Mercury: QoS-Aware Tiered Memory System

Jiaheng Lu*, Yiwen Zhang*, Hasan Maruf‡, Minseo Park‡, Yunxuan Tang, Fan Lai§, Mosharaf Chowdhury

University of Michigan ‡AMD §UIUC

Abstract

Memory tiering has received wide adoption in recent years as an effective solution to address the increasing memory demands of memory-intensive workloads. However, existing tiered memory systems often fail to meet service-level objectives (SLOs) when multiple applications share the system because they lack Quality-of-Service (QoS) support. Consequently, applications suffer severe performance drops due to *local memory contention* and *memory bandwidth interference*.

In this paper, we present Mercury, a QoS-aware tiered memory system that ensures predictable performance for coexisting memory-intensive applications with different SLOs. Mercury enables per-tier page reclamation for application-level resource management and uses a proactive admission control algorithm to satisfy SLOs via per-tier memory capacity allocation and intra- and inter-tier bandwidth interference mitigation. It reacts to dynamic requirement changes via real-time adaptation. Extensive evaluations show that Mercury improves application performance by up to 53.4% and 20.3% compared to TPP and Colloid, respectively.

1 Introduction

With the increasing memory demands of datacenter applications, tiered memory systems have been widely adopted to replace DRAM-only systems [27, 34, 43, 44, 49, 50]. Recent tiered memory systems are enabled by Compute Express Link (CXL) [11], a high-speed interconnect standard that allows memory capacity to grow by attaching memory to the CPU without losing nanosecond-scale memory access latency.

The increased memory capacity enables deployments of more memory-intensive applications that fall into two broad categories: (1) *latency-sensitive (LS)* applications, such as in-memory key-value store [13, 14], that require low-latency memory access, and (2) *bandwidth-intensive (BI)* applications, such as large-memory machine learning (ML) models (e.g., long-context language models or recommendation models [12, 38]), that require sustained memory bandwidth. In production environments, running a single application on a tiered memory system wastes both memory capacity and bandwidth [21, 27, 34, 36, 41, 44]. As a result, multiple memory-intensive applications often share the tiered memory system to maximize resource utilization and improve total cost of ownership (TCO).

Existing research on tiered memory systems primarily focuses on page temperature monitoring and efficient page migration to better utilize memory resources (i.e., fast-tier DRAM) [15, 32, 33, 36, 43, 50, 54]. However, these solutions optimize a single application running on a single server. They lack Quality-of-Service (QoS) support and cannot react to applications with different service level objectives (SLOs). In particular, existing tiered memory systems suffer from two sources of performance unpredictability (§2.2). First, multiple applications can *contend for local memory* (i.e., fast-tier memory), and the ones with hotter memory get more resources. As a result, a low-priority application may acquire more local memory than a high-priority application, leading to *priority inversion*. Second, high memory bandwidth generated by BI applications can degrade the performance of coexisting LS applications on the same tier or even across tiers – we refer to these phenomena as *intra- and inter-tier interference*, respectively. To the best of our knowledge, no existing solutions have systematically tackled both of these QoS challenges in a tiered memory system, including recent proposals on QoS support for tiered memory [27, 47].

We observe that providing predictable performance for applications with different SLOs in the presence of local memory contention and memory bandwidth interference requires addressing the following fundamental challenges:

- *Efficient resource tracking and control:* The memory management subsystems in existing operating systems (OSes) are not well designed for tiered memory. Hence, efficiently tracking and controlling memory resources on each memory tier is challenging.

- *Optimal memory allocation:* Determining the amount of memory to allocate to each application in each tier while maximizing resource utilization is non-trivial. For instance, migrating some local memory of a BI application with a relaxed SLO to CXL memory can save local memory for another LS application. However, excessive migration may cause inter-tier interference, leading to SLO violations.

- *Dynamic workload adaptation:* Because an application's workload can change after deployment, we need an efficient approach to performance monitoring and real-time adaptation.

In this paper, we present Mercury, a QoS-aware tiered memory system that provides predictable performance for memory-intensive applications. Mercury combines several key design ideas to overcome the aforementioned challenges. It achieves QoS-awareness in tiered memory by enabling *per-tier page reclamation* inside Linux kernel. This allows

Mercury to perform efficient application-level resource management while still leveraging existing page migration designs. At the core of Mercury's design is a novel admission control algorithm tailored for tiered memory. It proactively allocates the right amount of resources for applications to satisfy SLOs while mitigating intra-tier and inter-tier bandwidth interference. Mercury also performs real-time adaptation to unpredictable memory and bandwidth changes, preventing sudden load surges while maximizing the number of applications meeting their SLOs based on priority.

We evaluate Mercury's effectiveness in providing QoS using real-world applications. Mercury can closely track SLOs at different memory access latency and bandwidth targets. More importantly, Mercury is able to handle local memory contention as well as memory bandwidth interference at various multi-tenant settings. Extensive evaluations on 80 workloads across seven application categories show that Mercury achieves up to 53.4% better application performance than TPP and 20.3% better than Colloid, while satisfying more applications' SLOs. Mercury also achieves 8.4× longer SLO satisfaction time than TPP in a long-running experiment to handle dynamic workload changes.

We make the following research contributions:

- We conduct a thorough QoS analysis on production-ready tiered memory systems to qualitatively analyzing the performance impact of local memory contention and memory bandwidth interference.
- We propose a novel admission control and real-time adaptation algorithm tailored for tiered memory systems to achieve different SLOs for coexisting applications.
- We implement a new kernel-level resource management scheme to control resources on tiered memory. We plan to share our design with the Linux community for open discussion after the paper's publication.

2 Background and Motivation

We start by introducing two sources of performance unpredictability in tiered memory (§2.1), followed by a quantitative QoS analysis using CXL hardware and real-world applications (§2.2). We then motivate Mercury's design by analyzing why existing solutions fail to provide QoS support (§2.3).

2.1 Sources of Performance Unpredictability

There are two primary sources of unpredictable performance – local memory contention and memory bandwidth interference – as applications share memory resources in tiered memory.

Local memory contention. Multiple applications deployed on the same tiered memory system compete for pages on the fast tier (i.e., local memory). The core design principle of tiered memory systems is to keep hot pages on local memory (i.e., DRAM on the fast tier) while migrating cold pages to the slower tier [15, 27, 32, 33, 36, 43, 50]. However, such a design does not distinguish among applications with different SLOs. A low-priority application with a large amount of hot memory
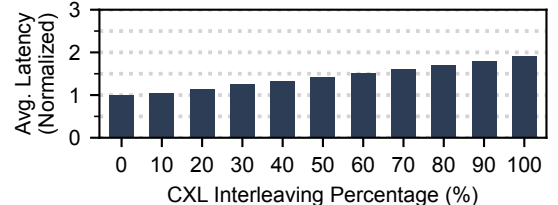
(a) Impact on Latency Performance
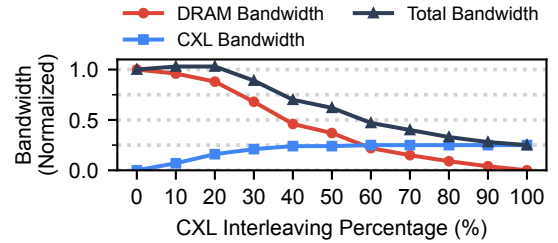
(b) Impact on Bandwidth Performance

Figure 1: Latency and bandwidth performance at different CXL interleaving ratios to illustrate the impact of local memory.

can still compete for local memory, hurting the performance of more important applications and causing priority inversion.

Memory bandwidth interference. Applications consuming high memory bandwidth can affect the performance of co-existing applications. The problem is further complicated in tiered memory systems, where applications can generate bandwidth with memory requests accessing different memory tiers. We identify two types of memory bandwidth interference – *intra-tier interference* and *inter-tier interference*. Intra-tier interference refers to the one happening on the same tier, which is also common in conventional, single-tier systems [22, 35, 37, 39]. Inter-tier interference occurs when excessive memory requests on one memory tier cause a slowdown of requests on another tier. Existing solutions on mitigating memory bandwidth interference work either (1) on the memory request level, such as memory request prioritization [39] or memory channel partitioning [37]; or (2) on the CPU core level to throttle memory bandwidth from best-effort tasks [22, 35]. None of these solutions is designed to work across multiple memory tiers and thus cannot handle inter-tier interference.

2.2 Quantitative Analysis: QoS in Tiered Memory

We now quantitatively analyze how local memory contention and memory bandwidth interference affect application performance and draw key insights to design Mercury. In this work, we classify applications into two types: (1) **Latency-sensitive (LS)** applications that desire low memory access latency, and (2) **bandwidth-intensive (BI)** applications that require sustained memory bandwidth. Our experiments run on dual-socket AMD Genoa servers with 96 physical cores and 12 DDR5 memory channels per socket. CXL memory is enabled via two Astera Labs memory expansion cards with 2
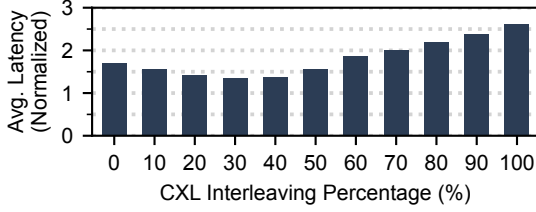
Figure 2: Performance of *LS* when *BI* requires bandwidth at different CXL interleaving percentage. Migrating *BI* to CXL does not always lead to better performance of *LS* due to inter-tier interference. *BI*'s performance is very close to Figure 1b and omitted for brevity.

DDR5 channels. Each card connects via x16 PCIe lanes (x32 lanes in total). The servers have a 1.8TB memory footprint with 768GB DDR5 per socket and 256GB CXL memory.

2.2.1 Impact of Available Local Memory

We develop two microbenchmarks to represent LS and BI applications, which we denote as *LS* and *BI* in this section for brevity. *LS* performs random memory access among a 4GB memory region. *BI* allocates enough CPU cores to generate memory bandwidth at maximum capacity on 128MB region per core. These microbenchmarks allow us to easily control the proportion of memory access on CXL memory (the rest goes to local memory), which we denote as the CXL interleaving percentage.

Figure 1 shows the impact of available local memory on latency and bandwidth performance by varying the CXL interleaving ratio of the two microbenchmarks. We observe the memory access latency of *LS* is proportional to its available local memory, and it becomes 2× slower when all memory is moved to the CXL tier in the worst case. On the other hand, *BI*'s bandwidth performance degrades as more memory is accessed from CXL, dropping to 25% of its original performance when all memory is accessed via CXL.

> **Takeaway #1**: Local memory directly affects the performance of both types of applications and should be allocated judiciously during memory contention.

2.2.2 Deep Dive in Memory Interference

We now take a closer look at how LS applications are affected by memory bandwidth interference from BI applications.

Varying *BI* across tiers. We first configure *LS* to always access local memory and vary *BI*'s CXL-interleaving percentage. The results are shown in Figure 2, and we make two key observations. First, initially the performance of LS improves due to the extra memory bandwidth from CXL channels. However, after a certain point, pushing more to the CXL degrades LS performance. Second, the interference becomes the worst when all memory bandwidth demand is generated from CXL; the total memory bandwidth is dictated by the CXL channel's bandwidth capacity.
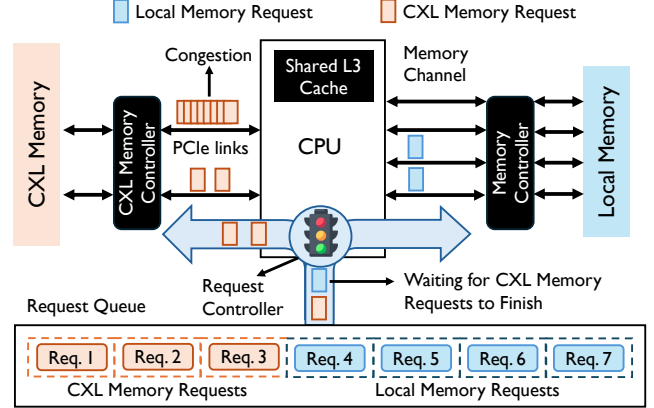

Figure 3: Architectural diagram of how memory requests are handled in CXL-enabled tiered memory.

This behavior can be explained by examining how memory requests are handled in a tiered memory system. Figure 3 illustrates the high-level overview of a CXL-enabled tiered memory architecture. There exist separate fixed-size queues in the hardware for memory requests to/from local memory and CXL memory. The effective bandwidth of the system depends on the depth of these queues. The interleaving policy [6] of OS processes memory requests by serving *m* local memory requests, followed by *n* CXL memory requests, where *m:n* represents the interleaving ratio between local and CXL memory (by default, it is 1:1). The high bandwidth demand on CXL memory can lead to a longer queuing delay when there is a shorter queue size. In addition, the total PCIe link bandwidth may not match the effective bandwidth of local memory channels and it will take comparatively longer to process a request. This delay, in turn, impacts the scheduling of local memory requests, resulting in inter-tier interference.

Going back to Figure 2, initially, when bandwidth demand on the local memory is high, its corresponding queue fills up, causing interference on local memory (i.e., intra-tier interference) to dominate. When *BI* starts to move requests from local memory to CXL, intra-tier interference decreases, causing latency to drop. When more requests are moved to the CXL side, the queue holding CXL requests builds up. Since both types of requests are handled by the same set of CPU cores, busy processing of the CXL requests can slow down the concurrent requests for local memory.

> **Takeaway #2**: We should determine the right amount of memory to migrate BI applications that *reduces intra-tier interference while keeping inter-tier interference low.*

Varying *LS* across tiers. Figure 4 shows the results of interference in a different setting, where we keep *BI* fixed on local memory and vary *LS*'s CXL interleaving percentage. This is used to simulate the scenario where one attempts to mitigate the interference on the fast tier by migrating *LS*'s requests away from local memory to CXL memory. However, such an
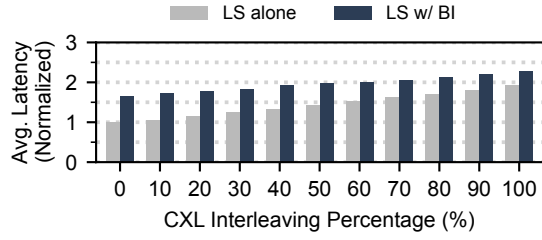
Figure 4: Performance of *LS* at different CXL interleaving ratios when *BI* is fixed on local memory. Migrating more requests away from local memory does not improve performance as more requests are accessing the slower tier.
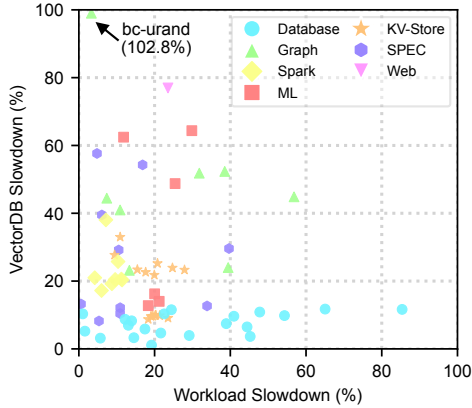

Figure 5: Unpredictable performance of 80 workloads and VectorDB as they compete for local memory on the fast tier. Existing solutions cannot distinguish among applications when migrating their hot pages, and thus cannot provide QoS guarantees.

approach leads to worse performance because more memory requests are now accessing the slow tier.

> **Takeaway #3**: We should proactively mitigate the interference on local memory to provide predictable performance for LS applications.

2.2.3 Impact on Real Applications

To illustrate the impact of memory capacity and bandwidth contentions on real-world applications in a tiered memory system, we perform the same set of experiments on local memory contention and memory bandwidth interference across 80 applications across seven diverse categories. Appendix A describes the full list of applications used. We run these experiments on dual-socket Intel Xeon Gold 6330 servers with 56 physical CPU cores where each socket contains 512GB DDR4 memory. For controlling the environmental setup, we use the remote socket as the slow tier, with uncore frequency tuned down to match the memory access latency in our real CXL hardware. All the experiments here allocate different physical cores to foreground and background workloads, ensuring that no core contention occurs.

	Database	Graph	KV-Store	ML	SPEC	Spark	Web
Workload Slowdown (%)	20.8	42.6	21.4	27.3	32.5	20.0	22.1
llama.cpp Slowdown (%)	4.9	17.2	5.0	6.9	12.2	11.2	2.9

(a) Intra-tier Interference

	Database	Graph	KV-Store	ML	SPEC	Spark	Web
Workload Slowdown (%)	8.4	13.9	11.6	6.5	18.5	20.7	13.5
llama.cpp Slowdown (%)	0.85	1.3	2.5	0.23	0.4	2.0	2.4

(b) Inter-tier Interference

Figure 6: llama.cpp's memory bandwidth creates interference, resulting in a significant drop in the throughput performance of all kinds of workloads. (a) shows intra-tier interference when llama.cpp is on the same fast tier with different workloads. (b) shows inter-tier interference after all llama.cpp's memory is migrated to CXL.

Figure 5 presents the impact of local memory contention. In this experiment, VectorDB is selected as the background application, and each time another application is launched as the foreground workload. We enable TPP [36], a state-of-the-art tiered memory system, to perform page migration. The total working set size (WSS) of the applications in each setting exceeds the available local memory on the fast tier. We observe the performance degradation between the foreground and background applications highly depends on their memory access frequency. The foreground application with lower memory access frequency than VectorDB (e.g., TPC-H) always experiences larger performance degradation, and vice versa (e.g., SPEC, DLRM). In cases of similar memory access frequencies, both applications are affected (e.g., Graph). The results are unsurprising as TPP does not distinguish between applications with different SLOs during page migration.

Memory bandwidth interference can also greatly impact coexisting applications. Figure 6 illustrates the impact of both types of memory bandwidth interference. We select llama.cpp [12] inference tasks (running on local DRAM) as the background task to create high memory bandwidth demand. Under TPP, all foreground applications we tested experienced up to 50% performance degradation due to intra-tier interference. When all llama.cpp's memory is migrated to the slow tier, up to 32% slowdown is observed due to inter-tier interference.

2.3 Lack of QoS Support in Existing Solutions

We now summarize why existing tiered memory systems fail in providing QoS, and thus calling for a new QoS-aware tiered memory system. There are two major design principles adopted by existing solutions [15, 27, 32, 33, 36, 43, 47, 50] that are proved to be beneficial for memory-intensive workloads: (1) optimizing *system-level* performance such as overall memory access latency and bandwidth; and (2) *reactively* allocating resources based on memory-level measurements. The first principle assumes a single application to be optimized, and the second principle assumes workloads remains
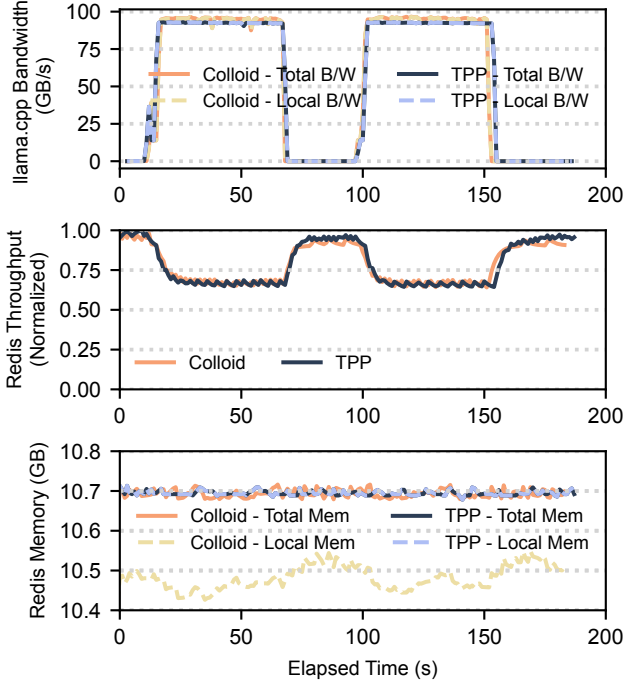
Figure 7: llama.cpp's memory bandwidth creates interference, resulting in a significant drop in Redis throughput.

in steady state after their ramp-up period. However, neither assumptions hold when it comes to providing QoS support.

We conduct a motivating experiment illustrating how TPP and Colloid [47], two state-of-the-art solutions, are inefficient in providing predictable performance. In this experiment, we measure how the performance of Redis is affected by two consecutive inference requests from llama.cpp, as shown in Figure 7. In this setup, Redis is considered more important than llama.cpp, which is configured to be an offline batch inference workload. Each of the two requests from llama.cpp caused 100 GB/s bandwidth spikes, which leads to two drops in Redis throughput.

Neither TPP nor Colloid identifies Redis as an application that is under bandwidth interference. TPP does not migrate any pages of llama.cpp and Redis because WSS of both applications fit in local memory, and TPP is not designed to handle interference. On the other hand, Colloid migrates the pages of Redis to the slow tier to balance out the memory access latencies caused by increased contention due to llama.cpp's requests. In addition, the demotion of pages are too slow to make noticeable impact to recover application performance.

In the next section, we discuss the decision choices we make in designing Mercury to enable *QoS-awareness* and *proactive resource allocation*.

3 Mercury Overview

With multiple applications sharing a tiered memory system, our goal in this work is to provide predictable performance among applications with different SLOs under memory resource contention and memory interference.

3.1 Design Principles

QoS granularity. The first step to QoS-awareness is determining at what granularity we should support QoS and manage memory resources. There exist multiple levels where QoS can be enforced, such as (1) an entire application, (2) individual data structures, (3) individual `mmap()` calls, or even (4) individual memory accesses. Although finer granularity allows more precise QoS assignments across different segments of the application, we build Mercury at the *application level* to ensure (1) easy deployment without modifying applications, (2) a clean QoS interface, and (3) low overhead in managing memory resources and performance monitoring. Additionally, providing QoS at the application level allows us to reuse existing kernel mechanisms for identifying and migrating hot and cold pages, which has proved to be efficient [36].

Proactive memory resource allocation. Instead of overprovisioning resources to handle worst-case overload scenarios, we decide to proactively allocate *the right amount of resource that can satisfy an application's SLO*. This allows Mercury to accommodate more applications while meeting their SLOs. Following this design principle, Mercury provides QoS via a combination of *admission control* and *real-time adaptation*. The former admits applications using the right amount of resources, and the latter adjusts resource allocations for applications when runtime workload changes affect QoS.

Choice of performance indicators. Selecting the right performance indicators is critical for providing QoS, as Mercury needs to react quickly to meet applications' SLO. In Mercury, we prefer *low-level metrics* as performance indicators compared to application-level metrics. Low-level metrics can be collected via hardware-based PMU counters. Their measurements require no application modification, and as we will soon show, they closely reflect the application performance and can react faster to real-time workload changes (§5). In particular, we collect *memory access latency* and *memory bandwidth* per application. On AMD processors, we measure memory access latency using memory load events from L3 cache misses provided by Instruction Based Sampling (IBS) [26], and memory bandwidth is measured by μProf [16]. On Intel platforms, latencies can be measured through processor event-based sampling (PEBS) [10] where bandwidth can be measured via Intel Platform QoS (PQoS) [9].

Prioritization. When multiple memory-intensive applications coexist, there is no guarantee that we will always have enough resources to satisfy everyone's SLO. For example, an important application may arrive later than less critical ones, and an application may require more resources to fulfill increasing load. To this end, Mercury leverages *strict priority* to ensure high-priority applications get guaranteed performance even when the tiered memory runs out of resources.[1] We

---

[1]Lower priority applications that yield (part of) their resources to higher priority ones continue to run as best-effort to preserve work conservation.
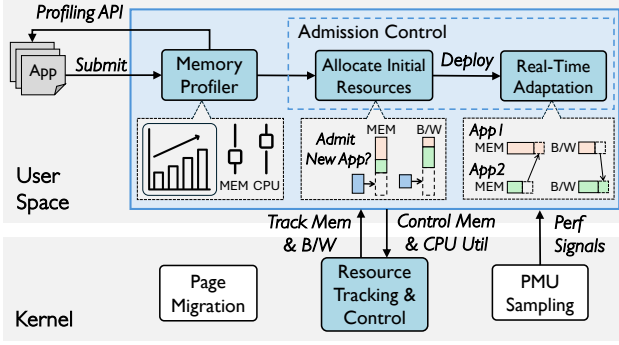
Figure 8: High level system overview of Mercury. The memory profiler determines the memory resources needed to maintain SLOs when applications are running in isolation. The admission controller takes this information and proactively determines the right resource to allocate for applications while mitigating interference. It also performs real-time adaptation to adjust resource allocation during runtime. The resource controller tracks and controls resources at the application level.

choose to apply a separate priority scheme on top of SLOs because more stringent SLOs do not always mean higher priority, and BI applications are not necessarily less important than LS ones. In Mercury, priority levels are uniquely assigned to avoid ties among applications.

3.2 System Overview

Figure 8 presents a system diagram of Mercury. Its design includes a memory profiler, an admission control algorithm, as well as a memory resource controller implemented inside the Linux kernel.

A Mercury user (e.g., a cluster operator) submits an application along with a list of information, including the number of CPU cores, memory requirement, application type, its priority level, and the SLO. The SLO of LS applications specifies a **memory access latency** target, and that of BI ones is a **maximum memory bandwidth** the application needs.

Mercury proactively controls two types of resources, **available local memory** and **cpu utilization** at the application level. Since the existing kernel mechanism on memory usage tracking and controlling does not distinguish among different tiers, Mercury enables per-tier page reclamation to control an application's available local memory (§4.1). Mercury adopts existing Linux page LRUs and NUMA Balancing [7] for page temperature detection and migration. When Mercury receives an application, it first profiles offline to find the minimum amount of memory resource needed to satisfy the SLO when running in isolation (§4.2). The profiling procedure only takes less than 1% of the workload's entire lifetime, incurring negligible overhead. The profiler also provides users with an API to measure the two SLO metrics to help them set an appropriate SLO. After profiling, Mercury's admission control determines whether the new application should be admitted onto the node, and if so, how much resource we should allocate to the application, along with any changes to the existing resource allocation of other applications based on the appli-

cation's priority level (§4.3.1). The admission control keeps monitoring an application's performance during its lifetime and performs real-time adaptation on its resource allocation to ensure SLO is maintained in case of workload changes and bandwidth interference (§4.3.2).

4 Mercury Design

In this section, we describe Mercury in details, including its application-level resource management, memory profiling, and admission control with a real-time adaptation module.

4.1 Application-Level Resource Management

Mercury manages two resources – local memory and CPU utilization – to enforce SLO. The available local memory for an application determines the ratio of memory accesses on local vs. CXL memory, which directly affects an LS application's performance. As local memory is limited and shared among all applications, Mercury decides for each LS application a *local memory limit* based on their SLO. Limiting an application's local memory can also affect its bandwidth, as accessing more memory from a slower tier decreases the aggregate bandwidth (**Takeaway #1** in §2.2.1). Meanwhile, it saves additional local memory to accommodate other applications. However, this approach alone is not enough to control an application's bandwidth for two reasons. First, placing too many memory requests on CXL memory creates inter-tier memory interference (§2.2.2). Second, the bandwidth cannot be tuned further down after migrating all pages to CXL memory (Figure 1b). Therefore, Mercury also limits the CPU utilization of BI applications in addition to local memory limit in order to achieve a finer-grained control over bandwidth.

After deciding what resources Mercury manages for applications, the next step is to seek an efficient way of tracking and controlling those resources. Linux kernel already provides cgroup [8] to track and limit the memory and CPU usage of a collection of processes. However, the memory control mechanism in cgroup cannot be directly applied in Mercury, because it does not distinguish between different memory tiers when tracking and controlling memory usage. Instead, one can only specify a total memory limit that accounts for the total memory usage across all tiers for a given process, then the virtual pages will be swapped to the disk if the memory limit is reached.

Per-tier memory management in Linux cgroup. We make a series of modifications on cgroup to enable per-tier memory tracking and control. While observing the total memory usage of an application, we break it down into individual tier-level usage. In a CXL-enabled tiered memory system, memory nodes appear to the system as separate NUMA nodes. We can thus assume that each tier consists of one or multiple NUMA nodes. We enhance cgroup to track the list of pages within each LRU associated with the NUMA nodes across a specific memory tier. Specifically, we introduce a new memory limit-controlling interface, `memory.per_numa_high` that

controls the max memory usage for an application on each NUMA node. Note that `memory.per_numa_high` works concurrently with the global cgroup interface (`memory.high`). While the latter controls the total system-wide memory usage of an application, our interface restricts the contribution of an application's memory footprint on each memory tier.

When allocating pages, Mercury uses Linux's default "allocate on fast memory tier first" page allocation policy unless specified otherwise. However, if the memory usage of a NUMA node within a specific memory tier exceeds its `memory.per_numa_high` threshold, Mercury stops memory allocation and initiates page reclamation only on that NUMA node. Here, the default reclamation mechanism is to demote (i.e., page migration) to the next available memory tier. Moreover, a change to `memory.per_numa_high` (e.g., by Mercury or system admins) immediately triggers reclamation across all the nodes where the new limit is below the current memory usage. Similar to TPP [36], we leverage NUMA Balancing [7] to allow page promotion among different nodes.

Mercury controls an application's CPU utilization using the native cgroup `cpu.max` interface. By configuring this parameter, Mercury can adjust the application's utilization across all CPU cores it employs. Compared with vendor-provided bandwidth control mechanisms – such as Intel's Memory Bandwidth Monitoring (MBM) [40] – managing CPU utilization in this manner offers a more fine-grained and predictable approach [51]. Details of how Mercury selects an application's local memory limit and CPU utilization to provide QoS will be described next.

4.2 Memory Profiler

Before deploying an application, we first need to determine the right amount of memory resource needed to meet its SLO to provide QoS for more applications. In Mercury, this task is handled by the memory profiler.

Taking a user application and its SLO as inputs, the profiler finds the right amount of local memory the application needs to satisfy its SLO when running in isolation. For both types of applications, it starts by putting all their memory on the fast tier with full CPU utilization. If the SLO is not met even at this initial stage, the application is marked as *inadmissible*; the user should adjust the SLO or deploy it on another machine with larger memory or bandwidth. Otherwise, the profiler decreases its local memory limit until the measured performance matches the SLO. For BI applications, if the actual bandwidth is still above the SLO after the local memory limit drops to zero, the profiler starts to decrease CPU utilization until the SLO is met.

Note that the local memory limit found during profiling represents the minimum memory the application needs in isolation and is used by the admission control (§4.3) to determine the actual local memory and CPU utilization to allocate during deployment. Mercury also records the profiled memory bandwidth for BI applications, which represents the target bandwidth to meet its initial load during admission.

Finally, Mercury also performs a one-time profiling per machine to characterize inter-tier memory bandwidth interference, which will later be used by our admission control and real-time adaptation. Specifically, Mercury determines *two thresholds*, (i) a threshold on *healthy local bandwidth* ($Thresh_{local\_bw}$) and (ii) a threshold on *the rate of remote NUMA hint fault* ($Thresh_{numa}$), to monitor whether the system has reached the boundary of triggering severe intra-tier and inter-tier interference. The profiler leverages the *LS* and *BI* microbenchmarks (§2.2) to perform this task. To determine $Thresh_{local\_bw}$, the profiler launches both *LS* and *BI* on the fast tier, and increases *BI*'s bandwidth until a noticeable interference on *LS*'s latency is found. Similarly, $Thresh_{numa}$ is determined by sweeping the CXL interleaving percentage of *BI* until an obvious performance degradation is observed on *LS* running on the fast tier (10% performance degradation is set for both thresholds in our implementation).

```
1  if Local_avail_mem > app.profiled_mem_limit:
2      Allocated_mem = app.profiled_mem_limit
3  else:
4      priority_queue.yieldMem(app.profiled_mem_limit)
5      Allocated_mem = Current_local_avail_mem
6
7  if app.type == "BI":
8      # Local_avail_bw considers intra-tier intf.
9      if Local_avail_bw > app.profiled_bw:
10         Allocated_bw = app.profiled_bw
11     else:
12         # yieldBW considers inter-tier intf.
13         priority_queue.yieldBW(app.profiled_bw)
14         Allocated_bw = Current_local_avail_bw
```

Listing 1: Admitting LS and BI applications.

4.3 Admission Control

At the core of Mercury's design is its admission control algorithm, which proactively assigns resources among multiple applications at the presence of local memory contention and memory bandwidth interference. A naive solution is to take each incoming application's profiling result and keep deploying applications until local memory or bandwidth capacity runs out; i.e., first-come-first-service (FCFS). However, this approach has two drawbacks. First, a critical application may arrive late, and the resources already assigned to less critical applications cannot be taken back. Second, it does not consider the impact of bandwidth interference, which can lead to SLO violation.

In Mercury, we design a new admission control algorithm tailored for tiered memory. It leverages *strict priority scheme* to prioritize important applications, and *maximizes local memory utilization* to admit more applications with SLO guarantees while *avoiding intra- and inter-tier interference*. To achieve the aforementioned goal, there are two key questions we need to answer during admission control: (1) How to allocate initial resources when admitting applications to shared tiered memory? (2) How to ensure admitted applications can preserve their SLO guarantees in the presence of memory interference and workload changes?

4.3.1 Determining Initial Resource Allocation

Listing 1 presents a high-level overview of how Mercury determines the initial resources when admitting applications. The key idea in this step is to *proactively* allocate initial resources to satisfy as many applications' SLOs as possible while mitigating potential memory interference. At the arrival of a new application, Mercury immediately drops it if it has been labeled inadmissible by the profiler since its SLO (either latency or bandwidth) can never be met. Otherwise, Mercury starts to assign resources based on the application type.

Admitting LS applications. Mercury directly admits an LS application if there is available local memory to satisfy its profiled local memory requirement. Otherwise, if the current local memory is smaller than its profiled memory limit, Mercury tries to find memory from applications with a lower priority to *yield local memory*, starting from the one with the lowest priority to give this memory. If the victim application's local memory limit drops to zero, Mercury moves on to the next victim until no existing applications with lower priority can yield local memory (line 1-5 in Listing 1).

Admitting BI applications. Given a BI application, the goal is to assign the right amount of local memory and CPU utilization to satisfy its bandwidth SLO. Mercury starts by assigning local memory to BI applications in the same way as it does to LS ones. However, assigning the profiled local memory to the incoming application does not always mean it can achieve its bandwidth SLO as existing applications may have already taken a large portion of the total bandwidth capacity. Under this scenario, Mercury finds existing BI applications with lower priority in ascending order to *yield bandwidth*. Similar to the procedure in *yield local memory*, if the victim BI application's bandwidth cannot be reduced anymore, it will move on to the next application until no existing applications with lower priority can yield bandwidth (line 7-14 in Listing 1). This is done by decreasing the local memory limit of the victim application(s). During this process, Mercury monitors the current rate of remote NUMA hint faults and avoids inter-tier interference by switching to decreasing the victim application's CPU utilization once $thresh_{numa}$ is exceeded (**Takeaway #2** in §2.2.2). In the meantime, Mercury watches out for potential *intra-tier interference* on the fast tier by verifying if (1) there exist LS applications with higher priority, and (2) the healthy bandwidth threshold ($thresh_{local\_bw}$, §4.2) for the fast tier is exceeded. If both conditions are met, Mercury stops assigning bandwidth on the fast tier to the incoming application. The additional checks in *yield memory* ensures newly admitted applications do not start by creating severe interference, before more fine-grained control on resources kicks in to maintain application SLOs, which we will describe next.

4.3.2 Adaptating to Interference and Workload Changes

So far, we have discussed how Mercury admits applications and by proactively allocating resources during applica-

tion launch time to prevent undesired interference. However, merely relying on this initial resource allocation does not always preserve SLOs. SLO violations can still happen in the following two scenarios. First, LS applications can still miss SLOs due to bandwidth interference when BI applications' bandwidth sits around one of the two interference thresholds. Second, the amount of resource needed to maintain an application's SLO can shift over time due to workload changes after launch. Therefore, Mercury should adjust resource allocations dynamically in real time to ensure QoS guarantees for critical applications in such scenarios.

```
1  for app in priority_queue:
2      if app.SLOSatisfied:
3          yieldResource(app)
4      elif !app.SLOSatisfied:
5          if app.type == "BI" and app.cpu_util < 1:
6              app.increaseCPUUtil()
7          elif BWReducible():
8              priority_queue.yieldBW()
9          elif app.total_mem < app.local_mem_limit:
10             priority_queue.yieldMem()
11     if all_apps_satisfied and lowestPrio(app):
12         priority_queue.workConservation()
```

Listing 2: Real-Time Adaptation in Admission Control

Mercury evaluates all applications and makes necessary real-time adjustments to resource allocation every 200 ms. Listing 2 presents an overview of this process. Applications are processed in the order of descending priority to ensure critical applications are protected first. For each application, Mercury verifies whether the measured performance (i.e., memory access latency for LS applications and memory bandwidth for BI applications) meets its SLO. If the measured performance of an application is better than its SLO, Mercury attempts to yield its resources to benefit existing or future applications (line 3). Mercury *monitors inter-tier interference* during this process (using $thresh_{numa}$) to prevent hurting other applications. For BI applications, once the local memory limit reaches zero, Mercury starts reducing CPU utilization to yield more bandwidth.

It is more challenging when the measured performance is worse than SLO, as there may be multiple possible causes. Mercury takes a series actions to identify the correct cause and make necessary adjustments.

(1) *Increase CPU utilization.* For a BI application, Mercury first attempts to increase its CPU utilization (line 6). By increasing CPU utilization first, we improve the application's bandwidth without unnecessarily consuming additional local memory resource.

(2) *Mitigate bandwidth interference.* Next, Mercury decreases the bandwidth of low-priority applications until no lower-priority BI applications are available for reduction (lines 7–8). The bandwidth reduction process follows the same approach used in yielding bandwidth during initial resource allocation (§4.3.1): it reduces the bandwidth of BI applications in order of priority by lowering their local mem-
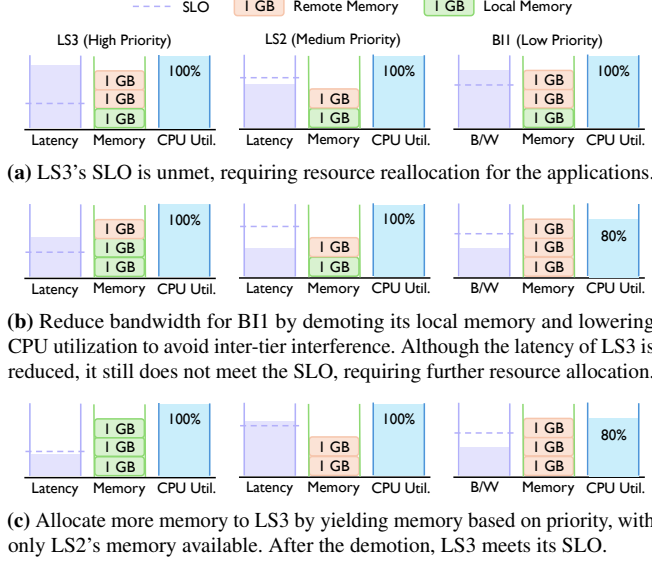
(a) LS3's SLO is unmet, requiring resource reallocation for the applications.

(b) Reduce bandwidth for BI1 by demoting its local memory and lowering CPU utilization to avoid inter-tier interference. Although the latency of LS3 is reduced, it still does not meet the SLO, requiring further resource allocation.

(c) Allocate more memory to LS3 by yielding memory based on priority, with only LS2's memory available. After the demotion, LS3 meets its SLO.

Figure 9: Examples of real-time adaptation. There are three applications in the system, with different SLO targets and priorities.

ory limits, while simultaneously monitoring inter-tier interference to initiate CPU utilization reduction. This step verifies if the performance drop is caused by interference. Although increasing local memory limit can also improve performance, we have shown it is more effective to reduce bandwidth interference first (**Takeaway #3** in §2.2.2). Since all applications go through our real-time adjustment in strict priority order, Mercury ensures the BI application that generates excessive bandwidth will eventually be punished, and other BI applications will retain SLO if there is enough bandwidth.

(3) *Increase local memory allocation.* If the measured performance is still worse than SLO, we are certain the target application needs more memory due to a workload change. In this case, Mercury allocates memory to the application in the same way, it finds the local memory from low-priority applications in ascending order of the priority level. Similar to how initial local memory is allocated (§4.3.1), Mercury finds the local memory from low-priority applications in ascending order of the priority level.

After going through all the applications in the current adaptation period, if there is additional available memory and all applications' SLOs are satisfied, Mercury allocates the remaining memory to applications based on descending priority for work conservation.

An example. Figure 9 illustrates an example of real-time adaptation by Mercury. The system hosts three applications: BI1, LS2, and LS3, with priorities in ascending order (BI1 has the lowest priority, LS3 the highest). At a specific time (Figure 9a), Mercury detects that the latency of LS3 exceeds its SLO target and decides to allocate more resources to improve its performance. First, Mercury reduces the bandwidth allocated to lower-priority BI applications in descending priority order. As a result, BI1's memory is demoted to the

CXL side. This demotion eventually exceeds $thresh_{numa}$, and Mercury decides to reduce BI1's CPU utilization to mitigate inter-tier interference (Figure 9b). The reduction in BI1's bandwidth improves the latencies of both LS2 and LS3. However, LS3's latency remains worse than its SLO, and thus Mercury proceeds to allocate additional memory to LS3. It reclaims memory from lower-priority applications based on their priority order. At this point, among applications with lower priority than LS3, only LS2 has local memory available. Therefore, Mercury demotes 1 GB of LS2's memory, reallocating it to LS3. Consequently, LS3's latency meets its SLO target (Figure 9c).

5 Evaluation

We evaluate Mercury's capability of providing QoS among applications sharing a tiered memory system, in the presence of local memory contention, bandwidth interference, and dynamic workload changes. Our key findings are as follows:

(1) Mercury closely tracks SLOs for both LS and BI applications by allocating right amount of resources (§5.2).

(2) Mercury effectively handles local memory contention and bandwidth interference, achieving up to a 53.4% improvement compared to TPP (§5.3–§5.5).

(3) Mercury's real-time adaptation mechanism accurately reallocates resources to prioritize critical applications during runtime workload changes, resulting in 14.9% and 20.3% performance improvement compared to TPP and Colloid (§5.6).

5.1 Experiment Setup

We implement Mercury's resource management module on Linux kernel v6.6. Mercury's user space components are written in C++ with 3000 lines of code. We illustrate Mercury's effectiveness in providing QoS among 80 workloads (Details on applications are in Appendix A). We also compare Mercury with TPP [36] and Colloid [47], two state-of-the-art open-source tiered memory systems. We run each experiment five times and report the average measurements.

5.2 SLO Compliance

We start by evaluating how closely Mercury tracks SLOs of both types of applications. We first look at simple scenarios where a single application is running and leave more complicated multi-tenant settings in the follow-up subsections. Figure 10a records the achieved memory access latency of Redis at different SLOs, along with the local memory limit (recorded as the percentage w.r.t its 20GB WSS) Mercury sets. Mercury is able to closely track the SLO by assigning the right amount of local memory.

Mercury is also good at tracking the SLO of BI applications. Figure 10b shows llama.cpp's bandwidth performance and its resource allocation by Mercury under different SLOs. When bandwidth SLO is small (e.g., 30GB/s and below in this experiment), Mercury further decreases its CPU utilization, as merely migrating all memory to CXL memory(by setting the
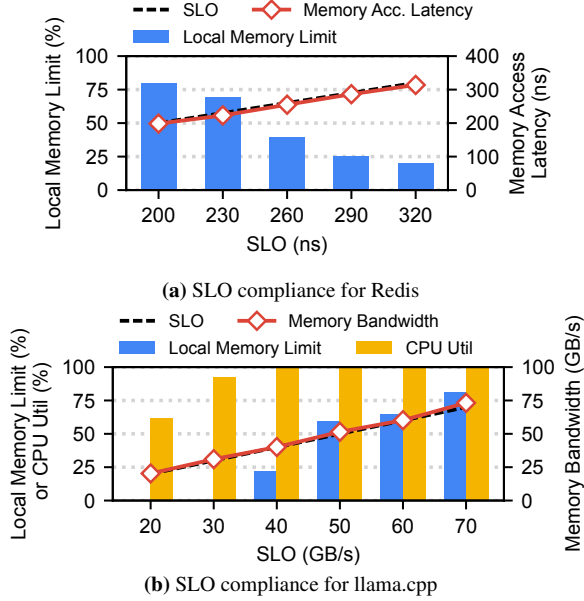
(a) SLO compliance for Redis

(b) SLO compliance for llama.cpp

Figure 10: Mercury provides SLO compliance on both memory access latency (for Redis) and bandwidth (for llama.cpp).

local memory limit to zero) still achieves higher bandwidth than needed. Note that in this case, Mercury does not limit its migration for inter-tier interference, as there is no other LS application running in the system.

5.3 Handling Local Memory Contention

Next, we evaluate Mercury in multi-tenant settings to understand its ability to manage local memory contention. The experiment follows the same workload setup as depicted in Figure 5, where VectorDB is running as a background application to compete with other workloads. The results comparing Mercury with TPP are shown in Figure 11. Notably, most workload coordinates shift towards (0,0), reflecting improved performance. For Database and KV-store workloads which fail to acquire sufficient local memory when competing with
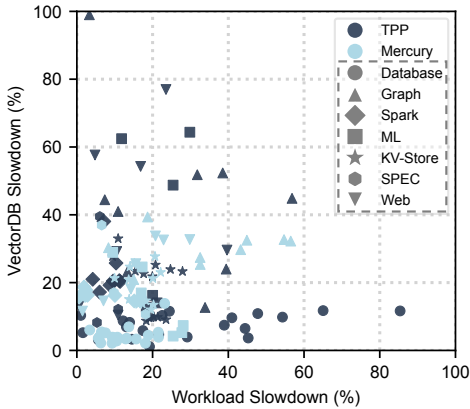

Figure 11: Comparing Mercury with TPP when handling local memory contention between workloads and VectorDB.
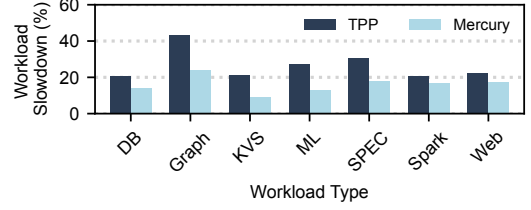

Figure 12: Mercury achieves better performance across all 7 types of workloads for handling memory bandwidth interference.

	Redis Mem. Acc. Lat. (ns)	Redis Tput (Kops/s)	llama.cpp B/W (GB/S)	llama.cpp Lat. per Token (ms)	VectorDB Mem. Acc. Lat. (ns)	VectorDB Query Lat. (ms)
SLO	460		40		290	
Mercury	453	179	40.5	963	286	26
TPP	198	325	34.1	1157	1187	55.8

Figure 13: Comparing Mercury with TPP when handling both local memory contention and memory bandwidth interference among Redis, llama.cpp, and VectorDB.

VectorDB under existing solutions, Mercury is able to ensure stable memory allocation, achieving a maximum slowdown reduction of foreground workload from 29% to 12%. In cases where workloads such as Graph, ML, SPEC, Spark, and Web can grab excessive local memory over vectorDB, Mercury manages to maintain VectorDB's SLO and achieves a maximum slowdown reduction of VectorDB from 75% to 14%, while minimally affecting the performance of foreground workloads.

5.4 Handling Memory Bandwidth Interference

In addition to addressing local memory contention, Mercury also tackles memory bandwidth interference as a key focus in its admission control design. We evaluate Mercury's effectiveness in managing bandwidth interference across various application combinations. In these experiments, we configure the working set size (WSS) of applications to eliminate local memory contention. As illustrated in Figure 12, we run llama.cpp as a background workload to simulate offline inference scenarios (batch inference), where llama.cpp's priority is low because latency is less critical in this context. We test 80 foreground workloads spanning 7 diverse types. For each workload type, Mercury effectively regulates llama.cpp's bandwidth usage, significantly reducing intra-tier interference while avoiding inter-tier interference. In contrast, TPP fails to meet SLO guarantees, causing foreground workloads to suffer performance degradation due to severe intra-tier interference. Compared to TPP, Mercury improves the performance of all 80 workloads while having minimal impact on llama.cpp's performance. Notably, for ML workloads, Mercury achieves up to a 40% performance improvement over TPP.

5.5 Mixing Two Sources of Unpredictability

It is quite likely that local memory contention and memory bandwidth interference appear simultaneously in a real-
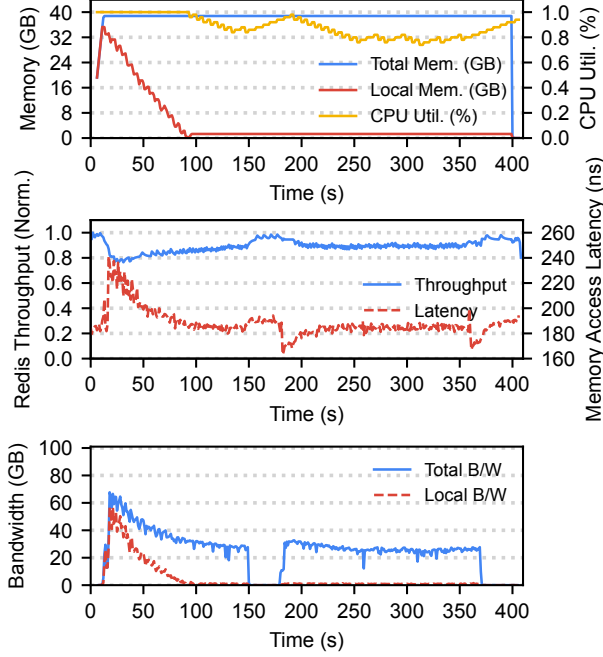
Figure 14: Performance of Redis and llama.cpp under real-time changes using Mercury. Priority of Redis is higher than llama.cpp

Figure 15: Performance of Redis and llama.cpp under real-time changes using Mercury. Priority of Redis is lower than llama.cpp

world setting. To evaluate how Mercury performs in this case, we deploy Redis, llama.cpp, and VectorDB all together with a local memory capacity of 40GB. The WSS for Redis, llama.cpp, and VectorDB is 40GB, 40GB, and 20GB, respectively. Figure 13 compares Mercury with TPP on achieved low-level and application-level performance. TPP allocates almost all local memory to Redis as Redis has the highest memory access frequency among the three applications, and thus most of llama.cpp's and VectorDB's memory are placed on CXL memory. This allocation satisfies only the SLO of Redis, as llama.cpp suffers from insufficient bandwidth, and VectorDB's performance degrades due to interference from llama.cpp and not enough local memory. In contrast, Mercury satisfies the SLOs for all three applications by allocating the right amount of memory (10GB for Redis, 20GB for VectorDB, and 10GB for llama.cpp) and managing llama.cpp's bandwidth to minimize interference, resulting in a 53.4% performance improvement for VectorDB.

5.6 Real-time Adaptation

5.6.1 Mitigating Dynamic Bandwidth Interference

In Section 2.3, TPP and Colloid failed to mitigate llama.cpp's bandwidth interference on Redis, leading to high latency spikes and missed SLO. An experiment illustrating how Mercury handles QoS in this scenario is shown in Figure 14. Initially, llama.cpp's local memory is demoted to remote memory (0–100s) to reduce bandwidth usage. Following this, as all of llama.cpp's memory has been migrated to remote memory, Mercury decreases its CPU utilization and ensure $thresh_{numa}$ is not exceeded to prevent inter-iteration interference (100–150s). Subsequently, Mercury increases llama.cpp's CPU uti-
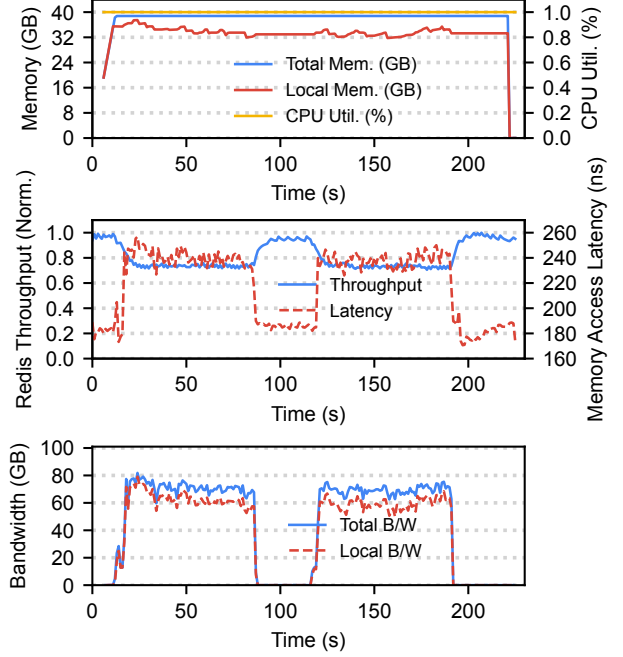
lization upon detecting that its bandwidth is too low, with no bandwidth interference since no requests are being processed during this period (150–200s). Once the second request arrives (after 200s), the system adjusts in a manner similar to the previous behavior. Compared to TPP and Colloid (in Figure 7), Mercury improves Redis's average throughput by 14.9% and 20.3%, respectively.

Next, we set llama.cpp's priority to be higher and repeat the same experiment to illustrate Mercury can handle various QoS settings. The results are shown in Figure 15. In this case, the bandwidth SLO of llama.cpp is set at 70 GB/s, which Mercury manages to maintain without being impacted by Redis. Redis experiences some latency spikes as it now has lower priority, and Mercury prioritize llama.cpp's high bandwidth over Redis performance based on strict priority.

5.6.2 Long-Running Experiment

We complete our evaluation with a long running experiment to illustrate how Mercury adapts to different dynamic changes. Here, we run Redis, llama.cpp, and VectorDB together and adjust llama.cpp's bandwidth and Redis's memory usage. The WSS for Redis, llama.cpp, and VectorDB is 30GB, 40GB, and 40GB, respectively, with the local memory capacity constrained at 70 GB. The priority level of the applications in descending order is Redis, llama.cpp, and VectorDB. We set the SLO for Redis, llama.cpp, and VectorDB to be 200ns, 70GB/s, and 180ns.

Initially, Redis and llama.cpp launch together without memory contention or bandwidth interference. When Mercury detects llama.cpp's load arrives at $t = 60s$, it quickly mitigates its bandwidth interference to ensure Redis can still maintain
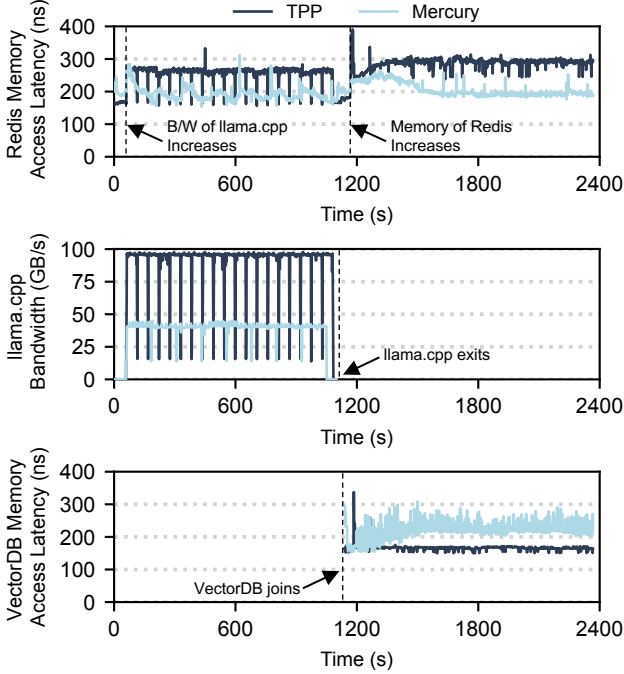
Figure 16: Performance of Redis, llama.cpp, and VectorDB under real-time changes using Mercury. SLO for Redis/llama.cpp/VectorDB is 200ns/180ns/70GBps, with Redis having the highest priority. llama.cpp's bandwidth surges during 60-1100s; Redis's memory usage increases during 1160-2366s.

its SLO, as Redis has higher priority. TPP, on the other hand, gives full bandwidth capacity to llama.cpp, causing severe interference that violates the SLO of Redis. After 1100 seconds, llama.cpp's task finishes. We launch VectorDB and start to gradually increase the load of Redis. The increase in the memory usage of Redis causes local memory contention. Under TPP, Redis cannot acquire more memory due to lower access frequency compared to VectorDB. In contrast, Mercury reallocates memory from VectorDB to Redis to ensure Redis meets its SLO. Mercury achieves up to 8.4× longer SLO satisfaction time for Redis compared to TPP and improved Redis's throughput performance by 33.21%.

6 Related Work

Tiered Memory Systems. Many prior solutions [27, 33, 43, 50] consider non-volatile memory (NVM) as the slow memory tier. HeMem [43] introduces a flexible, per-application memory management policy at the user level. AutoTiering [33] improves system utilization by considering access tier and locality without a predefined threshold. Diverging from NVM-focused methods, Pond [34], TPP [36] and Colloid [47] explore CXL memory; TPP provides an OS-level, application-transparent mechanism for CXL memory, while Pond develops a predictive model for latency and resource management in a CXL-based memory pool. Colloid balances every memory access latency to optimize overall system performance. Mercury decouples memory temperature from application im-

portance, allocating local memory based on SLOs and priority and outperforming previous work that primarily considers applications with hotter pages as more important.

QoS solutions. QoS is a full-stack concern addressed across the hardware/software stack. Memshare [23] maximizes hit rates and provides isolation in multi-tenant web applications with a log-structured, application-aware design. Aequitas [53] and DiffServ [18] prioritize traffic at the network edges, with Aequitas targeting data center environments to ensure latency SLOs for RPCs. TMTS [27] uses two metrics to meet SLOs in tiered memory systems, prioritizing LS applications, but it cannot handle memory bandwidth interference. Mercury introduces QoS management that ensures strict priority and fairness across application types.

Interference management. Prior systems like Heracles [35] and PARTIES [22] dynamically adjust partitions to handle memory bandwidth interference, with PARTIES providing enhanced isolation for memory capacity and disk bandwidth. MCP [37] reduces inter-application interference by mapping data to separate channels, while IMPS [37] prioritizes memory non-intensive applications, which can be unfair. The FQ memory scheduler [39] prioritizes memory requests by earliest virtual finish-time. ASM [45] minimizes memory interference by periodically giving each application's requests the highest priority. However, these systems do not address inter-tier interference. Mercury is the first to combine local memory limits and CPU utilization to avoid both intra-tier and inter-tier bandwidth interference.

Disaggregated Memory. Memory disaggregation exposes capacity available in remote hosts as a shared memory pool. Recent RDMA-based disaggregated memory solutions [19, 28, 30, 48] face significantly higher latency than CXL memory [29]. Memory management in these systems is orthogonal to Mercury; one can use both CXL- and network-enabled memory tiers and apply Mercury to manage tiered memory.

7 Conclusion

Tiered memory systems provides higher memory capacity to allow more memory-intensive applications to be deployed. However, existing tiered memory systems focus on optimizing a single application, and cannot provide QoS guarantees when multiple applications sharing memory resource. We present the design and implementation of Mercury, a QoS-aware tiered memory system to provide predictable performance for coexisting memory-intensive workloads. Mercury provides application-level resource management by enabling per-tier page reclamation inside Linux kernel. By designing a novel admission control algorithm, Mercury maximizes local memory utilization while mitigate both intra-tier and inter-tier memory bandwidth interference. Mercury outperforms the state-of-the-art solution when handling local memory contention, memory interference, and dynamic workload changes with significant performance improvement.

References

[1] HiBench Suite. https://github.com/Intelbigdata/HiBench.

[2] PostgreSQL Database Management System. https://www.postgresql.org/.

[3] SPEC CPU 2017. https://www.spec.org/cpu2017/.

[4] TPC Benchmark C (TPC-C). http://www.tpc.org/tpcc/.

[5] TPC Benchmark H (TPC-H). https://www.tpc.org/tpch/.

[6] Weighted interleaving for memory tiering. https://lwn.net/Articles/948037/.

[7] Numa balancing. https://mirrors.edge.kernel.org/pub/linux/kernel/people/andrea/autonuma/autonuma_bench-20120530.pdf, 2012.

[8] Linux cgroups. https://docs.kernel.org/admin-guide/cgroup-v2.html, 2015.

[9] Intel platform qos technologies. https://wiki.xenproject.org/wiki/Intel_Platform_QoS_Technologies, 2018.

[10] Pebs. intel 64 and ia-32 architectures software developer's manual. https://software.intel.com/articles/intel-sdm/, 2023.

[11] Compute express link (cxl). https://www.computeexpresslink.org/, 2024.

[12] Inference of meta's llama model (and others) in pure c/c++. https://github.com/ggerganov/llama.cpp/, 2024.

[13] memcached. https://memcached.org/, 2024.

[14] Redis. https://github.com/redis/redis/, 2024.

[15] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *ASPLOS*, 2017.

[16] AMD. μProf User Guide. 2024.

[17] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[18] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, , and W. Weiss. Rfc2475: An architecture for differentiated service. In *IETF*, 1998.

[19] Irina Calciu, M Talha Imran, Ivan Puddu, Sanidhya Kashyap, Hasan Al Maruf, Onur Mutlu, and Aasheesh Kolli. Rethinking software runtimes for disaggregated memory. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 79–92, 2021.

[20] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin Levandoski, James Hunter, and Mike Barnett. Faster: A concurrent key-value store with in-place updates. In *Proceedings of the 2018 International Conference on Management of Data*, pages 275–290, 2018.

[21] Lei Chen, Shi Liu, Chenxi Wang, Haoran Ma, Yifan Qiao, Zhe Wang, Chenggang Wu, Youyou Lu, Xiaobing Feng, Huimin Cui, et al. A tale of two paths: Toward a hybrid data plane for efficient far-memory applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 77–95, 2024.

[22] Shuang Chen, Christina Delimitrou, and José F. Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *ASPLOS*, 2019.

[23] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. Memshare: a dynamic multi-tenant key-value cache. In *ATC*, 2017.

[24] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154, 2010.

[25] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library. 2024.

[26] Paul J. Drongowski. Instruction-based sampling: A new performance analysis technique for amd family 10h processors. 2007.

[27] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David Culler, Zhiyi Xu, Jianing Fan, Chris Kennelly, Bill McCloskey, Danijela Mijailovic, Brian Morris, Chiranjit Mukherjee, Jingliang Ren, Greg Thelen, Paul Turner, Carlos Villavieja, Parthasarathy Ranganathan, and Amin Vahdat. Towards an adaptable systems architecture for memory tiering at warehouse-scale. In *ASPLOS*, 2023.

[28] Yixiao Gao, Qiang Li, Lingbo Tang, Yongqing Xi, Pengcheng Zhang, Wenwen Peng, Bo Li, Yaohui Wu, Shaozong Liu, Lei Yan, et al. When cloud storage meets RDMA. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pages 519–533, 2021.

[29] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, High-Performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294, 2022.

[30] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with Infiniswap. In *NSDI*, 2017.

[31] Rishabh Jain, Scott Cheng, Vishwas Kalagi, Vrushabh Sanghavi, Samvit Kaul, Meena Arunachalam, Kiwan Maeng, Adwait Jog, Anand Sivasubramaniam, Mahmut Taylan Kandemir, et al. Optimizing cpu performance for recommendation systems at-scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[32] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos: Os design for heterogeneous memory management in datacenter. In *ISCA*, 2017.

[33] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for multi-tiered memory systems. In *ATC*, 2021.

[34] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: Cxl-based memory pooling systems for cloud platforms. In *ASPLOS*, 2023.

[35] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *ISCA*, 2015.

[36] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. TPP: Transparent page placement for CXL-enabled tiered memory. In *ASPLOS*, 2023.

[37] Sai Prashanth Muralidhara, Lavanya Subramanian, Onur Mutlu, Mahmut Kandemir, and Thomas Moscibroda. Reducing memory interference in multicore systems via application-aware memory channel partitioning. In *2011 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 374–385, 2011.

[38] Maxim Naumov, Dheevatsa Mudigere, Hao-Jun Michael Shi, Jianyu Huang, Narayanan Sundaraman, Jongsoo Park, Xiaodong Wang, Udit Gupta, Carole-Jean Wu, Alisson G. Azzolini, Dmytro Dzhulgakov, Andrey Mallevich, Ilia Cherniavskii, Yinghai Lu, Raghuraman Krishnamoorthi, Ansha Yu, Volodymyr Kondratenko, Stephanie Pereira, Xianjie Chen, Wenlin Chen, Vijay Rao, Bill Jia, Liang Xiong, and Misha Smelyanskiy. Deep learning recommendation model for personalization and recommendation systems. *CoRR*, abs/1906.00091, 2019.

[39] 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, 2006.

[40] Khang T Nguyen. Introduction to memory bandwidth monitoring in the Intel Xeon processor e5 v4 family. 2016.

[41] Yuanjiang Ni, Pankaj Mehra, Ethan Miller, and Heiner Litz. Tmc: Near-optimal resource allocation for tiered-memory systems. In *SoCC*, 2023.

[42] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, et al. Renaissance: A modern benchmark suite for parallel applications on the jvm. In *SPLASH*, 2019.

[43] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. In *ASPLOS*, pages 392–407, 2021.

[44] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. vtmm: Tiered memory management for virtual machines. In *EuroSys*, 2023.

[45] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. The application slow-down model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory. In *MICRO*, 2015.

[46] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[47] Midhul Vuppalapati and Rachit Agarwal. Tiered memory management: Access latency is the key! In *SOSP*, 2024.

[48] Chenxi Wang, Haoran Ma, Shi Liu, Yuanqi Li, Zhenyuan Ruan, Khanh Nguyen, Michael D Bond, Ravi Netravali, Miryung Kim, and Guoqing Harry Xu. Semeru: A Memory-Disaggregated managed runtime. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 261–280, 2020.

[49] Chenxi Wang, Yifan Qiao, Haoran Ma, Shi Liu, Wen-guang Chen, Ravi Netravali, Miryung Kim, and Guo-qing Harry Xu. Canvas: Isolated and adaptive swapping for Multi-Applications on remote memory. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*, pages 161–179, 2023.

[50] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *ASPLOS*, 2023.

[51] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. MT^2: Memory Bandwidth Regulation on Hybrid NVM/DRAM Platforms. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 199–216, 2022.

[52] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A Fault-Tolerant abstraction for In-Memory cluster computing. In *9th USENIX symposium on networked systems design and implementation (NSDI 12)*, pages 15–28, 2012.

[53] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. Aequitas: Admission control for performance-critical rpcs in datacenters. In *SIGCOMM*, 2022.

[54] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing memory tiers with cxl in virtualized environments. In *Symposium on Operating Systems Design and Implementation*, 2024.

A Workloads

We use a total of 80 real-world applications:

- **Database:** TPC-C [4] on Silo [46] and TPC-H [5] on PostgreSQL [2] and Faiss [25]
- **Machine learning (ML):** DLRM benchmark [31, 38] and llama.cpp [12]
- **Key-value (KV) store:** YCSB [24] on FASTER [20], Redis [14]
- **Big data:** HiBench [1] on Spark [52]
- **Graph processing:** GAP benchmark [17]
- **Scientific computing:** SPEC CPU 2017 [3]
- **Web:** Renaissance [42]