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

Tiered memory systems have widely been adopted to provide larger memory capacity in response to increasing memory demands from memory-intensive workloads. Although increased memory capacity allows more applications to be deployed, existing solutions for tiered memory systems are not built with Quality-of-Service (QoS) support. As a result, they often cannot meet service-level objectives (SLOs) when multiple applications share a tiered memory system. Specifically, applications suffer from *local memory contention* and *memory bandwidth interference*, two sources of performance unpredictability unique to tiered memory systems. Indeed, we observe application performance drops by 43% and 70% during severe memory contention and interference.

This paper presents Mercury, a QoS-aware tiered memory system that provides predictable performance for coexisting memory-intensive applications, each with different SLOs. Mercury enables per-tier page reclamation to enforce application-level resource management. It leverages a novel admission control and real-time adaptation algorithm to maximize local memory utilization while mitigating memory interference. Evaluations with real-world applications show that Mercury can provide QoS guarantees among multiple applications sharing a tiered memory system with up to 53.4% improvement in performance.

1 Introduction

With the increasing memory demands of datacenter applications, tiered memory systems have widely been adopted to replace DRAM-only systems [18, 26, 29, 35, 36, 40]. Many recent tiered memory systems are enabled by Compute Express Link (CXL) [6], which is a high-speed interconnect standard that allows memory capacity to grow by attaching more 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 [8, 9] that require low-latency memory access, and (2) *bandwidth-intensive (BI)* applications such as large-memory ML models (e.g., long-context language models or recommendation models [7, 32]) that require a sustained memory bandwidth.

In production environments, running a single application on a tiered memory system wastes both memory capacity and bandwidth. As a result, multiple memory-intensive applications often have to share the tiered memory system to maximize resource utilization and improve total cost of ownership (TCO) [18, 26, 29, 34, 36].

Existing research on tiered memory systems primarily focuses on page temperature monitoring and efficient page migration to better utilize local memory resources (i.e., fast-tier DRAM) [10, 23, 24, 29, 35, 40]. However, these solutions optimize a single application running on a single server. They are not built with Quality-of-Service (QoS) support and thus cannot react to applications with different service level objectives (SLOs). In particular, existing tiered memory systems suffer from great performance unpredictability due to two reasons. 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 grab more local memory than a high-priority application, leading to *priority inversion* (§2.1). Second, high memory bandwidth generated by BI applications can hurt 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 bandwidth interference across tiered memory, including the recent proposal on QoS support for tiered memory [18].

In this paper, we aim to provide predictable performance for applications with different SLOs in the presence of local memory contention and memory bandwidth interference. However, achieving this goal faces the following unique challenges. First, it requires an efficient way to track and control memory resources on each memory tier, but the memory control mechanisms in existing operating systems (OSes) are not designed for tiered memory. Second, it is non-trivial to determine the amount of memory to allocate to each application in each tier while simultaneously maximizing resource utilization to accommodate more applications. For example, in the case of a BI application with a loose SLO, although migrating some of its local memory to CXL memory can save local memory for another LS application, too much migration may incur inter-tier interference, leading to SLO violations. Finally, an application's workload can change after its deployment, calling for an efficient approach toward performance monitoring and real-time adaptation.

We present Mercury, a QoS-aware tiered memory system that provides predictable performance for memory-intensive applications. Mercury incorporates the following key ideas to overcome the aforementioned challenges:

*Equal contributions.

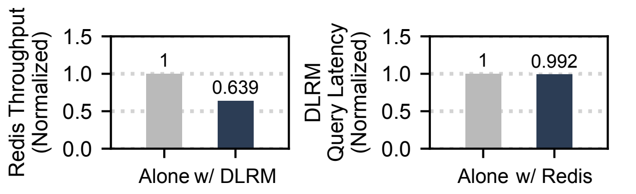


Figure 1. Unpredictable performance of Redis and DLRM as they compete for local memory on the fast tier. Existing solutions cannot distinguish among applications when migrating their hot pages, and thus cannot provide QoS guarantees.

(1) Mercury enforces application-level resource management by enabling *per-tier page reclamation*. This allows Mercury to control an application's available local memory while still leveraging existing page migration designs.

(2) Mercury includes a novel *admission control* algorithm to determine the right amount of resources for applications to satisfy SLOs with the goal of maximizing local memory utilization while mitigating memory bandwidth interference.

(3) Mercury provides real-time adaptation to unpredictable memory and bandwidth changes, preventing sudden system burdens while maximizing the number of applications meeting their SLOs based on priority.

We evaluate Mercury's effectiveness in providing QoS using real-world applications and find that 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, achieving up to 53.4% better application performance than TPP while satisfying more applications' SLO. Mercury also achieves 8.4× longer SLO satisfaction time than TPP in a long-running experiment to handle real-time workload changes.

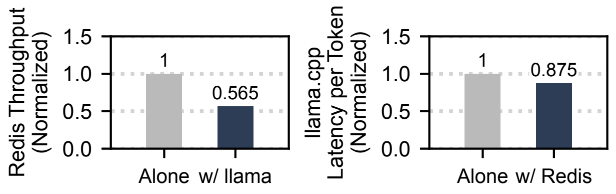
We make the following research contributions:

- We conduct a thorough QoS analysis on production-ready tiered memory systems, including local memory contention and memory bandwidth interference, and draw new observations.
- 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.

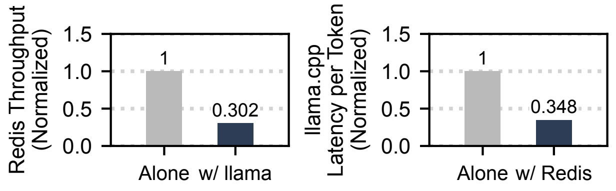
2 Lack of QoS Support in Tiered Memory

We start by introducing why existing multi-tiered memory systems lack QoS support. There are two root causes for unpredictable performance when multiple applications coexist – local memory contention (§2.1) and memory bandwidth interference (§2.2).

Characterizing applications. In this work, we classify applications into two types: (1) *Latency-sensitive (LS)* applications that desire low memory access latency, and (2)



(a) Intra-tier Interference



(b) Inter-tier Interference

Figure 2. llama.cpp's memory bandwidth creates interference, resulting in a significant drop in the throughput performance of Redis. (a) shows intra-tier interference when llama.cpp is on the same fast tier with Redis. (b) shows inter-tier interference after all llama.cpp's memory is migrated to CXL.

bandwidth-intensive (BI) applications that require sustained memory bandwidth. Both types of applications can be deployed onto the same server to increase system-level memory utilization. We do not assume LS applications are always more important than BI applications, and vice versa.

Hardware Specifications. Our motivating experiments run on dual-socket AMD Genoa servers each having 96 physical CPU cores and 12 channels of DDR5 memory per socket. CXL memory is enabled via memory expansion cards with four channels of DDR4 memory. The total memory footprint of a single server is 1.8TB; each socket has 768GB DDR5 and CXL memory has 256GB DDR4 memory.

2.1 Local Memory Contention

The core design principle of multi-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 [10, 18, 23, 24, 29, 35, 40]. However, as multiple applications have to share tiered memory to fully utilize the local memory and bandwidth capacity, such a design does not distinguish among applications with different SLO targets, and thus can not provide QoS guarantees. For example, a low-priority application with a large amount of hot memory can still compete for local memory, hurting the performance of more important applications. Figure 1 illustrates this issue with two real-world applications, Redis [9] and Deep Learning Recommendation Model (DLRM) [32] on a real CXL-based tiered memory setup (without GPUs) used in a production-ready cluster. We enable TPP [29], the state-of-the-art multi-tiered memory system, to handle the page migration mechanism between local memory (the fast tier) and CXL memory (the slow tier). We configure the applications such that the sum of their working set size (WSS) is greater than the available local memory on the fast tier.

The throughput of Redis drops by 36% due to insufficient local memory, as DLRM is competing for local memory at the same time. An ideal QoS-aware memory system should be able to *allocate the right amount of local memory* to each of the applications based on their QoS target.

2.2 Memory Bandwidth Interference

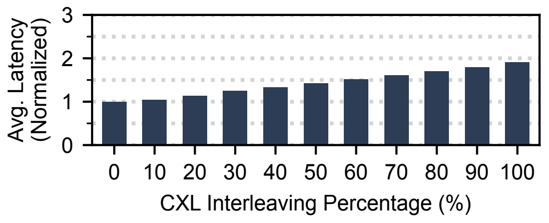
Another source of performance unpredictability is memory bandwidth interference, where applications with high memory bandwidth affect the performance of coexisting latency-sensitive applications. The problem is further complicated in tiered memory systems, where applications can generate bandwidth with 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 [14, 27, 31, 33]. Inter-tier interference occurs when excessive memory requests on one memory tier cause a slowdown of requests on another tier. Figure 2 illustrates both types of interference by colocating llama.cpp [7] inference tasks (without GPUs) and Redis. Redis's throughput degrades by 43.5% when llama.cpp is generating a high memory bandwidth on the same fast tier in Figure 2a. We then migrate all llama.cpp's memory to CXL memory and observe even worse Redis performance (dropped by 70%) due to inter-tier inference (Figure 2b).

Existing solutions on mitigating memory bandwidth interference work either (1) on the memory request level, such as memory request prioritization [33] or memory channel partitioning [31], or (2) on the CPU core level to throttle memory bandwidth from best-effort tasks [14, 27]. The request scheduling approaches assume a single-tier memory tier and do not work across multiple tiers. The CPU throttling approaches are agnostic to multiple memory tiers and thus cannot resolve inter-tier interference.

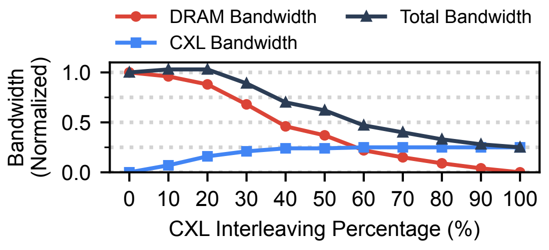
Mitigating interference plays an important role in providing SLO guarantees for LS applications. In the meantime, we also want to provide QoS for BI applications and meet their bandwidth SLO whenever possible. An ideal solution should mitigate interference by migrating pages off the tier experiencing interference while leveraging additional bandwidth in other tiers to satisfy SLOs of bandwidth-intensive applications. However, such migration is not straightforward given the existence of inter-tier interference, as we will show in our detailed analysis in the following section.

3 QoS Analysis in Tiered Memory

In this section, we quantitatively analyze how local memory contention and memory bandwidth interference affect application performance and draw key insights to design Mercury. All results are collected on the same CXL-based tiered memory system as in Section 2.



(a) Impact on Latency Performance



(b) Impact on Bandwidth Performance

Figure 3. Latency and bandwidth performance at different CXL interleaving ratios to illustrate the impact of local memory.

3.1 Impact of Available Local Memory

We start by studying the impact of available local memory on application performance, as applications can easily run short of local memory during memory contention. We develop two microbenchmarks to represent LS and BI applications, which we denote as *LS* and *BI* in Section 3 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 3 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 CXL 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.

3.2 Deep Dive in Memory Interference

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

Varying *BI* across tiers. In this experiment, we configure *LS* to always access local memory and vary *BI*'s CXL-interleaving percentage. The results are shown in Figure 4, and we make two key observations. First, as more bandwidth

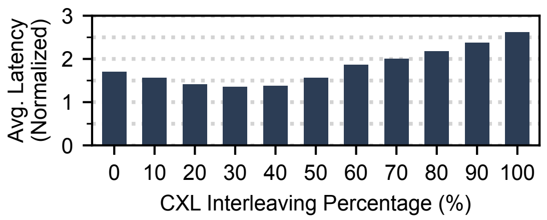


Figure 4. Performance of *LS* when *BI* is generating bandwidth at different CXL interleaving percentage. Migrating *BI* to CXL does not always lead to better performance of *LS* due to inter-tier interference. *BI*'s performance is very close to Figure 3b and omitted in the interest of space.

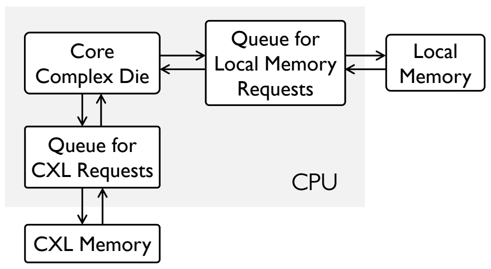


Figure 5. Architectural diagram of how memory requests are handled in CXL-based tiered memory.

is generated on CXL, instead of a monotonic change, the performance of the LS application initially decreases and then increases. Second, the interference becomes the worst when all memory bandwidth is generated from CXL, even though the total memory bandwidth is the lowest at this moment.

To explain the behavior in the last experiment, we need to understand how memory requests are handled in a tiered memory system. Figure 5 presents a high-level diagram. There exist separate fixed-size queues in the hardware for memory requests to/from local and CXL memory. Initially, when bandwidth on 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 6 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,

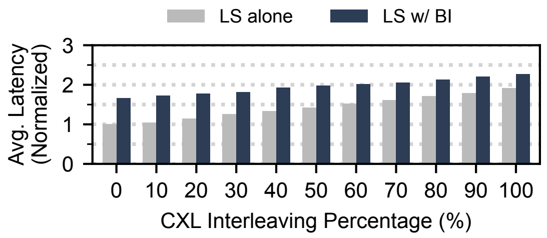


Figure 6. Performance of *LS* at different CXL interleaving ratios when *BI* is fixed on local memory. Migrating more requests away from local memory does not improve performance as more requests are accessing the slower tier.

such an 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.

4 Mercury Overview

With multiple applications sharing a tiered memory, our goal in this work is to provide predictable performance among applications with different SLOs under memory resource contention and memory interference. We first discuss our design principles, followed by a system overview of Mercury.

4.1 Design Principles

QoS granularity. The first question we ask is *at what granularity should we 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 on *the application level* due to (1) easy deployment without modifying applications, (2) a clean QoS interface, and (3) low overhead in managing memory resources and performance monitoring. Moreover, providing QoS on the application level also allows us to reuse existing kernel mechanisms for identifying and migrating hot and cold pages, which has proved to be efficient [29].

Choice of performance indicators. Selecting the right performance indicators is critical for providing QoS, as Mercury needs to react quickly to meet applications' requirements. In Mercury, we prefer *low-level metrics* as performance indicators compared to application-level performance 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 (§6). In particular, we collect *memory access latency and memory bandwidth* per application. Memory access latency is measured using memory load events from

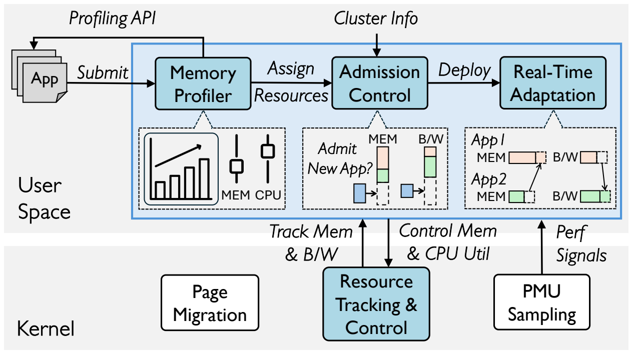


Figure 7. High-level system overview of Mercury. The memory profiler and the admission control determine the right resource to allocate for applications. Real-time adaptation dynamically adjusts resource allocation during runtime. The resource controller tracks and controls resources at the application level.

L3 cache misses provided by AMD processor's Instruction Based Sampling (IBS) [17] mechanism. On the AMD platform, bandwidth is measured by μProf [11]. For intel platforms, Mercury can be applied through processor event-based sampling (PEBS) [5] where bandwidth can be measured via Intel Platform QoS (PQoS) [4].

Memory resource allocation. Instead of overprovisioning resources to handle worst-case overload scenarios, we decide to dynamically allocate *the right amount of resource that can satisfy an application's SLO*. This allows Mercury to accommodate more applications while meeting their SLO. 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 allocation for applications when runtime workload changes affect QoS.

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/or 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 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.

4.2 System Overview

Figure 7 presents a system diagram of Mercury. Its design includes a profiler, an admission control algorithm, and real-time adaptation in the user space, as well as a memory resource controller implemented inside the Linux kernel.

[1]Lower priority applications that yield (part of) their resources to higher priority ones continue to run as best-effort to preserve work conservation.

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** on 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 (§5.1). Mercury adopts existing Linux page LRUs and NUMA Balancing [1] for page temperature detection and migration.

When Mercury receives the application, it first profiles offline to find the minimum amount of memory resource needed to satisfy the SLO when running in isolation (§5.2). The profiler also provides users with an API to measure the two SLO metrics (i.e., memory access latency and memory bandwidth) to help users 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 application's priority level (§5.3). Mercury 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 change and bandwidth interference (§5.4).

5 Mercury Design

This section describes the details of Mercury design. We start with its application-level resource management, followed by how Mercury provides QoS with its memory profiling, admission control algorithm, and real-time adaptation.

5.1 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 v.s. 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 adjust its bandwidth, as accessing more memory from a slower tier decreases the aggregate bandwidth. 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 (§3.2). Second, the bandwidth cannot be tuned further down after migrating all pages to CXL memory (Figure 3b).

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. The existing Linux kernel provides cgroup [3] 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, as it does not distinguish among 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.

Modifications to 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. In this regard, we enhance the existing cgroup to track the list of pages within each LRU associated with the NUMA nodes across a specific memory tier. We, therefore, 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`, whereas the latter controls the total system-wide memory usage of an application. In contrast, 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 [29], we leverage NUMA Balancing [1] to allow page promotion among different nodes.

Mercury controls CPU utilization of an application using native cgroup's `cpu.max` interface, whose value adjusts the utilization among all CPU cores the application uses. Details of how Mercury selects an application's local memory limit and CPU utilization to provide QoS will be described next.

5.2 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 for admission control (§5.3) to determine the 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.

Mercury also performs a one-time profiling per machine to characterize 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 (§3) 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).

5.3 Admission Control

Admission control ensures that admitted applications can coexist well in a shared tiered memory system. A naïve 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 has two drawbacks. First, a critical application may arrive late, only to find the resources have been taken by other less critical applications. 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*

Algorithm 1: Admission Control

```
Notation:
  $pQueue$: queue of apps in *ascending* order of priority
  $B_{slo,s}$: app $s$' b/w SLO, $M_{profile,s}$: app $s$' profiled local memory
  $\alpha$: memory update granularity, $\beta$: CPU update granularity
Function DeployApp($s$):
if !IsAdmissible(s) then
    NotAdmit($s$)
▷ Assign local memory for LS & BI apps
$v \leftarrow pQueue.top()$
while $v \neq s$ and GetLocalMemAvail() $< M_{profile,s}$ do
    YieldMem($v, M_{profile,s}$ − GetLocalMemAvail())
    while (GetNumaHintFaultRate() > Thresh$_{numa}$) do
        ReduceLowerPriorityAppBw($pQueue, s$)
    $v \leftarrow v$.next()
$s.mem\_limit \leftarrow \min($GetLocalMemAvail()$, M_{profile,s})$
▷ Assign local memory & cpu util for BI apps
if $s.type$ == "$BI$" then
    while GetAppUsableBw(s) $< B_{slo,s}$ do
        if ReduceLowerPriorityAppBw($pQueue, s$) then
            break
    $s.mem\_bw \leftarrow \min(B_{slo,s}$, GetAppUsableBw(s))
LaunchApp($s$)
$pQueue.enqueue(s)$

Function ReduceLowerPriorityAppBw($pQueue, s$):
$v \leftarrow pQueue.top()$
while $v \neq s$ do
    if $v.type$ == "$LS$" or (!$v.cpu\_util$ and !$v.mem\_limit$) then
        $v \leftarrow v$.next()
    if !IsInterTierHealthy($s$) or !$v.mem\_limit$ then
        $v.cpu\_util \leftarrow \max(0, v.cpu\_util - \beta)$
    else
        $v.mem\_limit \leftarrow \max(0, v.mem\_limit - \alpha)$
    return true
return false
```

local memory utilization to admit more applications while *avoiding intra-tier and inter-tier interference.*

Algorithm 1 describes Mercury's admission control. At the arrival of a new application, Mercury first checks if it is labeled as inadmissible by the profiler. If that is the case, Mercury immediately drops this application since its SLO (either latency or bandwidth) can never be met (lines 3-4). Otherwise, Mercury starts to assign resources based on the application type, as we describe below.

Admitting LS applications. Mercury directly admits an LS application if there is available local memory to satisfy its profiled local memory requirement. Otherwise, Mercury tries to find another existing application with a lower priority to *yield local memory*, starting from the one with the lowest priority (lines 7-8). However, yielding another application's memory means moving more of its requests to CXL memory, which has a risk of causing *inter-tier interference* when the victim is bandwidth-intensive (§3.2). Therefore, Mercury monitors the rate of NUMA hint faults and compares it with $Thresh_{numa}$ obtained from profiling (§5.2). If the threshold is met, Mercury stops decreasing the local memory limit of the victim application, as migrating more of its memory to CXL memory will cause inter-iter interference that harms latency performance. Instead, Mercury will reduce CPU utilization for BI apps with lower priority than the incoming app, in ascending order, until we are below $Thresh_{numa}$ (lines 9-10).

Mercury repeats this process until enough memory is yielded. 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.

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 (lines 6-12). 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 to *yield bandwidth*. This is done by decreasing the local memory limit of the victim application(s). During this process, Mercury checks 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 (lines 16, 26-27). In the meantime, Mercury watches out for potential *intra-tier interference* when bandwidth on the fast-tier becomes too high. This is verified by checking two conditions: (1) there exist LS applications with higher priority, and (2) the healthy bandwidth threshold ($thresh_{local\_bw}$, §5.2) for the fast tier is exceeded. If both are met, Mercury stops assigning bandwidth on the fast tier to the incoming application (line 15, details omitted in Algorithm 1 for brevity).

5.4 Real-time Adaptation

Although our memory profiling and admission control provide a good estimation of how many resources to assign to applications during launch time, we may still experience SLO violations in two scenarios. First, the amount of resource applications need to maintain their SLO can change over time due to workload changes. Second, LS applications may still be affected by minor interference when BI applications' bandwidth sits around one of the two interference thresholds. Mercury should adjust resource allocations quickly to ensure QoS guarantees for critical applications in such scenarios.

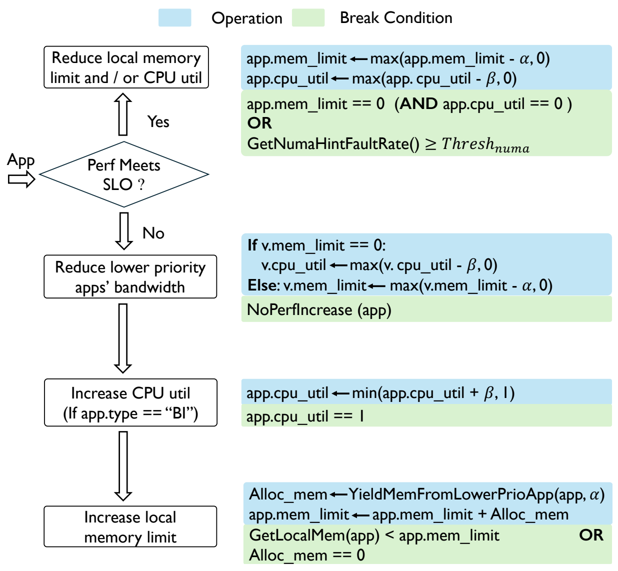

Operation
Break Condition
Reduce local memory limit and / or CPU util
app.mem_limit ← max(app.mem_limit - α, 0)
app.cpu_util ← max(app. cpu_util - β, 0)
app.mem_limit == 0 (AND app.cpu_util == 0)
OR
GetNumaHintFaultRate() ≥ Thresh_numa
Yes
App
Perf Meets SLO ?
No
Reduce lower priority apps' bandwidth
If v.mem_limit == 0:
v.cpu_util ← max(v. cpu_util - β, 0)
Else: v.mem_limit ← max(v.mem_limit - α, 0)
NoPerfIncrease (app)
Increase CPU util (If app.type == "BI")
app.cpu_util ← min(app.cpu_util + β, 1)
app.cpu_util == 1
Increase local memory limit
Alloc_mem ← YieldMemFromLowerPrioApp(app, α)
app.mem_limit ← app.mem_limit + Alloc_mem
GetLocalMem(app) < app.mem_limit OR
Alloc_mem == 0

Figure 8. Procedure of real-time adaptation.

Figure 8 describes the flow of Mercury's real-time adaptation. Mercury periodically checks the performance signals (i.e., memory access latency for LS applications, and memory bandwidth for BI applications) every 200ms and verifies if they meet SLO. To ensure critical applications receive guaranteed performance, we check applications in the order of higher priority levels. If the measured performance of the target application is better than SLO, Mercury retrieves excessive resources by decreasing its local memory limit until the performance is back to expectation again. Similar to our admission control, Mercury watches out for inter-tier interference during this process (by checking $thresh_{numa}$) to prevent too many requests from accessing CXL memory. In the case of a BI application, Mercury starts to decrease its CPU utilization when its local memory limit hits zero.

It is more challenging when the measured performance is worse than SLO, as there may be multiple possible causes. Mercury takes a series of actions to identify the right cause and make adjustments. Mercury first decreases the bandwidth of low-priority applications until no performance improvement can be detected. This step verifies if the performance drop is caused by bandwidth interference. Since all applications go through our real-time adjustment in the order of priority, 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. After the bandwidth check, if the measured performance is still worse than SLO, we are certain the target application needs more resources due to a workload change. In this case, Mercury allocates resources based on the type of the target application. Given a BI application, Mercury first increases its CPU utilization before increasing its local memory limit to ensure local memory is maximized among all applications. For an LS application, Mercury directly increases its local memory limit. Similar to the way in admission control, Mercury finds the local memory from low-priority applications in ascending order of the priority level.

Table 1. Four real-world applications used in evaluations.

Application	Application Level Metric	Type
Redis	Operations Per Second	LS
vectorDB	Latency per Query	LS
llama.cpp	Latency Per Token	BI
DLRM	Queries Per Second	BI

6 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 (§6.2).
(2) Mercury effectively handles local memory contention and bandwidth interference, achieving up to a 53.4% improvement compared to TPP (§6.3- §6.5).
(3) Mercury's real-time adaptation mechanism accurately reallocates resources to prioritize critical applications during runtime workload changes, resulting in 8.4× longer SLO satisfaction time compared to TPP (§6.6).

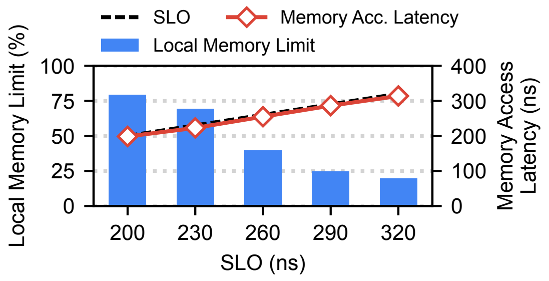
6.1 Experiment Setup

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.

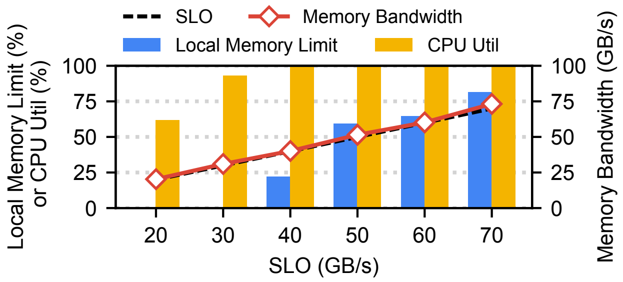
We illustrate Mercury's effectiveness in providing QoS using four real-world applications (Table 1):

- *Redis (LS)*: a widely used in-memory database. We launch memtier_benchmark [2] to generate a realistic workload.
- *vectorDB (LS)*: a vector database implemented using the Faiss library [16] with FlatL2 [25] and HNSWFlat [28] indices. Each client request contains a query of 10 nearest neighbors of randomly generated vectors.
- *DLRM (BI)*: a popular deep learning recommendation model, running on Criteo Kaggle Display Advertising Challenge dataset [22]. Compared to the setting in §2.1 which targets low serving latency, we use another practical throughput-intensive setup where providers use a larger number of embeddings, causing high bandwidth.
- *llama.cpp (BI)*: a C/C++ implementation of inference for large language models including LLaMA [38]. We run llama.cpp on local and CXL memory with the Llama 2 70B - GGUF model [30].

Emulation environment. We run experiments on dual-socket Intel Xeon Gold 6330 servers with 56 physical CPU cores. Each socket contains 512GB DDR4 memory. We emulate CXL by disabling all cores in one socket while keeping its memory accessible from the other socket, a method widely adopted in current CXL research [26, 29]. In the meantime,



(a) SLO compliance for Redis



(b) SLO compliance for llama.cpp

Figure 9. Mercury provides SLO compliance on both memory access latency (for Redis) and bandwidth (for llama.cpp).

we reduce the uncore frequency on the server to match the memory access latency from the isolated socket to the latency of accessing the same CXL setup in §2 and §3.

We compare all the experiments with TPP [29], a state-of-the-art open-source tiered memory system. Each experiment is run five times, and the average measurements are reported.

6.2 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 9a 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 for BI applications. Figure 9b 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 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.

6.3 Handling Local Memory Contention

Next, we move on to multi-tenant settings and evaluate how Mercury handles local memory contention. This experiment involves a high-priority Redis process running together with

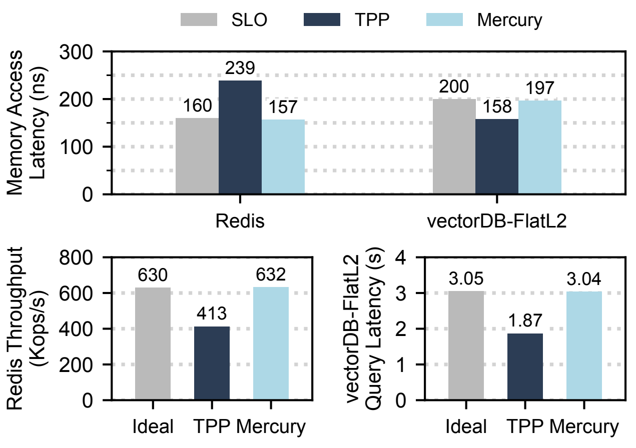


Figure 10. Comparing Mercury with TPP when handling local memory contention between Redis and vectorDB.

a low-priority vectorDB-FlatL2 workload configured as an LS application. We set the SLO of Redis and vectorDB to be 160ns and 200ns, respectively. The WSS of Redis and vectorDB are both set to 20GB, and we configure the local memory capacity to be 20GB as well to create local memory contention. Figure 10 compares the average memory access latency and application-level performance between TPP and Mercury. TPP fails to achieve the SLO for Redis, as TPP cannot prevent vectorDB from stealing most of the local memory due to higher memory access frequency than Redis, leading to priority inversion. In comparison, Mercury manages to achieve the SLO for both applications by assigning the right amount of local memory accordingly (18GB for Redis and 2GB for vectorDB), leading to 46% improvement in the throughput of Redis over TPP.

6.4 Handling Memory Bandwidth Interference

Besides local memory contention, memory bandwidth interference is another key aspect Mercury tries to resolve when designing its admission control. We evaluate Mercury's effectiveness in handling bandwidth interference among different combinations of applications. In those experiments, we configure the WSS of applications such that local memory contention never happens. We start with a high-priority Redis (20GB WSS) workload running alongside a low-priority llama.cpp inference task, and record both low-level and application-level performance of both applications in Figure 11. TPP has no control over Mercury's bandwidth, which causes significant interference on Redis, leading to missed SLO and worse throughput. Mercury, in contrast, mitigates bandwidth interference from llama.cpp by migrating most of its memory to CXL memory (with 25.8% memory left on local), ending up with both applications' SLO satisfied and 32.7% higher throughput of Redis compared to TPP.

We observe a similar trend when launching Redis (10GB WSS) together with DLRM (Figure 12). Similar to llama.cpp,

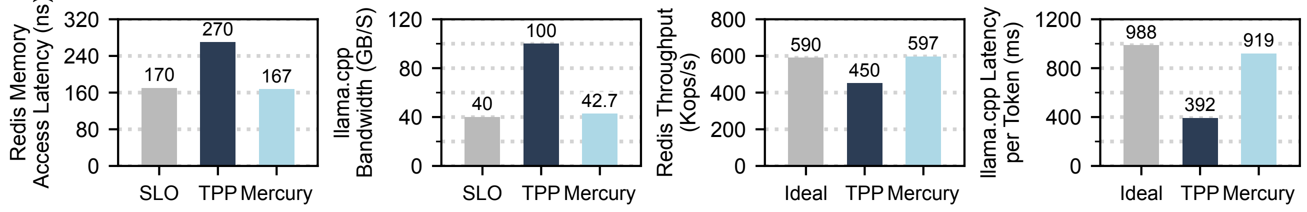


Figure 11. Comparing Mercury with TPP when handling memory bandwidth interference between Redis and llama.cpp.

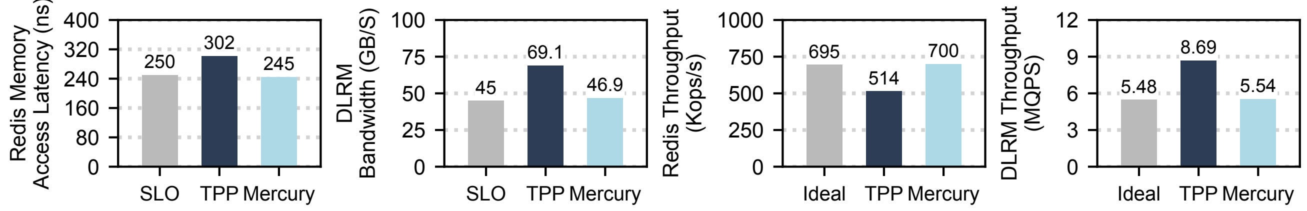


Figure 12. Comparing Mercury with TPP when handling memory bandwidth interference between Redis and DLRM.

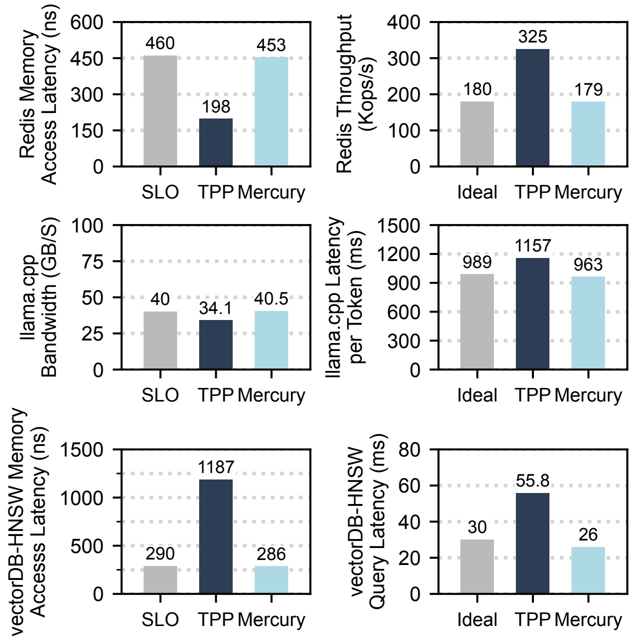


Figure 13. Comparing Mercury with TPP when handling both local memory contention and memory bandwidth interference among Redis, llama.cpp, and vectorDB.

DLRM causes bandwidth interference that TPP cannot mitigate. Mercury adjusts DLRM's CPU utilization (64%) and local memory limit (50% of its WSS) to meet DLRM's SLO while minimizing both intra-tier and inter-tier interference, resulting in a 36.2% improvement in the throughput of Redis.

6.5 Mixing Two Sources of Unpredictability

It is quite likely that local memory contention and memory bandwidth interference appear simultaneously in a real-world setting. To evaluate how Mercury performs in this case, we deploy Redis, llama.cpp, and vectorDB-HNSW 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, whereas 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.

6.6 Real-time Adaptation to Dynamic Changes

In practice, the memory or bandwidth usage of applications often changes over time. To justify Mercury's adaptability, we run Redis, llama.cpp, and vectorDB-FlatL2 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 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

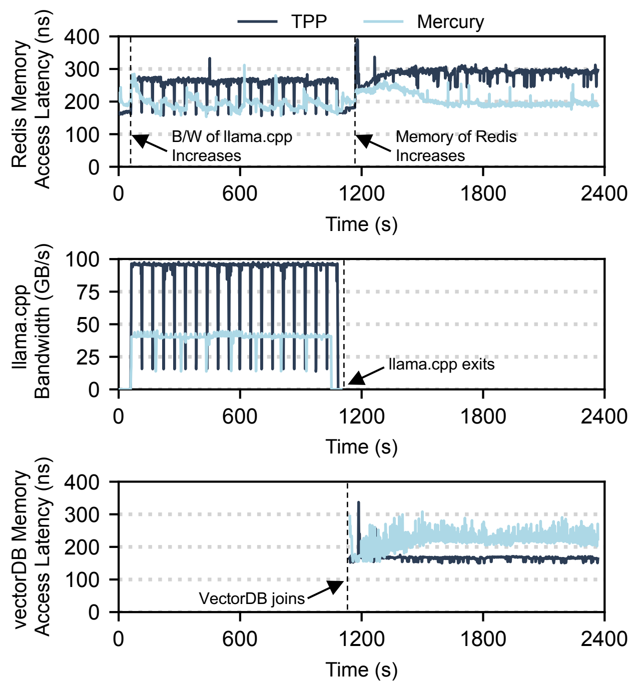


Figure 14. Performance of Redis, llama.cpp, and vectorDB under real-time changes. SLO for Redis/llama.cpp/vectorDB is 200ns/180ns/70GBps, with Redis having the highest priority. llama.cpp's bandwidth surges during 60-1100s; Redis's memory usage increases during 1160-2366s.

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. Throughout this experiment, Mercury results in up to 8.4× longer SLO satisfaction time for Redis compared to TPP and improved Redis's throughput performance by 33.21%.

7 Related Work

Tiered Memory Systems. Numerous solutions[18, 24, 35, 40] consider non-volatile memory (NVM) to be a slow memory tier. HeMem [35] introduces a flexible, per-application memory management policy at the user level. AutoTiering [24] addresses multi-tiered memory system utilization by considering access tier and locality without a predefined threshold. Diverging from NVM-focused methods, Pond [26] and TPP [29] 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. Mercury decouples memory temperature from application importance, 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 [15] maximizes hit rates and provides isolation in multi-tenant web applications with a log-structured, application-aware design. Aequitas [41] and DiffServ [12] prioritize traffic at the network edges, with Aequitas targeting data center environments to ensure latency SLOs for RPCs. TMTS [18] uses two metrics to meet SLOs in tiered memory systems, prioritizing LS applications. Mercury introduces QoS management that ensures strict priority and fairness across application types.

Interference management. Prior systems like Heracles [27] and PARTIES [14] dynamically adjust partitions to handle memory bandwidth interference, with PARTIES providing enhanced isolation for memory capacity and disk bandwidth. MCP [31] reduces inter-application interference by mapping data to separate channels, while IMPS [31] prioritizes memory non-intensive applications, which can be unfair. The FQ memory scheduler [33] prioritizes memory requests by earliest virtual finish-time. ASM [37] 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 [13, 19, 21, 39] face significantly higher latency than CXL memory [20]. 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.

8 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 and real-time adapatation 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] 2012. NUMA Balancing. https://mirrors.edge.kernel.org/pub/linux/kernel/people/andrea/autonuma/autonuma_bench-20120530.pdf. (2012).

[2] 2013. memtier_benchmark: A High-Throughput Benchmarking Tool for Redis & Memcached. https://redis.io/blog/memtier_benchmark-a-high-throughput-benchmarking-tool-for-redis-memcached/. (2013).

[3] 2015. Linux cgroups. https://docs.kernel.org/admin-guide/cgroup-v2.html. (2015).

[4] 2018. Intel Platform QoS Technologies. https://wiki.xenproject.org/wiki/Intel_Platform_QoS_Technologies. (2018).

[5] 2023. PEBS. Intel 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/articles/intel-sdm/. (2023).

[6] 2024. Compute Express Link (CXL). https://www.computeexpresslink.org/. (2024).

[7] 2024. Inference of Meta's LLaMA model (and others) in pure C/C++. https://github.com/ggerganov/llama.cpp/. (2024).

[8] 2024. memcached. https://memcached.org/. (2024).

[9] 2024. Redis. https://github.com/redis/redis/. (2024).

[10] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *ASPLOS*.

[11] AMD. 2024. μProf User Guide. (2024). https://www.amd.com/content/dam/amd/en/documents/developer/version-4-2-documents/uprof/uprof-user-guide-v4.2.pdf

[12] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, , and W. Weiss. 1998. RFC2475: An Architecture for Differentiated Service. In *IETF*.

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

[14] Shuang Chen, Christina Delimitrou, and José F. Martínez. 2019. PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services. In *ASPLOS*.

[15] Asaf Cidon, Daniel Rushton, Stephen M. Rumble, and Ryan Stutsman. 2017. Memshare: a Dynamic Multi-tenant Key-value Cache. In *ATC*.

[16] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. 2024. The Faiss library. (2024). arXiv:cs.LG/2401.08281

[17] Paul J. Drongowski. 2007. Instruction-Based Sampling: A New Performance Analysis Technique for AMD Family 10h Processors. (2007). https://www.amd.com/content/dam/amd/en/documents/archived-tech-docs/white-papers/AMD_IBS_paper_EN.pdf

[18] 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. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *ASPLOS*.

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

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

[21] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. 2017. Efficient Memory Disaggregation with Infiniswap. In *NSDI*.

[22] Olivier Chapelle Jean-Baptiste Tien, joycenv. 2014. Display Advertising Challenge. (2014). https://kaggle.com/competitions/criteo-display-ad-challenge

[23] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *ISCA*.

[24] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *ATC*.

[25] Brian Kulis, Prateek Jain, and Kristen Grauman. 2009. Fast similarity search for learned metrics. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 31, 12 (2009), 2143–2157.

[26] 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. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *ASPLOS*.

[27] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. 2015. Heracles: Improving Resource Efficiency at Scale. In *ISCA*.

[28] Yu A Malkov and Dmitry A Yashunin. 2018. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence* 42, 4 (2018), 824–836.

[29] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered Memory. In *ASPLOS*.

[30] Meta. 2024. Llama 2 70B - GGUF Model. (2024). https://huggingface.co/TheBloke/Llama-2-70B-GGUF

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

[32] 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. 2019. Deep Learning Recommendation Model for Personalization and Recommendation Systems. *CoRR* abs/1906.00091 (2019). https://arxiv.org/abs/1906.00091

[33] Kyle J. Nesbit, Nidhi Aggarwal, James Laudon, and James E. Smith. 2006. Fair Queuing Memory Systems. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 208–222. https://doi.org/10.1109/MICRO.2006.24

[34] Yuanjiang Ni, Pankaj Mehra, Ethan Miller, and Heiner Litz. 2023. TMC: Near-Optimal Resource Allocation for Tiered-Memory Systems. In *SoCC*.

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

[36] Sai Sha, Chuandong Li, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2023. vTMM: Tiered Memory Management for Virtual Machines. In *EuroSys*.

[37] Lavanya Subramanian, Vivek Seshadri, Arnab Ghosh, Samira Khan, and Onur Mutlu. 2015. The Application Slowdown Model: Quantifying and Controlling the Impact of Inter-Application Interference at Shared Caches and Main Memory. In *MICRO*.

[38] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. 2023. LLaMA: Open and Efficient Foundation Language Models. (2023). arXiv:cs.CL/2302.13971

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

[40] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2023. Nimble Page Management for Tiered Memory Systems.. In *ASPLOS*.

[41] Yiwen Zhang, Gautam Kumar, Nandita Dukkipati, Xian Wu, Priyaranjan Jha, Mosharaf Chowdhury, and Amin Vahdat. 2022. Aequitas: Admission Control for Performance-Critical RPCs in Datacenters. In *SIGCOMM*.