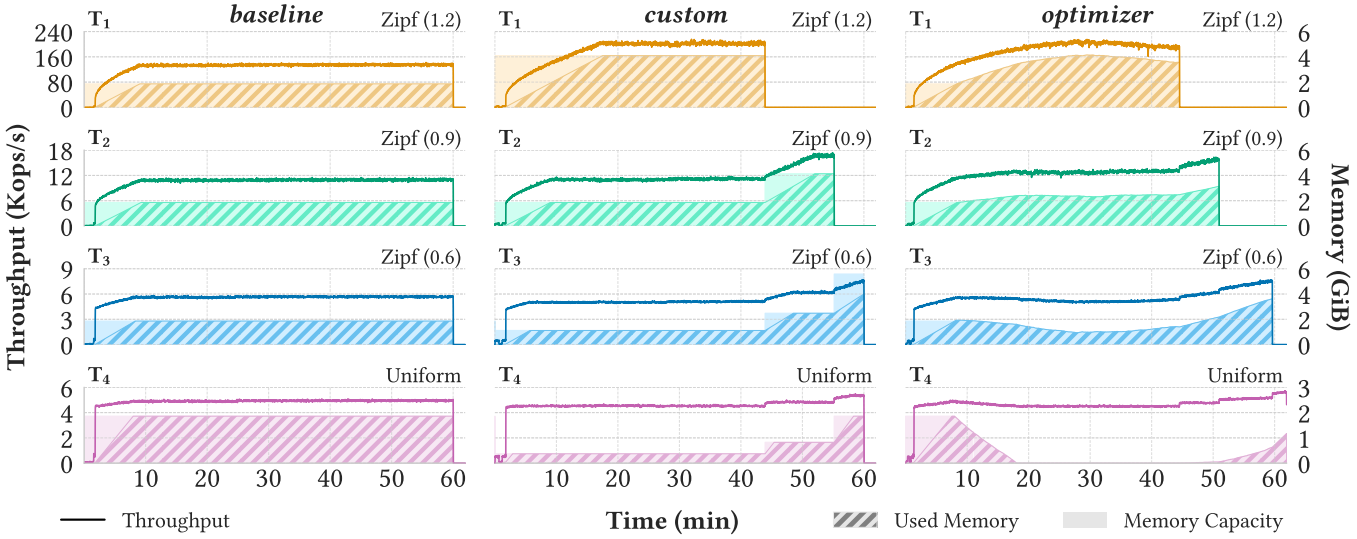



**Figure 2: Per-tenant performance of the *baseline*, *custom*, and *optimizer* setups (columns).** Rows depict tenants ($T_1$ to $T_4$). Left y-axis presents throughput, while the right y-axis presents the memory allocated (striped area) and the memory capacity (filled area).

## 3 Understanding CacheLib performance under shared environments

Despite CacheLib's widespread use, its behavior under evolving workloads and shared environments remains largely unexplored. This section presents an experimental study that characterizes the performance and resource efficiency of CacheLib's memory management mechanisms in multi-tenant and multi-instance settings.

**Hardware and OS configurations.** Experiments were conducted on a server equipped with 2× 64-core AMD EPYC 7742 processors, 256 GiB of memory, and a 480 GiB SSD, running RockyLinux 8.

**Methodology.** The experimental testbed consists of three components: *(i)* a benchmark that acts as the *application* and generates requests with varying access distributions, *(ii)* a CacheLib (v20240320) instance that caches the application's read requests, following a *write-around* caching policy, and *(iii)* using RocksDB (v6.15) as the persistent storage backend. Briefly, write operations are always submitted to RocksDB and, to avoid cache incoherence, propagated to the cache if the item is already cached. Read requests are first submitted to CacheLib – on a cache hit, the item is returned directly; on a cache miss, the application fetches the item from storage and inserts it into the corresponding memory pool. To isolate the performance benefits of CacheLib, both RocksDB's internal block cache and the OS page cache were disabled. For the multi-instance scenario, each *application* operates under a distinct CacheLib instance and RocksDB backend. We evaluate CacheLib's performance in a heterogeneous multi-tenant environment using 4 tenants in the single-instance setup and 4 instances in the multi-instance setup, each configured with different workload characteristics and memory limits.

**Workloads.** Experiments were conducted using read-only workloads, a common evaluation setup for *read-based caches*. In §5, we then consider production traces and synthetic workloads with distinct read:write ratios. Each tenant is configured with a distinct workload distribution: tenants $T_1$, $T_2$, and $T_3$ were assigned with *Zipfian* distributions with skew factors of 1.2, 0.9, and 0.6, respectively, while tenant $T_4$ was configured with a *uniform* distribution. Each tenant operated in an exclusive key space, each containing 20 million unique items of 1 KiB. The number of operations per tenant (or instance) is the same across all experiments. Moreover, for the multi-tenant setup, the CacheLib instance was configured to cache only up to 10% of the dataset (as in [32, 42]), corresponding to a total memory capacity of 8 GiB partitioned into four memory pools, one per tenant. The multi-instance setup followed the same workload and dataset configuration for instances $I_1$ to $I_4$, with the difference that each instance used a single memory pool of 2 GiB.

**Setups.** To explore the impact of different memory management strategies, we consider three CacheLib configurations: *baseline* refers to the default CacheLib; *custom* extends the baseline by integrating a memory allocation policy that enforces per-pool memory limits at different time periods (*i.e.*, dynamically) according to a static predefined configuration; and *optimizer* enables *pool resizer* and *pool optimizer* workers. Workers from both *custom* and *optimizer* execute every second; the *pool resizer* is configured with a resize strategy to optimize hit ratio.

### 3.1 Memory partitioning under single-instance, multi-tenant environments

We begin by evaluating the performance and adaptability of different CacheLib mechanisms under a single-instance, multi-tenant environment. In these experiments, all tenants execute their workloads concurrently, under three distinct memory allocation configurations. In the *baseline* setup, all memory pools are statically configured with 2 GiB each. The *custom* setup allocates more memory to tenants with higher *zipfian* skew, enforcing a predefined memory allocation ratio of 55:25:15:5 for $T_1$ to $T_4$, applied only to active tenants (*e.g.*, if $T_1$ is inactive, the remaining tenants receive the leftover memory proportionally to their ratios). Finally, the

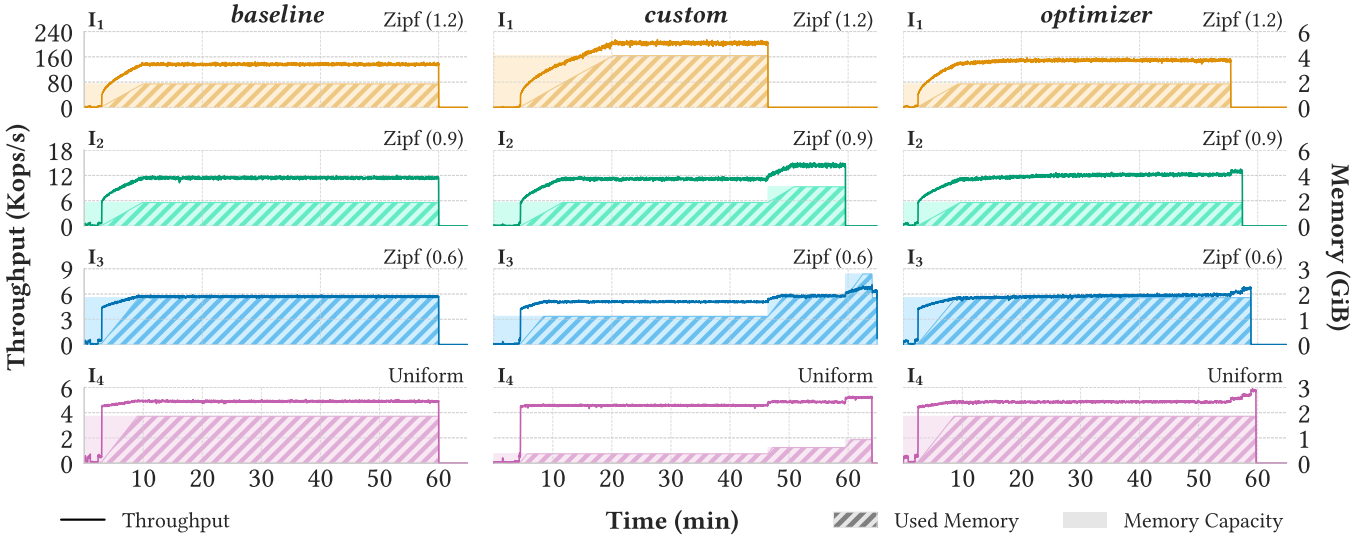



**Figure 3: Per-instance performance of the *baseline*, *custom*, and *optimizer* setups (columns).** Rows depict instances ($I_1$ to $I_4$). Left y-axis presents throughput, while the right y-axis presents the memory allocated (striped area) and the memory capacity (filled area).

*optimizer* starts with the same uniform memory allocation as the *baseline* but dynamically adjusts memory distribution throughout its execution (§2.1), aiming to maximize overall throughput.

**Results.** Fig. 2 illustrates the per-tenant throughput and allocated memory under the three CacheLib configurations over time. We observe that, despite the uniform memory allocation of the *baseline* setup, throughput varies significantly across tenants due to their varying workload skewness. Particularly, in the *baseline* setup, after the initial cache warm-up, $T_1$ reaches 142 kops/s, outperforming $T_2$, $T_3$, and $T_4$ by 12×, 23×, and 28×, respectively.

The *custom* setup shows that prioritizing memory allocation for highly skewed workloads yields significant performance gains. As depicted in Fig. 2, $T_1$'s throughput increases up to 1.6× compared to the *baseline* (≈230 kops/s), but requires 2.2× more memory (*i.e.,* 4.4 GiB *vs.* 2 GiB). While the throughput gain is smaller than the proportional memory increase, both $T_1$ and the overall system achieve the highest absolute throughput under this allocation strategy. This non-linear improvement arises because, as workload access skew intensifies, achieving further improvements in cache hit ratio requires increasingly larger memory allocations. However, since backend I/O requests decrease inversely with the hit ratio, even modest improvements in hit ratio can yield substantial throughput gains. Conversely, tenants $T_3$ and $T_4$, which exhibit lower workload skew, show minimal deviation from the *baseline* performance during the majority of their execution, while their memory allocations decreased by 0.6× and 0.2×, respectively. Once $T_1$ completes its workload, however, its released memory is redistributed to the remaining tenants, increasing $T_2$'s performance by up to 1.5× (between minutes 43 and 55).

The *optimizer* setup begins with the same uniform memory distribution as the *baseline* but gradually reallocates memory based on observed workload behavior. After the warm-up period, the *optimizer* improves $T_1$ and $T_2$ performance of up to 1.5× (214 kops/s) and 1.4× (14 kops/s), respectively, compared to the *baseline*. Indeed, the *custom* and *optimizer* setups outperformed the *baseline* configuration overall throughput by 1.44× (236 kops/s) and 1.45× (237 kops/s) under the same memory capacity.

> **Observation 1.** *Under heterogeneous workloads, uniform memory allocation policies lead to severe performance imbalances. Dynamically reallocating memory based on workload characteristics can outperform static configurations, even under stable workloads and identical memory constraints.*

Interestingly, a closer inspection reveals that the *optimizer* reallocates nearly all $T_4$'s memory to the other tenants, resulting in starvation and a 5% increase in $T_4$'s execution time compared to the *baseline*. While such behavior might be suitable for environments where maximizing the global throughput is the primary goal, it is otherwise problematic in scenarios that require per-tenant quality-of-service (QoS) guarantees. In these cases, even if global throughput improves, the *optimizer*'s greedy reallocation strategy may create severe performance contention for some tenants.

> **Observation 2.** *In the absence of QoS or prioritization mechanisms, CacheLib's dynamic reallocation strategy can lead to unfair memory distribution, compromising performance isolation among tenants.*

## 3.2 Memory partitioning under multi-instance environments

We now evaluate CacheLib's memory management strategies when multiple cache instances operate concurrently. The experiments include four CacheLib instances ($I_1$ to $I_4$), each serving a single tenant, under different memory allocation configurations. The memory allocation strategy of all setups was similar to §3.1.

**Results.** Fig. 3 depicts the per-instance throughput and memory allocation under the *baseline*, *custom*, and *optimizer* setups. Unsurprisingly, the *baseline* behaves similarly as in §3.1, with $I_1$ dominating the overall throughput with an average of 145 kops/s, while the remainder instances achieve up to 12 kops/s. This performance imbalance stems from the uniform memory allocation strategy and the different workload skewness, as observed in §3.1. Moreover, the *optimizer* setup experiences a similar behavior as the *baseline*, as each instance operates in isolation, preventing the *pool optimizer* from redistributing memory across instances. Indeed, some instances completed their workloads earlier than in the *baseline*, but due only to LRU-2Q, *optimizer*'s eviction policy, which is more effective than LRU for these workloads.

In contrast, the *custom* setup shows that dynamically reallocating memory across instances, while prioritizing those with higher workload skew, yields significant performance gains. As depicted in Fig. 3, $I_1$'s throughput increases up to 1.5× compared to the *baseline* (≈213 kops/s), primarily by reclaiming memory from $I_3$ and $I_4$. Once $I_1$ completes its execution, the custom allocation policy redistributes the memory of the remaining active instances proportionally to their ratios and workload skew. However, as instances complete and their memory is released back to the OS (*e.g.,* after $I_1$ and $I_2$ finish, the effective memory capacity is halved), maintaining the performance of the remaining instances becomes harder, particularly for low skew instances that have previously shared memory; these instances can experience longer execution times than the *baseline*.

> **Observation 3.** *Under multi-instance scenarios, isolated memory management limits optimization opportunities, leading to suboptimal resource utilization and performance. Instead, coordinated memory reallocation across instances can significantly improve overall throughput and decrease execution time.*

## 4 Holpaca: Holistic and Adaptable Caching

Building on the observation that static and uncoordinated memory allocation leads to fragmented resource usage and unbalanced memory distribution (§3), we propose Holpaca, a general-purpose caching middleware that improves the performance and resource efficiency of shared caching environments, in multi-tenant and multi-instance scenarios.

Fig. 4 depicts the high-level architecture of Holpaca. It extends CacheLib's design by adopting ideas from the software-defined paradigm [27, 29], decoupling the memory management mechanisms from the policies that govern them into two main components. The *data layer* (§4.1) acts as a shim middleware that sits between the application and the underlying caching system. It is co-located with each CacheLib instance (forming an *agent*) and is responsible for collecting fine-grained runtime metrics (*e.g.,* hit ratios, miss ratio curves) and dynamically adjusting the CacheLib's internal mechanisms to redistribute memory across pools. The *orchestrator* (§4.2) is a centralized controller that manages all *data layer agents* to make holistic memory allocation decisions across tenants and instances, maximizing overall cache performance and resource usage.

Next, we detail the key components and techniques that underpin its design (§4.1-§4.2), introduce two memory management strategies designed for shared cache environments (§4.3), and discuss implementation details (§4.4).

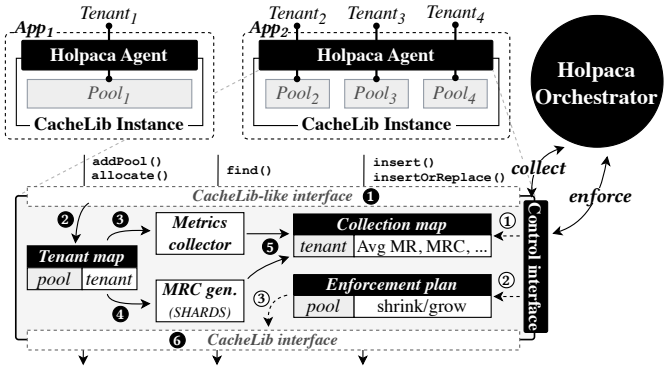



**Figure 4:** Holpaca high-level architecture.

### 4.1 Data layer

Holpaca's *data layer* is a multi-agent component that operates as a middleware between the application and the underlying caching system. It continuously monitors cache activity and adjusts its configurations to enforce informed, system-wide memory allocation decisions. As shown in Fig. 4, each agent operates within a single CacheLib instance and manages the operations of multiple tenants. Agents expose a CacheLib-compliant API, transparently intercepting cache requests while maintaining full compatibility with existing applications. To handle cache operations, agents are organized into three main components.

**Tenant differentiation and monitoring.** We assume a scenario where the system designer configures the CacheLib instance, setting the number of memory pools and their capacity that fit the performance requirements of the application. Each agent intercepts incoming cache requests (❶) and identifies the corresponding memory pool they are targeted to (❷). During this process, the agent collects runtime metrics that will enable the orchestrator to generate memory allocation decisions, including per-instance and per-tenant throughput, hit/miss ratio, memory usage, and disk operations (❸). Once the metrics are recorded, requests are forwarded to the corresponding memory pool for execution (❻).

**MRC estimation.** In practice, workloads exhibit dynamic characteristics that evolve over time (*e.g.,* workload skew, read/write ratio), and thus, the cache miss ratio depends strongly on which subset of data resides in the cache [24, 43]. To capture this behavior, Holpaca uses *spatially hashed approximate reuse distance sampling* (SHARDS) to compute *miss-ratio curves (MRCs)* that describe how the cache miss ratio changes for different pool sizes per tenant [38]. Specifically, each agent intercepts every `find` and `insert` request to the cache and records these with SHARDS (❹), which applies randomized spatial sampling (in our case, 1 every 1000 for each recorded entry) to generate accurate MRC curves (❺). These curves are then periodically sent to the orchestrator, which uses them to determine each pool's memory size configuration according to the optimization goal.

**Memory pool resizing.** Agents expose a simple control interface that allows the *orchestrator* to query collected runtime metrics

and MRC estimations (①), and to enforce memory allocation rules according to the user-defined objectives (②). These rules instruct agents to shrink or expand the size of individual memory pools (③), ensuring that each pool maintains an appropriate capacity for the observed workload while maximizing cache efficiency.

## 4.2 Orchestrator

Holpaca's *orchestrator* is a centralized component with system-wide visibility responsible for managing memory distribution both within and across CacheLib instances. It does so by periodically communicating with the *data layer agents* to collect cache-related metrics and enforce memory allocation decisions to ensure the user-defined objectives are met. As depicted in Fig. 4, the orchestrator is decoupled from the data layer, which enables the separation of the control logic from the actual memory building blocks.

**Managing tenants and instances.** The orchestrator maintains a hierarchical mapping of memory pools. Specifically, the caching space consists of multiple instances, each instance contains multiple tenants, and each tenant operates on a distinct memory pool. The orchestrator maintains a registry of all active agents, along with their per-instance and per-pool memory capacities. Whenever a new agent (*i.e.,* a new CacheLib instance) is created, it registers itself with the orchestrator, providing its identifier, number of pools, and associated memory capacities. Conversely, when an agent disconnects (*e.g.,* due to completion or failure), it is removed from the registry, and the total memory capacity is updated accordingly, as the instance's allocation is released back to the OS. If the orchestrator fails, the system remains under a static memory allocation configuration (*i.e.*, using the latest values enforced by the orchestrator), until a new orchestrator is respawned.

**Coordinated cache management.** Holpaca's memory allocation strategy is specified in a *feedback control loop* logic, where the orchestrator repeatedly performs four main steps.

*Collect:* The orchestrator gathers runtime metrics from all agents, including both per-tenant and per-instance throughput, MRC estimations, memory utilization, backend IOPS, among others.

*Compute:* These metrics are fed into *optimization algorithms* (§4.3) that determine new memory allocations for each pool. These algorithms are independent of the underlying caching engine, allowing system designers to specify different performance goals to achieve across the overall caching space, such as maximizing global throughput or maintaining per-tenant/per-instance QoS guarantees.

*Enforce:* Once the new allocation decisions are computed, the orchestrator disseminates them to the corresponding agent, which then resizes its memory pools accordingly.

*Sleep:* Finally, the orchestrator waits for a configurable interval before starting a new control cycle (*e.g.,* perform the aforementioned control steps at 1-second intervals). With short intervals, agents are adjusted more frequently but impose higher control overhead (*i.e.,* frequent statistic collection and memory reconfiguration), while larger intervals reduce overhead but agents become unsupervised for longer periods, which under volatile workloads, can lead to outdated memory allocations and suboptimal performance. The interval is user-configurable according to the requirements of each use case; we defer automatic interval tuning to future work.

By maintaining centralized control with system-wide visibility, the orchestrator enables holistic and coordinated memory allocation decisions across both tenants and instances.

## 4.3 Optimization algorithms

To demonstrate the performance and feasibility of Holpaca, we implemented two memory management algorithms for shared cache environments, each tailored for distinct optimization goals. These algorithms extend prior work [26] to support low latency and fine-grained memory (re)allocations over multi-tenant and multi-instance caching environments. Next, we describe each algorithm in detail. For clarity, the discussion focuses on redistributing memory between tenants within a given instance, but the same logic generalizes to multi-instance setups, as demonstrated in §5.

**Global throughput maximization.** The first algorithm aims to maximize the overall throughput of a CacheLib instance (*i.e.,* aggregated throughput of all tenants) by dynamically reclaiming memory from low-impact tenants (*i.e.*, those who contribute minimally to overall throughput, regardless of their allocated memory) and reallocating it to the high-impact ones (*i.e.,* those who contribute the most to global cache throughput).

The orchestrator begins by collecting runtime metrics from all active tenants, including their MRC estimates, current memory allocations, backend IOPS, and hit/miss ratios. For all new tenants (*i.e.*, those recently registered for which no performance history exists), the algorithm allocates memory by evenly reclaiming a small fraction from active tenants, proportional to $\frac{\text{cache size}}{\#\text{tenants}}$. For the remainder tenants, the algorithm first converts each tenant's MRC into throughput estimation curve, using $\text{IOPS} = \frac{\text{IOPS}_{\text{storage}}}{\text{miss ratio}}$, where $\text{IOPS}_{\text{storage}}$ represents the average IOPS of the underlying storage backend. Due to the discrete nature of the estimated MRCs (and consequently, the throughput curves), we use spline interpolation to obtain smooth, continuous functions for each tenant, enabling more accurate optimization. Furthermore, the algorithm employs simulated annealing, a probabilistic optimization technique, to explore thousands of memory distribution configurations and selects the one that yields the highest aggregated throughput. However, this technique is sensitive to the configured parameters and the accuracy of the estimated throughput curve, which can lead to deviations from the optimal solution. To mitigate this, we restrict the maximum memory that can be reallocated per control loop iteration to a configurable percentage of the total cache memory (default: 1%). This prevents large, abrupt memory reallocations and ensures smooth, feedback-driven adaptation across control cycles.

**QoS enforcement with throughput maximization.** The second algorithm extends the previous one to incorporate per-tenant QoS guarantees, specified by the system designer during tenant configuration, where each tenant has a minimum throughput target that must be met. Specifically, the algorithm evaluates all candidate memory allocations but discards those that would violate any tenant's QoS constraint. In practice, this means that tenants whose measured throughput falls below their defined threshold are excluded from memory reduction decisions. Instead, the algorithm reallocates memory preferentially from tenants operating above

their targets, prioritizing overall throughput while ensuring performance isolation for critical workloads.

## 4.4 Implementation

We have implemented Holpaca's *data layer* and *orchestrator* with approximately 1.5K lines of C++ code, which are publicly available at **https://github.com/dsrhaslab/Holpaca**.

**Transparently intercepting CacheLib operations.** To minimize intrusiveness and ensure general applicability, the data layer extends CacheLib's *CacheAllocator* class. Specifically, we reimplemented the methods `addPool`, `insert`, `insertOrReplace`, and `find`. The `addPool` operation is responsible for creating memory pools and was extended to register each pool identifier with the orchestrator during agent initialization and the support of an optional QoS parameter to enable applications to specify minimum throughput guarantees for each tenant. The remaining operations, responsible for fetching and inserting data into the cache, were extended to feed SHARDS to generate per-tenant and per-instance MRC curves, and to ensure tenant differentiation at runtime by extracting the pool identifier from internal structures associated with each operation.

**Communication.** Communication between the data layer and the orchestrator is implemented through RPC calls, using the gRPC framework [2]. The communication interface consists of four main operations: `register` and `leave`, used by agents to joint or detach from the orchestrator, respectively; `stats`, through which the orchestrator periodically collects runtime metrics from each agent; and `resize`, used to enforce new memory allocation decisions across tenants and instances.

## 4.5 Discussion

**Applicability to distributed environments.** While this paper focuses on managing memory across co-located cache instances within a single compute node, Holpaca's design generalizes to distributed deployments in which agents operate across different nodes. In large-scale environments, however, with hundreds to thousands of cache instances, the centralized orchestrator may become a scalability bottleneck [31]. We defer exploring the scalability of Holpaca's orchestrator to future work.

**Accuracy of control decisions.** Holpaca estimates performance under different memory allocations by *(i)* approximating MRCs using SHARDS and *(ii)* computing a near-optimal cache partitioning via simulated annealing (SA). Consequently, control decisions depend on workload observability (assuming non-adversarial tenants), the accuracy of MRC estimations, and quality of the optimization result. Since SHARDS and SA trade accuracy for computational efficiency (§5.4) and are sensitive to parameter tuning, the estimations may not accurately address highly dynamic workloads. To mitigate this, the orchestrator recomputes estimations using new collected metrics at each control interval. Further, Holpaca enables system designers to integrate alternative estimation or optimization techniques without modifying the overall workflow.

**Generalizability across caching engines.** While Holpaca is implemented on top of CacheLib, its design is not inherently tied to a specific caching engine. The *orchestrator* directly coordinates Holpaca's agents, not depending on specific cache internals, making the control logic generally applicable across engines. The *data layer*, however, assumes that caching instances can shrink or expand their memory allocations. Such elasticity is commonly supported by production-ready caching engines (*e.g.,* Memcached [1], Redis [5]), making Holpaca applicable to other caching engines.

# 5 Evaluation

Our evaluation aims to answer the following questions:

- Can Holpaca improve cache management under multi-tenant (§5.1) and multi-instance scenarios (§5.2)?
- Can Holpaca enforce per-instance QoS objectives while maximizing overall cache throughput (§5.3)?
- What is the overhead of Holpaca's components (§5.4–§5.5)?

**Experimental setup and methodology.** We used the same hardware, OS configuration, and methodology as described in §3. All reported results correspond to the average of 5 runs, with standard deviation remaining below 5% in all cases.

**Workloads.** We used four production traces from Twitter [43]: *cluster19* (2.97M, 97:1, ≈2.1), *cluster40* (5.71M, 45:1, ≈1.2), *cluster53* (10.94M, 1:1, ≈0.9), and *cluster18* (27.53M, 3:1, ≈0.7), each with different working set sizes (number of unique keys), read:write ratios, and zipfian skews (shown in parentheses). Due to the size of the traces, we trimmed each to the first 270 M requests, with all value sizes fixed at 1 KiB, avoiding slab calcification effects (an orthogonal problem to this paper). For the multi-tenant scenario (§5.1), traces were respectively assigned to tenants $T_1$ to $T_4$, and to instances $I_1$ to $I_4$ for the multi-instance scenario (§5.2).

**Setups.** We evaluated and compared CacheLib under three configurations: *baseline*, *optimizer*, and *holpaca*. The *baseline* and *optimizer* follow the same configurations as in §3, while *holpaca* corresponds to CacheLib integrated with our system (configured with a 1-second interval control periods). In *optimizer*, workers execute every second, and the *pool resizer* is configured to maximize hit ratio. We do not consider the *custom* setup from §3, as it represents a manually tuned scenario designed for synthetic and stable workloads, and to highlight CacheLib's limitations.

## 5.1 Single instance with multiple tenants

We begin by evaluating the impact of Holpaca in a single caching instance shared by multiple tenants. Tenants start at the same time and stop once all operations have been processed. For this experiment, *holpaca* was configured with the *global throughput maximization* memory allocation strategy (§4.3). Fig. 5 depicts the per-tenant throughput and allocated memory results of all setups.

**Results.** Holpaca achieves the highest per-tenant and overall throughput among all setups. As depicted in Fig. 5, it significantly improves the performance of high skew tenants (*i.e.,* $T_1$ and $T_2$), outperforming *baseline* and the *optimizer* setups up to 2.5× and 1.55×, respectively. At the same time, it ensures that tenants with low skew ($T_3$ and $T_4$) do not suffer from starvation and complete execution in a similar time as the other setups, as leftover memory is dynamically reallocated when higher-impact tenants finish.

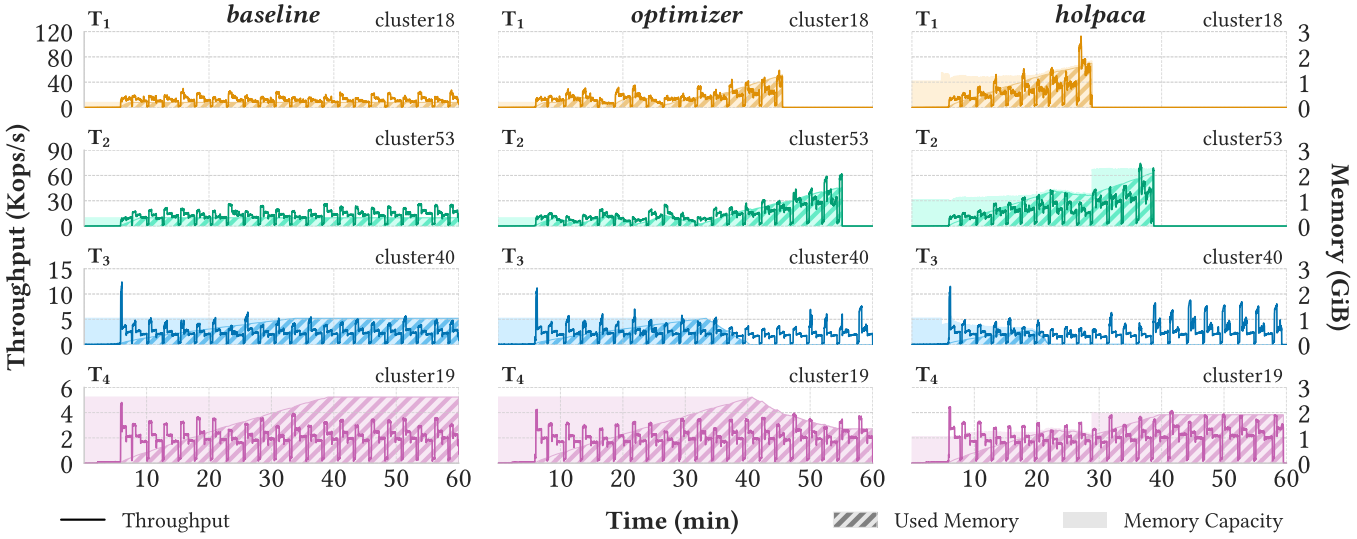



**Figure 5: Per-tenant performance of the *baseline*, *optimizer*, and *holpaca* setups (columns).** Rows depict tenants ($T_1$ to $T_4$). Left y-axis presents throughput, while the right y-axis presents the memory allocated (striped area) and the memory capacity (filled area).

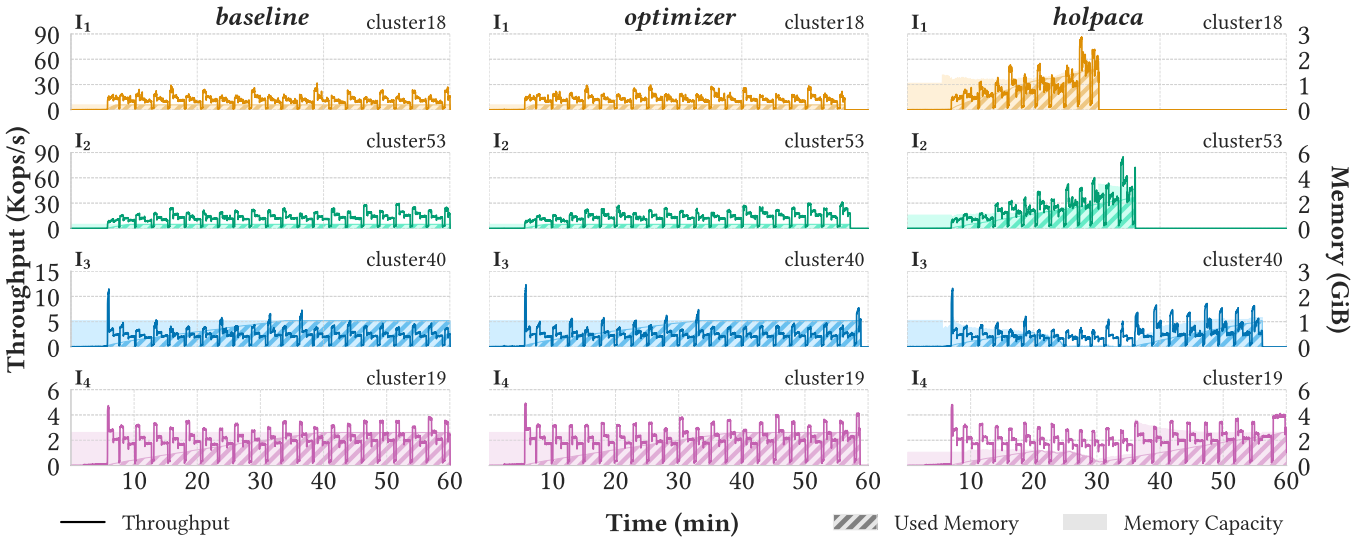



**Figure 6: Per-instance performance of the *baseline*, *optimizer*, and *holpaca* setups (columns).** Rows depict tenants ($T_1$ to $T_4$). Left y-axis presents throughput, while the right y-axis presents the memory allocated (striped area) and the memory capacity (filled area).

These improvements stem from Holpaca's memory allocation strategy (§4.3), which prioritizes tenants contributing most to global performance ($T_1$ and $T_2$) by dynamically moving memory from lower-impact tenants ($T_3$ and $T_4$). In contrast, the *baseline* employs static memory allocations, where memory pools are configured with 10% of their respective working set size and do not change during execution. For the *optimizer*, the reason behind this performance difference is twofold. First, its memory reallocation process relies on tracking hits at the tail of the eviction queue (§2.1), which delays memory allocation decisions until enough metrics are collected. Conversely, Holpaca estimates MRCs online using SHARDS, making memory reallocation decisions much sooner. Second, there are several periods where the *optimizer* allocates memory inefficiently, overprovisioning low skew tenants ($T_4$ between minutes 35 and 55) at the expense of high skew ones ($T_1$ and $T_2$), resulting in inefficient memory usage and suboptimal throughput.

## 5.2 Multiple instances

We now evaluate Holpaca's performance under multi-instance caching environments, where each instance ($I_1$ to $I_4$) serves a single tenant. Instances start at the same time and stop after all operations have been processed. Fig. 6 depicts the per-instance throughput and memory allocation for all setups.

**Results.** Similarly to §5.1, Holpaca outperforms both *baseline* and *optimizer* configurations. As shown in Fig. 6, it improves $I_1$ and $I_2$ throughput by up to 2.71×, 2.88×. This performance difference

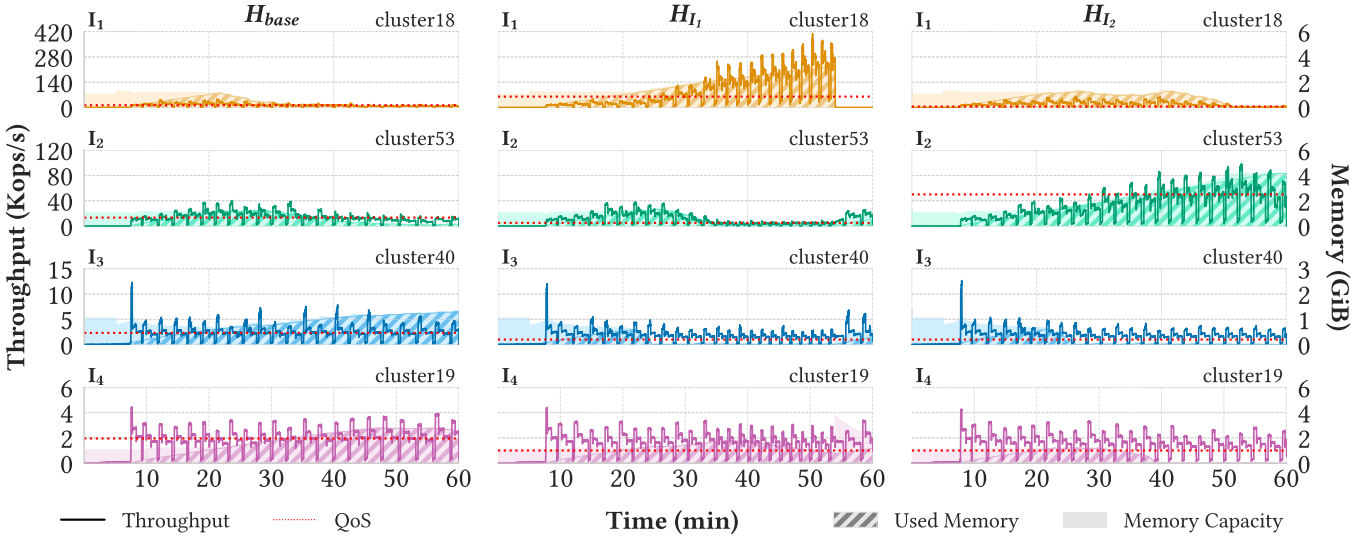



**Figure 7: Per-instance performance of Holpaca under $H_{base}$, $H_{I_1}$, and $H_{I_2}$ QoS configurations (columns).** The red line depicts the minimum throughput guarantee defined for each instance.

is caused by the inability of both *baseline* and *optimizer* setups to make memory allocation decisions that consider all active instances, leading to resource underutilization and suboptimal throughput. Contrarily, Holpaca follows a holistic memory allocation strategy that ensures memory is redistributed across instances according to their impact on the global throughput, highlighting the importance of an adaptive and holistic cache management design. Moreover, the *optimizer* setup performs slightly better than *baseline*, due to its use of LRU-2Q eviction policy compared to the *baseline*'s LRU.

## 5.3 Per-instance QoS guarantees

We now evaluate Holpaca's ability to enforce minimum per-instance throughput guarantees while maximizing global throughput.

**Setups.** We configured *holpaca* to run the *QoS enforcement with throughput maximization* algorithm (§4.3), under three configurations, each with distinct per-instance minimum throughput guarantees: $H_{base}$ ensures that all instances meet a minimum throughput equal to the average achieved in §5.2's *baseline*; $H_{I_1}$ prioritizes $I_1$ over the other instances; and $H_{I_2}$ prioritizes $I_2$ instead. We omit both *baseline* and *optimizer* setups, as neither provides QoS support.

**Results.** Fig. 7 shows the per-instance throughput and allocated memory for the three aforementioned configurations. The red dotted lines present each instance's minimum throughput goal. Across all scenarios, Holpaca successfully enforces the specified QoS guarantees, apart from two exceptions: *(i)* during the initial cache warm-up, when not enough data is cached to reach the performance goal, and *(ii)* when the workload intensity is too low to achieve the target. The $H_{base}$ setup serves as a reference point, demonstrating that Holpaca consistently meets all QoS objectives, and the presence of minimum performance guarantees leads the algorithm to adopt a more conservative memory redistribution strategy. In particular, it moves memory from low-skew instances to high-skew ones (as observed in the previous testing scenarios), while still reserving enough memory for each instance to meet its target. For $H_{I_1}$, Holpaca prioritizes $I_1$, achieving a peak performance exceeding 400 kops/s. After the initial cache warm-up (minutes 10 to 35), the algorithm gradually redistributes memory from instances $I_2$ to $I_4$, ordered by the instance that benefits the least from its current allocated memory (*i.e.,* low impact in global throughput), stopping just before any instance would fall below its QoS target. For $H_{I_2}$, we draw similar observations as in $H_{I_2}$, but prioritizing $I_2$ instead.

## 5.4 Overhead of Holpaca's *data layer*

We now measure the performance of Holpaca's *data layer*.

**Methodology.** To isolate cache performance, CacheLib was configured without a storage backend. Before each run, we preload the cache with the entire dataset to avoid misses. Experiments were executed with an increasing number of concurrent tenants and instances (from 1 to 32 in both scenarios).

**Workloads.** Each tenant/instance accesses an exclusive key-space of 2 M unique keys, each storing a 1 KiB value. We consider three workloads with distinct read:write ratios: read-only (1:0), mixed (1:1), and write-heavy (1:9). Each workload runs for 10 minutes.

**Setups.** We evaluate two setups: *baseline*, which corresponds to the default CacheLib; and *holpaca*, where the caching instance integrates an *agent*. To isolate the overhead of the agent itself, Holpaca is executed without the *orchestrator*.

**Results.** Fig. 8 depicts the average throughput and latency of both *baseline* and *holpaca* setups. Across all scenarios, *holpaca* closely follows the *baseline*'s performance curve, achieving similar throughput performance and introducing minimal latency overhead. Specifically, under multiple tenants (solid lines), *holpaca* adds an average latency overhead of 0.14 ms, while for multiple instances (dashed lines), it introduces an average latency of 0.04 ms. This overhead is primarily caused by the MRC updates and additional retrieved metadata information from CacheLib's internal structures, both of which occur in the critical path of cache operations. Moreover,

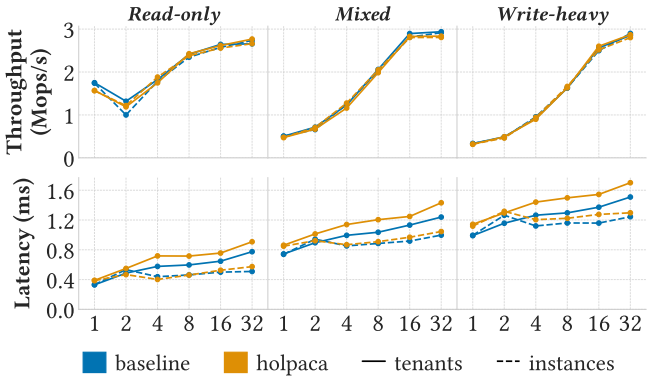



**Figure 8: Throughput and latency across *baseline* and *holpaca* setups** under multiple tenants (solid lines) and instances (dashed lines), with *read-only*, *mixed*, and *write-heavy* workloads.

because baseline CacheLib instances experience lower latency than multi-tenant workloads (due to concurrency control overhead when multiple tenants share the same instance), Holpaca's additional overhead becomes more noticeable in these scenarios.

## 5.5 Performance of Holpaca's *orchestrator*

We now evaluate the scalability of the Holpaca's *orchestrator*, focusing on two aspects: *(i)* the latency breakdown across the different steps of the control loop, and *(ii)* the CPU usage of the *orchestrator* under varying control intervals.

**Methodology.** We measured the performance of the interaction between the *orchestrator* and *data layer* by executing 1000 iterations of the main steps of the feedback control loop. The compute phase executes the *global throughput maximization* algorithm (§4.3). We consider a multi-instance scenario, where we vary the number of instances (from 1 to 32). To isolate the cost of the *orchestrator* operations, the data items within each instance remain constant throughout the experiment, and no operations are issued.

**Results.** Table 1 presents the latency breakdown of the *orchestrator*'s control loop with an increasing number of instances at a fixed control interval (1-second). The total latency increases with the number of controlled instances, ranging from ≈62 ms to ≈430 ms. This increase is expected, as adding more instances enlarges both the optimization space and the amount of metadata processed at each iteration. Nevertheless, typical production deployments use intervals on the order of seconds, as workload, although dynamic, generally exhibit stable periods [11, 32]. For example, when configured with 1-second control intervals (§5.1–§5.3), the total latency remains below the control period even at 32 instances (≈430 ms), ensuring that each control cycle completes within a single interval. Most of this latency is spent in the *compute* phase, due to the cost of analyzing the per-instance MRC estimations and conducting the simulated annealing process to search for optimal memory allocations. Additionally, as the number of controlled instances increases, the cost of *collecting* statistics also grows, since more information must be transferred to the orchestrator. In contrast, the *enforce* phase contributes minimally to the overall control loop latency.

Table 2 depicts the CPU overhead of the *orchestrator* when configured with different control intervals, ranging from 1 ms to 10s.

**Table 1:** Latency breakdown of Holpaca's orchestrator control logic with increasing number of instances (1 to 32).

| Instances | Total | Collect | Compute | Enforce |
|---|---|---|---|---|
| 1 | 61.76 ms | 2.1% | 97.4% | 0.5% |
| 2 | 77.43 ms | 3% | 96.6% | 0.4% |
| 4 | 97.93 ms | 4.3% | 95.3% | 0.4% |
| 8 | 133.37 ms | 6% | 93.7% | 0.3% |
| 16 | 206.33 ms | 7.8% | 91.9% | 0.3% |
| 32 | 429.85 ms | 8.1% | 91.8% | 0.1% |

**Table 2:** CPU usage of Holpaca's orchestrator with increasing number of instances (1 to 32) and control intervals.

| Instances | 1ms | 10ms | 100ms | 1s | 10s |
|---|---|---|---|---|---|
| 1 | 65% | 55% | 36% | 15% | 12% |
| 2 | 86% | 79% | 44% | 20% | 15% |
| 4 | 85% | 79% | 50% | 21% | 15% |
| 8 | 84% | 78% | 55% | 23% | 15% |
| 16 | 81% | 78% | 60% | 27% | 16% |
| 32 | 81% | 80% | 70% | 38% | 17% |

CPU usage increases along two dimensions: *(1)* with the number of instances, since simulated annealing must explore a larger allocation space; and *(2)* when the control period decreases, as the orchestrator will execute its control logic more frequently.

# 6 Related work

This section describes prior work and places our work in context.

**Improving cache internals.** A significant body of research has focused on optimizing cache internals to improve performance and memory efficiency. Systems such as MICA [28], MemC3 [21], SegCache [44], and FrozenHot [32] reduce data-access latency by optimizing data structures, metadata management, and concurrency control. Other works, including zExpander [42] and Kangaroo [30], reduce memory footprint by compressing or offloading cached data to secondary storage. On the other hand, Holpaca focuses on shared cache environments, an aspect not covered by the aforementioned systems. Importantly, Holpaca is complementary to these solutions and could be combined with them to further improve performance under shared scenarios.

**Memory allocation and resource management.** Several works have explored cache memory allocation strategies, though many rely on simplified deployment assumptions or target specific scenarios. Systems such as DynaCache [17], Cliffhanger [18], and LAMA [24] improve memory utilization across slab classes by dynamically resizing them; however, they do not consider multi-tenancy caching scenarios. In contrast, Holpaca dynamically redistributes memory across distinct tenants as they compete for shared cache resources. Multi-tenant cache management solutions, including Memshare [19], mPart [15], BLAZE [16], AdaptCache [10], and various cache-as-a-service systems [7], treat caches as a single shared resource among multiple tenants. This is problematic because, as shown in Fig. 8, multiple tenants sharing a single caching instance can create contention, leading to performance slowdowns. Lastly, MIMIR [33], HPUCache [39], and SPREDS [8] improve data partitioning in distributed caches such as Redis [5]

and Memcached [1]. While our work addresses similar challenges (*i.e.,* performance estimation as a function of cache size, partitioning, coordinated cache management), the underlying system model differs fundamentally, as distributed caches expose a single logical keyspace spanning multiple nodes, whereas CacheLib and Holpaca provide multiple independent cache instances within a node. Thus, Holpaca is the first to support multiple independent cache instances used by different applications or services, and to coordinate memory allocation across them. Further, these works typically focus on maximizing hit ratio, which does not necessarily translate into proportional improvements in application-level performance.

**Performance optimization and QoS guarantees.** Another body of work explores dynamic resource management for multi-tenant caches with explicit performance objectives. Systems like Pisces [34] and RobinHood [13] improve fairness across tenants by dynamically adjusting per-tenant cache partitions. Centaur [26] and Moirai [36] leverage per-tenant MRCs to guide memory reallocations and enforce QoS guarantees. These approaches, however, focus exclusively on single-instance scenarios (either local or remote caches), monitoring tenant performance and coordinating memory redistribution within a single cache instance. In contrast, Holpaca coordinates memory allocation across multiple cache instances, while still meeting per-tenant or per-application performance targets.

## 7 Conclusion

We have presented Holpaca, a general-purpose caching middleware that coordinates memory management across multiple tenants and cache instances. By combining online MRC estimation with a centralized orchestrator, Holpaca enables adaptive, globally informed memory allocation decisions, maximizing performance and resource efficiency. Our extensive evaluation shows that Holpaca consistently outperforms CacheLib's default and auto-tuning configurations over shared environments.

## Acknowledgements

We thank the anonymous reviewers for their comments and suggestions. This work was co-funded by national funds through FCT - Fundação para a Ciência e a Tecnologia, I.P., under the support UID/50014/2025 (https://doi.org/10.54499/UID/50014/2025) (Pedro Peixoto), with the PhD grant 2025.01528.BD, on the scope of the project *"Protocolo Operação Supercomputador Deucalion"* funded by Portuguese Foundation for Science and Technology I.P. (Cláudia Brito), by FCCN within the scope of the project Deucalion (2024.00014.TEST. DEUCALION), and by the European Regional Development Fund through the NORTE 2030 Regional Programme under Portugal 2030, within the scope of the project BCDSM, reference 14436 (NORTE2030-FEDER-00584600) (João Paulo). This work was also supported in part by the National Science Foundation under award numbers 2323100, 2338457, 2402328, and CNS-1956229, as well as generous donations from NetApp and Seagate.

## References

[1] 2009. *Memcached.* https://memcached.org/
[2] 2015. *gRPC: A high performance, open source univeral RPC framework.* https://grpc.io/
[3] 2021. *CacheLib: Facebook's open source caching engine.* https://engineering.fb.com/2021/09/02/open-source/cachelib/
[4] 2021. *Pelikan Cache: A framework for building local and distributed caches.* https://github.com/pelikan-io/pelikan
[5] 2023. *Redis.* https://redis.io
[6] 2024. *Feature Caching for Recommender Systems with CacheLib.* https://medium.com/pinterest-engineering/feature-caching-for-recommender-systems-w-cachelib-8fb7bacc2762
[7] n.d.. *MemCachier.* https://www.memcachier.com/
[8] Xavier Andrade, Jorge Cedeno, Edwin Boza, Harold Aragon, Cristina Abad, and Jorge Murillo. 2019. Optimizing Cloud Caches For Free: A Case for Autonomic Systems with a Serverless Computing Approach. In *2019 IEEE 4th International Workshops on Foundations and Applications of Self* Systems (FAS*W).*
[9] Nikos Armenatzoglou et al. 2022. Amazon Redshift Re-invented. In *2022 International Conference on Management of Data.*
[10] Omar Asad and Bettina Kemme. 2016. AdaptCache: Adaptive Data Partitioning and Migration for Distributed Object Caches. In *Proceedings of the 17th International Middleware Conference.*
[11] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. 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.*
[12] Benjamin Berg, Daniel S. Berger, Sara McAllister, et al. 2020. The CacheLib Caching Engine: Design and Experiences at Scale. In *14th USENIX Symposium on Operating Systems Design and Implementation.*
[13] Daniel S. Berger, Benjamin Berg, Timothy Zhu, Siddhartha Sen, and Mor Harchol-Balter. 2018. RobinHood: Tail Latency Aware Caching - Dynamic Reallocation from Cache-Rich to Cache-Poor. In *13th USENIX Symposium on Operating Systems Design and Implementation.*
[14] Timm Böttger et al. 2018. Open Connect Everywhere: A Glimpse at the Internet Ecosystem through the Lens of the Netflix CDN. *ACM SIGCOMM Computer Communication Review* (2018).
[15] Daniel Byrne, Nilufer Onder, and Zhenlin Wang. 2018. mPart: miss-ratio curve guided partitioning in key-value stores. In *ACM SIGPLAN International Symposium on Memory Management.*
[16] Gregory Chockler, Guy Laden, and Ymir Vigfusson. 2010. Data caching as a cloud service. In *4th International Workshop on Large Scale Distributed Systems and Middleware.*
[17] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2015. Dynacache: Dynamic Cloud Caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing.*
[18] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *13th USENIX Symposium on Networked Systems Design and Implementation.*
[19] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. 2017. Memshare: a Dynamic Multi-tenant Key-value Cache. In *2017 USENIX Annual Technical Conference.*
[20] Siying Dong, Shiva Shankar P, Satadru Pan, Anand Ananthabhotla, Dhanabal Ekambaram, Abhinav Sharma, Shobhit Dayal, Nishant Vinaybhai Parikh, Yanqin Jin, Albert Kim, Sushil Patil, Jay Zhuang, Sam Dunster, Akanksha Mahajan, Anirudh Chelluri, Chaitanya Datye, Lucas Vasconcelos Santana, Nitin Garg, and Omkar Gawde. 2023. Disaggregating RocksDB: A Production Experience. *Proceedings of the ACM on Managment of Data* 1, 2, Article 192 (June 2023), 24 pages. https://doi.org/10.1145/3589772
[21] Bin Fan, David G. Andersen, and Michael Kaminsky. 2013. MemC3: compact and concurrent MemCache with dumber caching and smarter hashing. In *10th USENIX Symposium on Networked Systems Design and Implementation.*
[22] Yihui Feng, Zhi Liu, Yunjian Zhao, Tatiana Jin, Yidi Wu, Yang Zhang, James Cheng, Chao Li, and Tao Guan. 2021. Scaling Large Production Clusters with Partitioned Synchronization. In *2021 USENIX Annual Technical Conference.*
[23] Yupeng Fu and Chinmay Soman. 2021. Real-time Data Infrastructure at Uber. In *2021 International Conference on Management of Data.*
[24] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. 2015. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache . In *2015 USENIX Annual Technical Conference.*
[25] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. 2013. An analysis of Facebook photo caching. In *24th ACM Symposium on Operating Systems Principles.* ACM, 167–181. https://doi.org/10.1145/2517349.2522722
[26] Ricardo Koller, Ali José Mashtizadeh, and Raju Rangaswami. 2015. Centaur: Host-Side SSD Caching for Storage Performance Control. In *2015 IEEE International Conference on Autonomic Computing.*
[27] Diego Kreutz, Fernando M. V. Ramos, Paulo Jorge Esteves Veríssimo, Christian Esteve Rothenberg, Siamak Azodolmolky, and Steve Uhlig. 2015. Software-Defined Networking: A Comprehensive Survey. *Proceedings of the IEEE* 103, 1 (2015), 14–76. https://doi.org/10.1109/JPROC.2014.2371999
[28] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. 2014. MICA: a holistic approach to fast in-memory key-value storage. In *11th USENIX Symposium on Networked Systems Design and Implementation.*

[29] Ricardo Macedo, João Paulo, José Pereira, and Alysson Bessani. 2020. A Survey and Classification of Software-Defined Storage Systems. *ACM Computing Surveys* 53, 3, Article 48 (May 2020), 38 pages. https://doi.org/10.1145/3385896

[30] Sara McAllister, Benjamin Berg, Julian Tutuncu-Macias, Juncheng Yang, Sathya Gunasekar, Jimmy Lu, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2021. Kangaroo: Caching Billions of Tiny Objects on Flash. In *28th ACM Symposium on Operating Systems Principles*.

[31] Mariana Miranda, Yusuke Tanimura, Jason Haga, Amit Ruhela, Stephen Lien Harrell, John Cazes, Ricardo Macedo, José Pereira, and João Paulo. 2024. Can Current SDS Controllers Scale To Modern HPC Infrastructures?. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE.

[32] Ziyue Qiu, Juncheng Yang, Juncheng Zhang, Cheng Li, Xiaosong Ma, Qi Chen, Mao Yang, and Yinlong Xu. 2023. FrozenHot Cache: Rethinking Cache Management for Modern Hardware. In *18th European Conference on Computer Systems*.

[33] Trausti Saemundsson, Hjortur Bjornsson, Gregory Chockler, and Ymir Vigfusson. 2014. Dynamic Performance Profiling of Cloud Caches. In *Proceedings of the ACM Symposium on Cloud Computing*.

[34] David Shue, Michael J. Freedman, and Anees Shaikh. 2012. Performance Isolation and Fairness for Multi-Tenant Cloud Storage. In *10th USENIX Symposium on Operating Systems Design and Implementation*.

[35] Kevin Song, Jiacheng Yang, Zixuan Wang, Jishen Zhao, Sihang Liu, and Gennady Pekhimenko. 2025. HybridTier: an Adaptive and Lightweight CXL-Memory Tiering System. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ACM, 112–128. https://doi.org/10.1145/3676642.3736119

[36] Ioan A. Stefanovici, Eno Thereska, Greg O'Shea, Bianca Schroeder, Hitesh Ballani, Thomas Karagiannis, Antony I. T. Rowstron, and Tom Talpey. 2015. Software-defined caching: managing caches in multi-tenant data centers. In *Proceedings of the Sixth ACM Symposium on Cloud Computing*.

[37] Chunqiang Tang, Kenny Yu, Kaushik Veeraraghavan, Jonathan Kaldor, Scott Michelson, Thawan Kooburat, Aravind Anbudurai, Matthew Clark, Kabir Gogia, Long Cheng, Ben Christensen, Alex Gartrell, Maxim Khutornenko, Sachin Kulkarni, Marcin Pawlowski, Tuomas Pelkonen, Andre Rodrigues, Rounak Tibrewal, Vaishnavi Venkatesan, and Peter Zhang. 2020. Twine: A Unified Cluster Management System for Shared Infrastructure. In *14th USENIX Symposium on Operating Systems Design and Implementation*. USENIX Association, 787–803.

[38] Carl A. Waldspurger, Nohhyun Park, Alexander T. Garthwaite, and Irfan Ahmad. 2015. Efficient MRC Construction with SHARDS. In *13th USENIX Conference on File and Storage Technologies*.

[39] Qianli Wang, Si Wu, Yongkun Li, Yinlong Xu, Fei Chen, Pengcheng Wang, and Lei Han. 2022. HPUCache: Toward High Performance and Resource Utilization in Clustered Cache via Data Copying and Instance Merging. In *2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS)*.

[40] Tom White. 2015. *Hadoop: The Definitive Guide*. O'Reilly.

[41] Daniel Lin-Kit Wong, Hao Wu, Carson Molder, Sathya Gunasekar, Jimmy Lu, Snehal Khandkar, Abhinav Sharma, Daniel S. Berger, Nathan Beckmann, and Gregory R. Ganger. 2024. Baleen: ML Admission & Prefetching for Flash Caches. In *22nd USENIX Conference on File and Storage Technologies*. USENIX Association, 347–371.

[42] Xingbo Wu, Li Zhang, Yandong Wang, Yufei Ren, Michel Hack, and Song Jiang. 2016. zExpander: a key-value cache with both high performance and fewer misses. In *11th European Conference on Computer Systems*.

[43] Juncheng Yang, Yao Yue, and K. V. Rashmi. 2020. A large scale analysis of hundreds of in-memory cache clusters at Twitter. In *14th USENIX Symposium on Operating Systems Design and Implementation*.

[44] Juncheng Yang, Yao Yue, and Rashmi Vinayak. 2021. Segcache: a memory-efficient and scalable in-memory key-value cache for small objects. In *18th USENIX Symposium on Networked Systems Design and Implementation*.