Leave No One Behind: Towards Fair and Efficient Tiered Memory Management for Multi-Applications

Wenda Tang
China Telecom Cloud Computing Research Institute
Beijing, China
tangwd1@chinatelecom.cn

Yiduo Wang
China Telecom Cloud Computing Research Institute
Beijing, China
wangyd22@chinatelecom.cn

Yanwen Wang
China Telecom Cloud Computing Research Institute
Beijing, China
wangyw22@chinatelecom.cn

Jie Wu
China Telecom Cloud Computing Research Institute
Beijing, China
wujie@chinatelecom.cn

Abstract

The emergence of byte-addressable memory technologies, such as CXL-attached memory, has catalyzed extensive research into tiered memory management. Existing tiering solutions optimize system-wide performance by migrating frequently accessed ("hot") data to fast-tier memory via page migration, which serves as the *de facto* mechanism in modern OS. However, these strategies often fail in the multi-tenant environment, where diverse workloads interfere with each other. For example, latency-critical workloads co-located with throughput-oriented ones may face the "cold page dilemma," where critical pages are misclassified as "cold" and migrated to slower tiers, leading to significant performance degradation. Moreover, current methods often assume negligible migration overhead, which becomes problematic in multi-core systems handling write-intensive workloads that incur substantial costs. This paper proposes VULCAN, a workload-aware tiered memory management framework that targets fair and efficient tiering in multi-tenant environments. VULCAN introduces four key innovations: (1) workload-dependent migration mechanism, which decouples page migration from the OS kernel to enhance operational flexibility for multi-workloads; (2) QoS-aware fair resource partitioning, which dynamically optimizes fast memory distribution through per-workload fast tier hit ratios and fairness-oriented allocation policies; (3) per-thread page table replication, which minimizes TLB coherence overhead during migration; and (4) biased page migration policy, which optimizes efficiency by considering both access characteristics (read-intensive vs. write-intensive) and thread-level page ownership (private vs. shared). We evaluated VULCAN using multiple representative cloud applications with realistic working sets in co-location scenarios. VULCAN improves performance by 12.4% on average and achieves a 75.3% improvement in fairness compared to existing state-of-the-art memory tiering solutions.

CCS Concepts

• **Software and its engineering → Memory management**; • **Hardware** → *Emerging architectures.*

Keywords

Memory Tiering, Memory Management, Isolation, Fairness

ACM Reference Format:
Wenda Tang, Yiduo Wang, Yanwen Wang, and Jie Wu. 2025. Leave No One Behind: Towards Fair and Efficient Tiered Memory Management for Multi-Applications. In *54th International Conference on Parallel Processing (ICPP '25), September 08–11, 2025, San Diego, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3754598.3754626

1 Introduction

The rise of big data applications has resulted in a significant increase in main memory requirements [3, 18, 24]. For example, machine learning (ML) models are expected to grow by 50× in the next few years [25]. As data-intensive applications continue to scale, the existing memory hierarchy struggles to keep pace with the growing demand. Fortunately, emerging memory technologies, such as Compute eXpress Link (CXL) [23, 44] and Non-Volatile Memory (NVM) (e.g., Intel Optane DC PMem) [19] present new opportunities to address this challenge. Using a tiered memory architecture [20, 23, 44] that integrates various types of memory with varying capacity, latency, and cost characteristics, systems can achieve a balance between scalability and cost-effectiveness.

However, such architectures pose significant challenges for data management due to the highly diverse memory access patterns of workloads. These patterns differ significantly depending on the workload type: latency-critical (LC) workloads, such as online services, require low-latency responses, while best-effort (BE) workloads, like batch processing, prioritize throughput. For example, large-scale graph processing involves intensive irregular random access [9], while in-memory databases combine sequential and random access [17]. This diversity significantly complicates maintaining consistent performance, particularly when LC and BE workloads are co-located to maximize resource utilization.

Researchers have proposed various dynamic data management techniques that aim to optimize data placement and movement within tiered memory architectures [8, 18, 23, 27, 29, 31, 39–41]. These approaches commonly utilize a placement algorithm that, at runtime, moves hot data to the fast-tier memory (promotion) and cold data to the slow-tier memory (demotion) [18, 31, 37, 40]. These methods vary in migration granularity, such as cache lines [44], pages [31, 39–41], or objects [5, 32], and often incorporate strategies based on data access frequency [29], recency [41], or a combination of both [12]. Among these, page-based migration is popular for balancing control and overhead. Compared to cache-coherent-based migration (via cache-line granularity), it reduces metadata overhead and migration frequency, avoids inefficiencies such as cache thrashing, and does not require specialized hardware support. It also offers better adaptability than object-based migration, which relies on application-level code modifications to manage data placement. Furthermore, page-based migration integrates seamlessly with existing memory management mechanisms, such as paging-based virtual memory and Translation Lookaside Buffers (TLBs), making it both practical and efficient.

Limitations of Prior Successes. Although existing page-based memory tiering solutions are effective in optimizing system-wide performance, they have two fundamental limitations. *First,* they often rely on raw page access statistics without normalizing across different workload characteristics, which poses a fundamental challenge for providing performance isolation in the multi-tenant environment. For example, when co-located with BE workloads, which generate sustained and frequent memory accesses, LC workloads are forced into the "cold page dilemma" because BE workloads tend to monopolize fast memory by making their working sets appear persistently "hot". This imbalance in data placement causes essential hot pages of LC workloads to be classified as cold and migrated to slower memory tiers, resulting in performance degradation to **0.8**× of the standalone baseline. This phenomenon highlights the need to reevaluate memory tiering solutions in the multi-tenant environment to address the "dilemma" and improve fairness. We provide a detailed analysis of the dilemma in § 2.2.

Second, existing memory tiering solutions often assume that the overhead of page migration between memory tiers is negligible. However, as we demonstrate in § 2.2, this assumption does not hold in practice, particularly in multi-core systems. We found that page migration introduces two main sources of overhead: *migration preparation* and *page remapping*. During the preparation phase, operations such as draining per-CPU LRU caches and acquiring migration locks incur substantial cross-core synchronization costs, with preparation time increasing by up to **30**× when scaling from 2 to 32 cores. Furthermore, during the remapping phase, updating page table entries triggers TLB coherence operations across all cores [4], consuming up to **65%** of total page migration cycles. These overheads not only degrade system performance but also limit the scalability of tiered memory systems.

Insights. To address the "cold page dilemma" and mitigate the high overhead associated with page migration, we present Vulcan[1], a novel workload-aware tiered memory management framework that targets fair and efficient memory tiering for multi-tenant environments, striving to ensure that *no one is left behind.* The design of Vulcan is driven by four key insights: ❶ workload-dependent tiering could prevent the "cold page dilemma" by dynamically allocating fast-tier memory based on workload-specific requirements, ensuring performance isolation and reducing unnecessary global synchronization. ❷ Fairness and resource efficiency are crucial in the multi-workload environment to prevent resource starvation, ensure equitable memory distribution, and enhance performance for tiering-sensitive workloads, thus improving overall system efficiency. ❸ Precise knowledge of page-sharing threads can be leveraged to minimize the scope of TLB shootdowns, as coherence actions are only necessary for threads actually sharing the same page-table entries (PTEs), enabling targeted updates instead of system-wide operations. ❹ The diverse performance impact of page migration calls for a more fine-grained policy, as migration overhead varies significantly based on access frequency (read-intensive vs. write-intensive) and thread-level ownership (private vs. shared).

Sketch of Vulcan Design. Based on these insights, Vulcan employs workload-dependent tiering mechanisms to achieve performance isolation, effectively preventing the "cold page dilemma" by dynamically adjusting fast-tier memory allocation according to workload-specific requirements. This approach ensures that critical workloads maintain high performance while minimizing interference between workloads. Additionally, such tiering reduces system-wide CPU synchronization overhead by avoiding unnecessary global synchronization, further improving migration efficiency. Beyond performance isolation, Vulcan ensures fairness and resource efficiency by dynamically balancing resource allocation based on the effectiveness of tiering. This adaptive mechanism allows workloads with higher sensitivity to tiering to receive prioritized resources, maximizing overall system efficiency while maintaining fairness across workloads. To further enhance migration efficiency, Vulcan leverages precise knowledge of page-sharing threads to minimize the scope of TLB shootdowns during page migration. By maintaining per-thread page tables, Vulcan limits TLB coherence operations to only the necessary cores, avoiding unnecessary system-wide coherence. Finally, Vulcan prioritizes asynchronous copying for read-intensive and private pages, while deferring write-intensive and shared pages to synchronous copying, thereby avoiding execution stalls during page migration.

To summarize, the contribution of this paper mainly includes:

- **Analyses.** We thoroughly analyze the behavior of existing tiered memory systems on multi-core architectures under multi-workload co-location scenarios and present two key findings: 1) we identify the "cold page dilemma", which arises from conventional hotness-based allocation that disproportionately favors high-intensity workloads, causing LC applications to lose access to essential fast-tier memory; 2) we reveal the hidden costs of page migration, often ignored in existing systems, which incur substantial overhead in multi-threaded applications due to TLB shootdowns, memory copy costs, and cross-core synchronization, often offsetting the intended performance benefits of memory tiering.
- **Vulcan Design.** We propose Vulcan, the first tiered memory page management framework that addresses the above problems with four key innovations: 1) workload-dependent migration mechanism to achieve greater flexibility and performance isolation in the multi-tenant environment (**§ 3.2**); 2) online profiling-guided memory allocation to ensure fair and efficient fast memory distribution across workloads (**§ 3.3**);

[1]Vulcan, the Roman god known for crafting diverse weapons, each tailored for unique purposes, mirrors our system's ability to optimize memory for diverse applications.

3) per-thread page table replication to reduce TLB coherence overhead and improve multi-threaded memory access efficiency (**§ 3.4**), and 4) biased page migration policy to prioritize selecting pages with lower migration overhead by analyzing their access patterns and page ownerships (**§ 3.5**).

- **Evaluation.** We conduct a comprehensive evaluation of VULCAN using various microbenchmarks and representative memory-intensive applications, comparing its performance against several leading memory tiering solutions in co-location scenarios. Our results show that VULCAN improves performance by an average of approximately 12.4% across different workloads and improves the fairness of memory allocation by an average of 75.3%.

2 Background and Motivation

Tiered memory systems have emerged as a critical solution to address the increasing memory demands of modern workloads by combining fast, expensive memory (e.g., DRAM or High Bandwidth Memory (HBM)) with slower, larger and cheaper memory (e.g., NVM). These systems rely on effective page management to dynamically allocate memory across tiers and maximize performance. However, managing tiered memory is non-trivial due to the competing demands of diverse workloads, the overhead of page migration, and the need to ensure fairness among co-located applications.

To better understand the challenges and limitations of existing tiered memory systems, we first discuss their core components, including profiling mechanisms, migration mechanisms, and migration policies. We then analysis the key challenges faced by current tiered memory systems, particularly under co-located workloads, and identify the hidden costs of page migration. This analysis highlights the inefficiencies in current approaches and motivates the need for novel solutions.

2.1 Analysis of Tiered Memory Management

Most of the multi-tier memory page management systems consist of three components: a profiling mechanism, a migration mechanism, and a migration policy [31, 40].

(1) Profiling mechanism plays a crucial role in understanding page access patterns and making effective migration decisions in tiered memory systems. Three primary profiling methods are commonly used: *NUMA (Non-uniform memory access) hinting faults*, *page table scanning/profiling* and *CPU performance counters. NUMA hinting faults* utilize a fake page fault by poisoning through reserved PTE bits to estimate access frequency, as used in AutoTiering [16], FlexMem [40] and TPP [23], but incur extra latency from fault handling. A variant of NUMA hinting faults is the timer-based hotness measurement method, as recently implemented in Chrono [27], which improves the estimation of access frequency by recording idle time. *Page table scanning* periodically checks PTE reference bits to determine access frequency, adopted by systems such as Nimble [41] and MULTI-CLOCK [22], but faces scalability challenges with per-page scanning. Telescope [24] introduces an advanced variant, *page table profiling*, for efficient terabyte-scale tiered memory systems. The last method uses Processor Event-Based Sampling (PEBS) to monitor hardware events such as LLC (Last-Level Cache) misses for per-page access statistics [8, 18, 20, 29, 40], though it suffers from high false negatives at the terabyte scale due to sampling limitations [24]. Unfortunately, none provide a universal solution due to inherent compromises, indicating that applications should independently select the profiling approach that best fits their needs.

(2) Migration mechanism describes how pages are transferred between memory tiers through five key steps: ❶ kernel trapping, ❷ PTE locking and unmapping, ❸ TLB shootdown through IPIs (Inter-Processor Interrupts), ❹ content copying between tiers, and ❺ PTE remapping [39]. Pages can be migrated either synchronously, blocking program execution for immediate migration (as in TPP's page promotion [23]), or asynchronously [18, 31, 39] through dedicated threads like `kswapd` off the critical path. Recent research has explored various optimization techniques to reduce the migration overhead associated with the migration mechanism. For example, Nimble [41] introduces optimized mechanisms like transparent huge page and concurrent multi-page migration for better scalability. HeMem [29] employs DMA engines to accelerate page copying between tiers. MEMTIS [18] and TMTS [8] move migration off the critical path. NOMAD [39] further removes page migration from the critical path of program execution by employing *transactional migration*, making migration completely asynchronous. However, existing methods inevitably incur significant TLB coherence overhead during migration, which adversely affects system performance and scalability.

(3) Migration policy determines when and which pages to migrate between memory tiers for better performance. Modern systems like Linux, TPP [23], and AutoTiering [16] use proactive page reclamation when fast-tier memory falls below a `low_watermark` threshold, while Colloid [37] adaptively balances hot pages across tiers based on tier-specific access latencies. However, most existing policies overlook migration overhead. To the best of our knowledge, NOMAD [39] is one of the pioneering works that addresses this issue by introducing *page shadowing* to mitigate performance impact. Despite this, it fails to adapt policies based on page access characteristics, rendering it suboptimal. Although another approach, MTM [31], utilizes both asynchronous and synchronous page copying based on write intensity, i.e., synchronous page copying for write-intensive pages and asynchronous page copying for read-intensive ones, it still lacks a fine-grained consideration of the migration costs inherent in multi-CPU core scenarios, which can lead to suboptimal performance during concurrent migrations.

2.2 Motivation

In this section, we identify and summarize key observations on two fundamental issues that current tiered memory systems fail to address in real-world multi-tenant scenarios on multi-core platforms.

Cold Page Dilemma. Existing memory tiering solutions struggle to meet *Quality of Service (QoS)* demands under the co-location of LC and BE workloads. The presence of BE workloads leads to significant performance degradation for LC applications, particularly when fast memory resources are constrained. To illustrate this issue, we use MEMTIS [18], a state-of-the-art capacity-based tiering system that represents modern tiering solutions to classify hot and cold pages and evaluate the performance of co-located

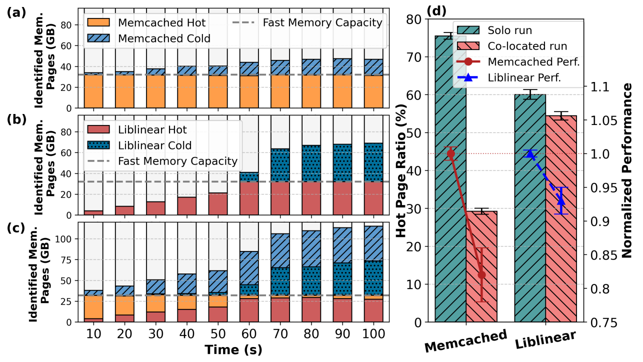

Figure 1: Hot and cold pages identified over time in MEMTIS [18] for Memcached (LC) in (a) and Liblinear (BE) in (b), under solo scenarios, and in the co-located scenario (c). The impact of co-location on performance and hot page ratio is shown in (d).

Memcached [10] and *Liblinear* [35] workloads. Our analysis of the workload behavior reveals the following key observation.

Observation #1: *Current memory tiering is agnostic to workload memory sensitivity, forcing LC workloads into the "cold page dilemma," where hot pages are downgraded to cold as BE workloads monopolize fast memory, causing higher latency and performance degradation.*

Figure 1 illustrates the distribution of hot and cold pages for Memcached and Liblinear across three scenarios under the same hardware settings (as described in § 5): (a) Memcached running in isolation, (b) Liblinear running in isolation, and (c) Memcached and Liblinear running in a co-located environment. We observe that when Memcached and Liblinear are co-located, the number of identified cold pages for Memcached significantly increases compared to its solo execution. This is because MEMTIS ranks memory pages based on their absolute access frequency and promotes them to fast memory in descending order of heat until the fast memory capacity is fully utilized. Due to Liblinear's higher memory access intensity, it dominates the fast memory capacity under co-location, leaving Memcached's hot pages with limited opportunities for promotion. This behavior is particularly evident in Figure 1 (c), where Liblinear's memory heavily occupies the fast tier.

Figure 1 (d) quantifies the impact of co-location on performance. For Memcached, the average hot page ratio drops drastically from approximately 75% in solo execution to less than 28% under co-location, leading to a noticeable performance degradation (normalized performance drops to 0.8). In contrast, Liblinear experiences a relatively lower performance impact due to its BE workload characteristics and its ability to tolerate slower memory. The results suggest a need for more effective memory management strategies to ensure QoS guarantee in a co-located workload environment.

Hidden Costs of Page Migration. To better understand the overheads of page migration, which are essential for designing efficient memory tiering mechanisms, we perform three sets of experiments to investigate the costs associated with page migration: 1) breaking down single base-page migration overhead with varying CPU counts (ranging from 2 to 32), 2) analyzing migration overhead under different numbers of pages and threads with fixed 32 CPUs, and 3) comparing the performance of synchronous and

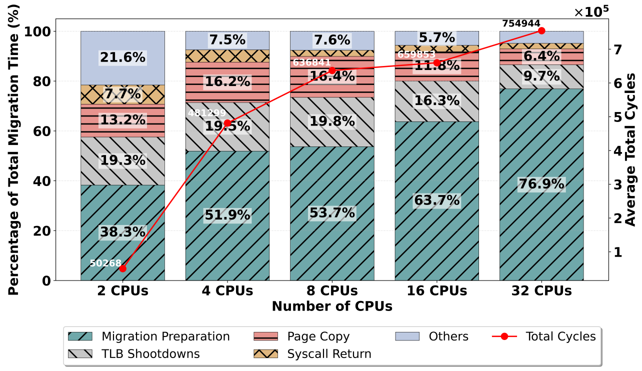

Figure 2: Breakdown of migration costs for single base-page (4KB page size) across varying numbers of CPUs.

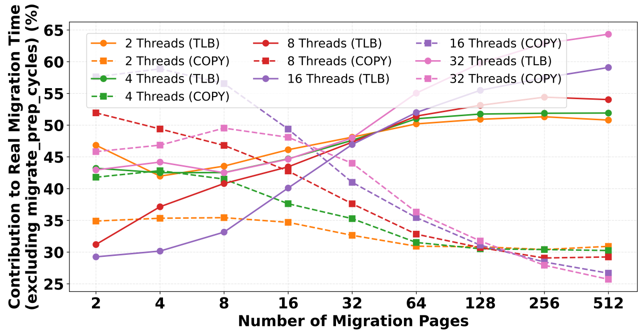

Figure 3: Contribution of TLB operations and page copy operations to real migration time across varying numbers of migration pages and threads.

asynchronous page copying for hot page migration under different memory access patterns. These experiments reveal three additional key observations about page migration.

Observation #2: *Page migration suffers from poor scalability due to system-wide synchronization overhead in migration preparation, which dominates total migration time as CPU count increases.*

As shown in Figure 2, when scaling from 2 to 32 CPUs, the migration preparation cost dominates and increases dramatically from 38.3% to 76.9% of the total migration time, while other phases, such as page unmapping and copying, decrease proportionally. More critically, the total migration latency for this single page scales poorly with CPU count, rising from 50K to 750K cycles, highlighting the severe impact of multi-CPU synchronization overhead. This poor scalability stems from the migration preparation in the Linux kernel, driven by the LRU cache flushing mechanism (`lru_add_drain_all()`), which uses `on_each_cpu_mask()` for CPU synchronization. The synchronization process introduces various overheads, including lock contention, cache line invalidation, and potential scheduling delays, all of which escalate significantly as the CPU count increases.

Observation #3: *TLB coherence management presents a scalability challenge in page migration, particularly in migrations requiring multi-core TLB synchronization.*

Even with migration preparation overhead eliminated through some optimizations, fast page migration remains challenging due to the cross-CPU TLB coherence requirements during page remapping.

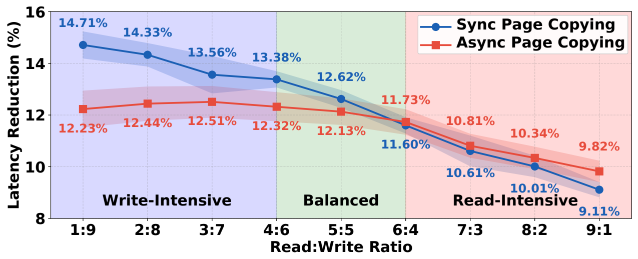

Figure 4: Performance comparison of synchronous and asynchronous page copying for hot page migration across different read-write ratios (higher is better).

As shown in Figure 3, our experiments on a 32-core system reveal an interesting trade-off between TLB operations and page copying costs. When migrating few pages, page copying dominates the total migration time as TLB synchronization overhead is minimal. However, as the number of pages increases, the TLB coherence overhead grows significantly due to increased cross-CPU synchronization, while the page copying overhead grows relatively slowly. This leads to TLB operations that consume up to 65% of the total migration time with 32 threads and 512 pages, becoming a major bottleneck in many core systems.

Observation #4: *To sync or to async? That is the question in migrating hot pages across varied memory access patterns.*

We perform a microbenchmark workload that promotes a single base-page from a slow tier to a fast tier while simultaneously accessing the page with varying read/write ratios. This setup allows us to compare the performance of synchronous and asynchronous page copying mechanisms under different access patterns, highlighting their respective trade-offs in handling promotion of hot pages. As shown in Figure 4, asynchronous copying, which performs background page copying and remapping at appropriate times, excels in read-intensive workloads by reducing immediate page copying stalls. However, it may struggle in write-intensive workloads due to high dirty page rates, leading to repeated copying or migration failures. In contrast, synchronous copying performs better for write-intensive workloads as it handles high dirty page rates more effectively, avoiding migration inefficiencies.

Implications. The above observations motivate us to design a workload-aware memory tiering framework to improve performance and scalability in the multi-tenant environment. The framework should dynamically adapt memory allocation policies to avoid the "cold page dilemma" and ensure low-latency workloads are not penalized by fast memory monopolization (***Observation #1***). When handling page migrations, the framework should optimize system-wide synchronization processes and manage TLB coherence efficiently to overcome scalability bottlenecks in multicore systems (***Observation #2*** and ***Observation #3***). Finally, the migration strategy should be adaptive, leveraging synchronous or asynchronous approaches based on workload characteristics and memory access patterns, thus improving flexibility and reducing migration overhead (***Observation #4***).

3 Design of Vulcan

In this section, we elaborate on the design details of Vulcan starting with an overview, followed by detailed discussions of its main parts.

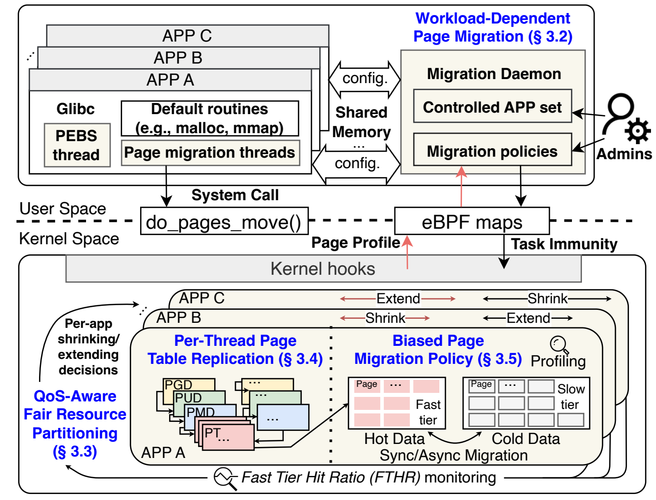

Figure 5: The overall architecture of Vulcan.

3.1 Overview

Design Goals. We derive the following three first-order design goals for Vulcan: (**G#1**) The system should enable workload-specific migration strategies, providing the flexibility needed to adapt to diverse and dynamic workload demands. (**G#2**) The system should ensure fair and balanced allocation of fast memory resources among co-located workloads, guided by workload demands and QoS requirements. (**G#3**) The system should optimize page migration efficiency to minimize performance impact by improving both migration strategies and mechanisms.

Architecture. To achieve the above goals, Vulcan's design consists of four components (see Figure 5 for details, introduced later).

(1) Workload-Dependent Page Migration. Vulcan integrates a lightweight user-space manager with applications, enabling fine-grained and transparent migration that is precisely tailored to the specific needs of each workload.

(2) QoS-Aware Fair Resource Partitioning. As discussed in § 2.2, applications exhibit diverse memory behaviors. Vulcan ensures long-term fairness by dynamically partitioning fast memory tiers across multiple applications.

(3) Per-Thread Page Table Replication. To mitigate TLB shootdowns caused by page migration in multithreaded applications, Vulcan maintains per-thread page table replicas, reducing TLB coherence overhead.

(4) Biased Page Migration Policy. Through access pattern analysis and thread-level page table replication, Vulcan prioritizes read-intensive and private pages for biased migration while deferring write-intensive and shared pages to minimize overhead.

3.2 Workload-Dependent Page Migration

Vulcan adopts a decentralized approach, allowing applications to manage memory page migration independently. This is implemented through a dynamically linked library (via `LD_PRELOAD`) and dedicated migration threads created for each application. This approach satisfies the first goal **G#1** presented in § 3.1.

As shown in Figure 5, the migration daemon operates exclusively on a controlled set of whitelisted applications managed by the system administrator. This access control mechanism ensures operational security while maintaining system stability. The daemon leverages eBPF (extended Berkeley Packet Filter) [2] to support flexible profiling choices through multiple mechanisms (e.g., PEBS sampling, page table scanning), enabling decoupled page profiling selection. For example, the daemon could utilize Linux's `perf` interface (i.e., `perf_event_open()`) to collect memory access patterns for hot page identification, while an eBPF program hooked to `do_pages_move()` provides targeted monitoring exclusively for managed processes. Inspired by FlexMem [40], Vulcan by default adopts a hybrid profiling approach that integrates performance counter-based profiling and page hinting fault-based profiling to overcome the limitations of sampling-based memory tracking. To optimize memory migration, migration decisions, made by the access pattern-aware migration policy (to be discussed in § 3.5), are triggered periodically by the daemon and executed in parallel by each application's dedicated migration threads through shared memory interfaces, eliminating system-wide synchronization cost.

3.3 QoS-Aware Fair Resource Partitioning

Intuitively, allocating enough fast memory to each application helps to preserve performance, as frequent access to slower memory tiers can significantly degrade it [8]. A straw-man solution is uniform allocation, which evenly distributes fast memory across co-located workloads in tiered memory systems. Although simple and initially fair, this approach often results in inefficiencies due to its inability to adapt to the dynamic and variable memory demands of workloads. High-demand workloads may experience resource shortages, while others receive excessive resources, leading to suboptimal efficiency. To address this issue, we propose a dynamic partitioning strategy that simultaneously improves resource efficiency while guaranteeing fair allocation of fast memory among co-located workloads in tiered memory systems.

Tiered Memory QoS Policy. To quantify memory tiering performance, we define a workload-specific guaranteed performance target $GPT_i = \frac{\text{GFMC}}{\text{RSS}_i}, i \in \{1, 2, ..., n\}$, where n denotes the number of co-located workloads. The GPT_i, akin to a QoS baseline, quantifies the relationship of workload i's fast memory allocation to the system's overall fast memory capacity. Here, GFMC (Guaranteed Fast Memory Capacity) refers to the fast memory equally allocated among all co-located workloads, dynamically adjusting based on n. RSS_i (Resident Set Size) denotes the memory actively used by workload i. When $\text{GFMC} \geq \text{RSS}_i$, we keep $GPT_i = 1$, indicating that fast memory fully covers the active memory of the workload. Otherwise, when $\text{GFMC} < \text{RSS}_i$, this indicates that only part of the active memory resides in fast memory, with the rest in slower tiers.

Although GPT_i provides a workload-specific QoS guarantee to evaluate fast memory allocation, it does not capture the dynamic behavior of fast memory utilization over time. To this end, we define the *Fast-Tier Hit Ratio (FTHR)* as a dynamic metric that reflects the effectiveness of fast memory usage over time. For workload i, the FTHR_i is updated iteratively based on both recent and historical sampling data. At time t, N samples are collected to compute the

Algorithm 1: Credit-Based Fair Resource Partitioning

```
Input  : Latency-critical & best-effort workloads: LC, BE
         Credits of all workloads: $C$
         Resource demand of workloads: $\vec{D} = \{demand_i\}$
Output : Updated fast tier memory allocation: $\vec{A} = \{alloc_i\}$
         Updated credits: $C'$
1  for $i \in \{1, 2, ..., n\}$ do
2      $alloc_i \leftarrow min\{demand_i, \text{GFMC}\}$ ;
3  LC_borrowers ← $\{i \in \text{LC} \mid alloc_i < demand_i\}$;
4  BE_borrowers ← $\{i \in \text{BE} \mid alloc_i < demand_i\}$;
5  donors ← $\{i \mid alloc_i > demand_i\}$;
6  while LC_borrowers $\neq \emptyset$ or BE_borrowers $\neq \emptyset$ do
7      $b^\star \leftarrow$ Select borrower from LC_borrowers if
         non-empty, else BE_borrowers;
8      if donors $\neq \emptyset$ then
9          $d^\star \leftarrow$ Select donor with minimum credits;
10         Transfer 1 unit from $d^\star$ to $b^\star$ and update credits $C$;
11     else if $b^\star \in LC$ and $BE \neq \emptyset$ then
12         $d^\star \leftarrow$ Randomly select BE task with
             $alloc_{d^\star} > \text{GFMC}$;
13         Transfer 1 unit from $d^\star$ to $b^\star$ and update credits $C$;
14     else
15         return;
16     if $alloc_{b^\star} == demand_{b^\star}$ then
17         Remove $b^\star$ from LC_borrowers if $b^\star \in$ LC, else
             from BE_borrowers;
```

average hit ratio, denoted as:

$$\overline{H_{i,\text{t}}} = \frac{\sum_{k=1}^{N} a_{\text{fast},i,k}}{\sum_{k=1}^{N} (a_{\text{fast},i,k} + a_{\text{slow},i,k})}, \quad (1)$$

where $a_{\text{fast},i,k}$ and $a_{\text{slow},i,k}$ represent the number of memory accesses to fast and slow memory, respectively, during the k-th sample. The value of FTHR_i is then updated using the exponential moving average (EMA), which considers historical data while prioritizing recent observations, balancing responsiveness with stability:

$$FTHR_i = \alpha \cdot \overline{H_{i,t}} + (1 - \alpha) \cdot \overline{H_{i,t-1}}, \quad (2)$$

where $\alpha \in [0, 1]$ is a weighting factor that determines the relative importance of recent sampling data compared to historical data. We empirically set $\alpha = 0.8$. If $FTHR_i < GPT_i$, the workload is considered under-allocated in fast memory. Conversely, if $FTHR_i \geq GPT_i$, the current allocation is deemed sufficient. Then, the fast memory demand is updated based on the following formula:

$$demand_i = alloc_i + (GPT_i - FTHR_i) \cdot RSS_i \cdot \log^2(RSS_i), \quad (3)$$

where $alloc_i$ denotes the current fast memory allocation. The logarithmic scaling factor ensures that the adjustment is proportional to the workload's memory footprint, thereby providing a scalable and workload-sensitive mechanism for dynamic memory tiering.

Credit-Based Fair Resource Partitioning (CBFRP). To allocate fast memory resources based on calculated demands (i.e., demand_i),

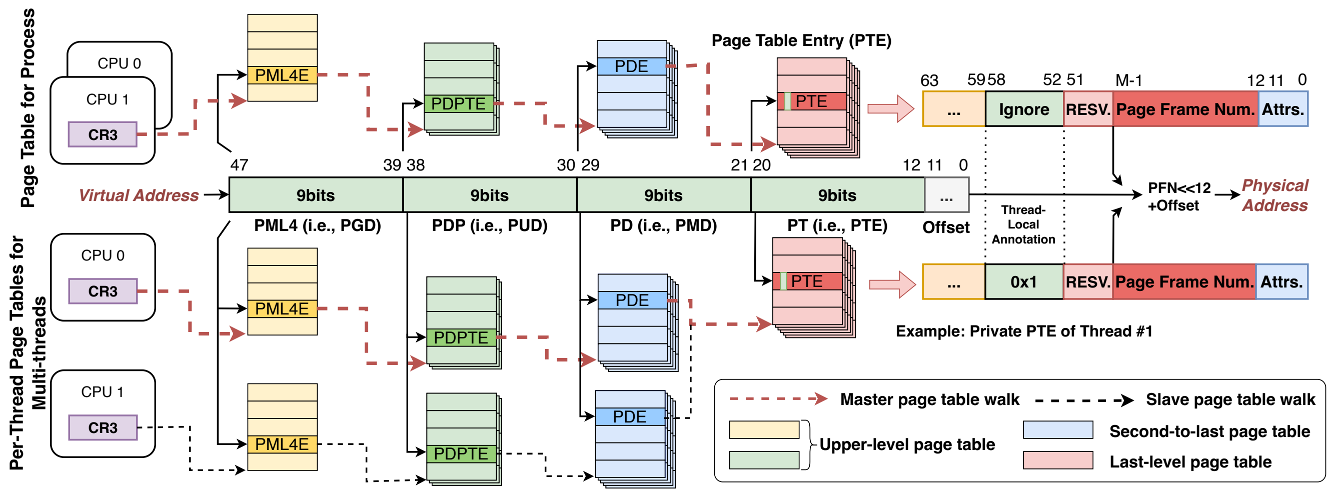

Figure 6: Comparison of vanilla process-wide and VULCAN's pre-thread page table management. Unlike process-wide sharing, VULCAN maintains pre-thread upper-level page tables while sharing the last-level page tables across threads, which constitute the majority of the page table structure.

we propose a CBFRP strategy. Inspired by *Karma* [38], which incorporates historical allocations into max-min fairness through "credit" tracking, CBFRP borrows this concept to address memory tiering challenges. VULCAN dynamically manages fast memory resources and credit balances to adapt to diverse workload characteristics, achieving long-term fairness while optimizing resource utilization. Initially, fast memory is evenly distributed among all controlled workloads as a baseline allocation, with each workload assigned initial credits to facilitate fair memory sharing. We then classify black-box workloads as either LC or BE based on resource utilization patterns [34] to ensure differentiated QoS guarantees. Based on calculated demands, workloads are further categorized as *borrowers* ($demand_i > alloc_i$), which require additional fast memory, or *donors* ($demand_i < alloc_i$), which have surplus fast memory available for redistribution.

As outlined in Algorithm 1, the reallocation process prioritizes LC workloads to meet their strict QoS requirements. Fast memory is dynamically reassigned from donors to borrowers until either all demands are met or no surplus memory remains. This iterative approach ensures efficient and fair allocation of fast memory while adhering to the differentiated QoS requirements of LC and BE workloads, thereby fulfilling design goal **G#2** in § 3.1.

3.4 Per-Thread Page Table Replication

In conventional systems, page migration requires TLB shootdowns through IPIs across all CPUs running the process threads, due to the process-wide shared page table structure. Our key insight is that this global synchronization is unnecessarily conservative for private memory pages, as TLB coherence only needs to be maintained among CPUs actually sharing the migrating page. Based on this understanding, VULCAN introduces a novel per-thread page table replication mechanism that efficiently identifies and manages thread-private memory pages. By maintaining per-thread page table entries, the system minimizes TLB coherence overhead through selective synchronization during migration operations. Given that

Table 1: Page promotion priority and strategy.

Page Type	Read/Write Pattern	Priority	Strategy
Shared	Read-intensive	★★★	Async copy
Shared	Write-intensive	★	Sync copy
Private	Read-intensive	★★★★	Async copy
Private	Write-intensive	★★	Sync copy

last-level page tables constitute the majority of page table memory [43], VULCAN achieves high memory efficiency by sharing them across threads while repurposing unused PTE bits (52-58) [13] of last-level page tables for thread ownership tracking. This design requires only per-thread upper-level tables, which are relatively small in size. We currently focus on base-page (4KB page size) TLB shootdowns because, inspired by MEMTIS [18], VULCAN manages huge-page (2MB page size) promotions by splitting them into base pages to prevent memory wastage, thus allowing more efficient memory resource management. Figure 6 presents a comparison of vanilla process-wide and VULCAN's per-thread page table management while providing a conceptual overview of this design.

3.5 Biased Page Migration Policy

Memory copy operations, being resource-intensive, could significantly degrade system performance by introducing application stalls during synchronous migration. Inspired by MTM [31], which uses asynchronous and synchronous page copying based on write intensity, we augment this approach by adding thread-level page table replication to further address migration inefficiencies. Building on the insight that shared pages require more coherence effort than private pages, we categorize hot pages into four types with distinct migration priorities and strategies, as shown in Table 1.

Based on categorization, VULCAN introduces four priority queues for page promotion, where memory pages are queued according to their types. Within each queue, pages are processed based on their heat levels using queue-specific migration strategies. Read-intensive and private pages are placed in the highest priority queue

(★★★★) due to the minimal TLB shootdown overhead and optimal asynchronous migration benefits. Read-intensive and shared pages are prioritized higher than write-intensive and private pages, as the overhead of page copying is lower than that of TLB shootdown operations. Write-intensive and shared pages are assigned to the lowest priority queue (★) due to the high overhead in both page copying and TLB shootdown operations. Our design also incorporates a Multi-Level Feedback Queue (MLFQ) [7] mechanism that allows pages to promote to higher-priority queues as their heat levels increase, preventing hot pages from stagnating in lower-priority queues. In addition, VULCAN enables transparent huge pages (THPs) to maximize TLB coverage by default, despite proactively splitting them into base pages during promotion.

As the fast memory capacity for each workload is dynamically adjusted according to the CBFRP strategy (see § 3.3), page demotion frequently occurs when the capacity is insufficient for promotion or when memory capacity shrinks. This process, similar to promotion, prioritizes the migration of colder pages. To mitigate migration thrashing, we borrow ideas from NOMAD's *page shadowing* technique, allowing recently promoted pages to retain shadow copies in slower tiers. This approach reduces demotion costs by remapping non-dirty pages, which are often the read-intensive and private/shared pages we previously prioritized for promotion.

Overall, the improvements in the underlying mechanisms of thread-level page table replication (discussed in § 3.4) and the biased page migration policy both directly contribute to achieving design goal **G#3** in § 3.1.

3.6 Limitations and Discussion

Although VULCAN offers several advantages, it also has inherent trade-offs. Workload-dependent page migration may face challenges when multiple workloads compete for limited system resources (e.g., memory bandwidth). Per-thread page table replication introduces memory and manipulation overhead, which can be problematic for some workloads, such as *function-as-a-service (FaaS)* [36]. Despite these limitations, the system offers noteworthy benefits, such as significantly enhanced flexibility, fairness, and efficiency in memory tiering. Many of these limitations can be mitigated through future optimizations, such as automatically enabling/disabling the thread-level page table replication mechanism based on performance trade-offs. Further, integrating with Colloid [37] could enable VULCAN to suspend the migration process of co-located workloads when the fast tier's access latency no longer offers significant advantages over alternate tiers due to memory bandwidth contention.

4 Implementation

We implemented the prototype of VULCAN on x86_64 architecture, which involves modifications to both the Linux kernel and Glibc-2.23, leveraging PTEditor [33] and the system call interception library [26]. We extended the v5.15 kernel data structures by adding the following fields: ❶ The existing PGD pointer in `struct mm_struct` is repurposed to point to the thread-private page table, while a new `process_pgd` is introduced for the process-wide page table. ❷ A thread_id field (7 bits) in PTEs, using previously ignored bits to encode either thread ownership (via thread ID) or shared status (all-ones pattern).

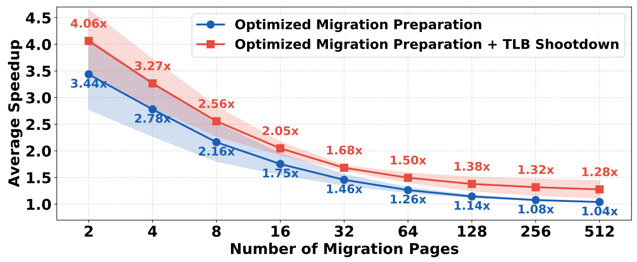

Figure 7: Speedup analysis of memory migration optimizations in VULCAN (higher is better).

We modify `switch_mm_irqs_off()` to load per-thread page tables into `CR3` during context switches, replacing traditional process-wide page table loading. We also augmented page fault handler to supports both per-thread and process-wide page tables management: it creates new mappings with thread ID for unmapped pages, while for existing mappings, it either shares the last-level page table (for shared pages with thread ID 0x7F) or establishes new shared mappings (for private pages with specific thread ID). Moreover, the migration threads in modified Glibc maintain four migration queues and classifies sampled hot pages based on their PTE flags and access patterns, dispatching them to corresponding queues for page migration.

5 Evaluation

We first analyze the performance of VULCAN's migration policies and mechanisms with a number of microbenchmarks. We then evaluate a number of real applications in a variety of different co-location configurations.

Our evaluation seeks to answer the following questions:

- What are the benefits of VULCAN's migration mechanisms and policy optimizations (§ 5.2)?
- How effectively does VULCAN maintain performance and fairness under dynamic co-location scenarios (§ 5.3)?

5.1 Experimental Setup

We use a dual-socket server with Intel Xeon Platinum 8378A CPUs (32 cores, 48MB LLC, 8×3200MHz DDR4 channels per socket) offering 205GB/s memory bandwidth and 25GB/s UPI bandwidth per direction. We use processors on a single socket, with locally-attached fast memory (32GB capacity, 70ns unloaded latency) and emulate slow memory (256GB capacity, 162ns unloaded latency) to mimic upcoming CXL memory. The emulation is achieved using a remote NUMA node[2], with the cross-NUMA interconnect frequency adjusted via BIOS settings to match CXL latency characteristics. This setup is informed by prior research, which indicates that CXL memory introduces an additional 70–90ns latency compared to local memory [20]. We compared VULCAN with three state-of-the-art tiered memory systems: TPP [23], MEMTIS [18], and NOMAD [39].

[2]Commercial ASIC CXL hardware is currently scarce and Intel has discontinued its PMem; thus, we use emulation without compromising our experimental validity.

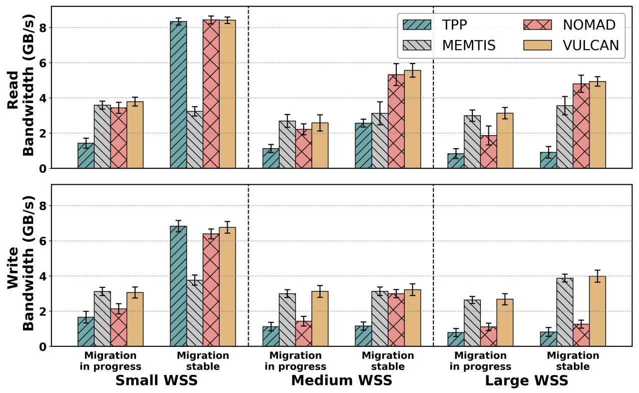

Figure 8: Migration performance comparison between TPP, Memtis, Nomad, and Vulcan (higher is better).

5.2 Microbenchmarks

Migration Mechanism. Following the methodology in § 2.2, we varied the number of pages per migration from 2 to 512 using synchronous page migrations to measure performance. By comparing the baseline implementation with our optimized migration preparation and TLB shootdown optimization, we observed significant speedups, particularly for small-scale page migrations. As shown in Figure 7, our optimizations can achieve up to 3.44× speedup with optimized migration preparation alone, and up to 4.06× speedup when combined with TLB shootdown optimization for 2-page migrations. While the benefits decrease for larger migrations due to the increasing page copying overhead, these findings suggest that our optimization strategies effectively reduce the migration preparation and TLB coherence overheads.

Migration Policy. We then borrow the microbenchmarks used to evaluate Nomad. These microbenchmarks involve 1) allocating data to specific segments of the tiered memory; 2) running tests with various working set size (WSS) and RSS values; and 3) generating memory accesses to the WSS data that mimic real-world memory access patterns with a Zipfian distribution. We created three scenarios representing small, medium, and large WSS values, to thoroughly evaluate tied memory management behavior under different realistic memory pressures. We perform over 10 trials, plotting the mean with shaded regions and error bars for 95% confidence intervals.

Figure 8 illustrates the comparison of migration performance between TPP, Memtis, Nomad and Vulcan in different sizes of working sets (small, medium, large) and migration states (migration in progress vs. migration stable). Vulcan consistently demonstrates superior read and write bandwidth, particularly in the migration stable phase, where it significantly outperforms other systems. This highlights Vulcan's ability to minimize migration overhead and maintain high performance across varying workloads, showcasing its scalability and efficiency.

5.3 Real-World Applications Study

To measure how well Vulcan can react to dynamically changing workloads, Vulcan was extended by testing three real-world applications with distinct memory access patterns: Memcached [10], a high-performance key-value store with 90% GETs, 10% SETs, and a hot key set accessed 90% of the time; PageRank [1], a memory- and compute-intensive graph algorithm execution; and Liblinear [35], a machine learning library running with the KDD12 dataset. Table 2 summarizes the workloads. To minimize CPU interference, we allocate each application to its own dedicated set of 8 CPU cores, with each application running 8 threads. We start Memcached at the beginning and ensure it is warmed up. At 50 seconds, we start PageRank, and at 110 seconds, we start Liblinear.

Table 2: Workloads and RSS in tiered memory.

App	Workload	RSS
Memcached	In-memory database engine using YCSB-C	51 GB
PageRank	Compute the PageRank score of Web pages	42 GB
Liblinear	Linear classification of KDD12 dataset [35]	69 GB

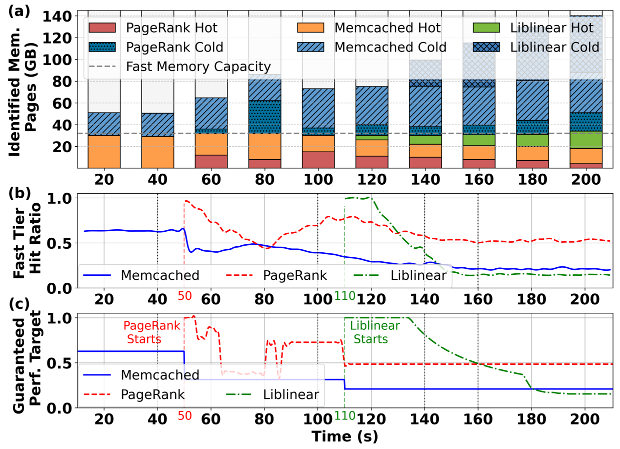

Figure 9: Dynamic memory allocation and measurement of memory tiering performance of co-located workloads.

Figure 9 demonstrates the dynamics of memory allocation and quantifies the performance of memory tiering for three co-located workloads. Figure 9 (a) illustrates the proportion of hot and cold pages in fast and slow tiers, showing how memory is allocated among the workloads. Figure 9 (b) provides dynamic measurements of memory tiering performance, showing the fast tier hit ratio over time for each workload. Figure 9 (c) presents the guaranteed performance target during execution, highlighting adjustments to the performance baseline as the RSS and co-location change. These results show Vulcan's ability to dynamically balance resource allocation and optimize memory tiering performance across workloads.

Fairness Model. To evaluate both the fairness in resource distribution and the efficiency of usage over time, we apply *Jain's fairness index* [14] to the cumulative efficiency-adjusted allocation $X_i = \sum_{t=1}^{T} x_i(t) \cdot FTHR_i(t)$, resulting in the *FTHR-weighted Cumulative Jain's Fairness Index (CFI)*:

$$\text{CFI} = \left(\sum_{i=1}^{N} X_i\right)^2 \Bigg/ \left(N \cdot \sum_{i=1}^{N} X_i^2\right) \tag{4}$$

We compute the CFI metric for the three workloads based on their respective fast memory allocations and the measured *FTHR* values over time. Figure 10 (a) shows that Vulcan consistently outperforms TPP, Memtis, and Nomad across all workloads in terms of performance (normalized to the lowest-performing approach, with means plotted over 10 trials and error bars representing 95%

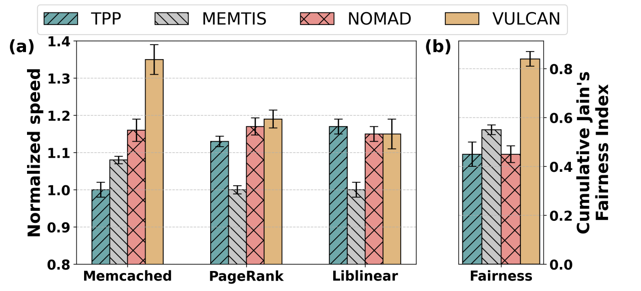

Figure 10: Performance and fairness comparisons of Memcached, PageRank, and Liblinear between TPP, Memtis, Nomad, and Vulcan (higher is better).

confidence intervals). For Memcached, Vulcan achieves approximately 35% higher performance compared to TPP (baseline) and 25% higher than Memtis, showcasing its significant advantage in handling the cold page dilemma for LC workloads. Similarly, in PageRank, Vulcan shows an improvement of approximately 5.3% over TPP and an improvement 19% over Memtis. For Liblinear, Vulcan maintains its superiority with an approximate 15% performance increase compared to Memtis, but slightly underperforms compared to TPP.

In terms of fairness (see Figure 10 (b)), Vulcan achieves greater fairness for fast memory allocation, outperforming Memtis by approximately 52% and Nomad by 86%, demonstrating superior resource allocation and performance, making it the most effective solution for diverse workloads. In conclusion, Vulcan significantly improves performance and fairness, achieving an average performance increase of 12.4% and a fairness improvement of 75.3% over existing solutions. This demonstrates Vulcan's effectiveness in optimizing memory tiering for the multi-tenant environment.

6 Related Work

In this section we discuss the related work that has not been covered in earlier sections. Surprisingly, we are not aware of any other work that comprehensively addresses workload-aware tiered memory management in multi-tenant environments with fairness and isolation. In the following, we discuss the most closely related work.

Page Table Replication. Hydra [11], WASP [28], and Mitosis [3] focus on replicating process-level page tables across NUMA nodes to address cross-NUMA running performance challenges. In contrast, RadixVM [6] implements process-level page table replication at the CPU core level, aiming to eliminate TLB shootdowns, although it faces scalability issues [11]. Our work introduces per-thread page table replication, offering finer-grained control over TLB coherence during page migrations. This orthogonal approach could enhance existing mechanisms by enabling integration of both process-level and per-thread page table replication for joint optimization.

Memory Tiering in User Space. Vulcan's user-space design takes inspiration from recent advancements in user-space system architectures, such as ExtMem [15], which elevates memory management policies to the user space. Similarly, HeMem [29] introduces a tiered memory management system optimized for DRAM and NVM, leveraging asynchronous memory tracking and migration to improve scalability and reduce overhead. While HeMem achieves significant performance gains, its lack of application-specific adaptability can result in suboptimal resource allocation for multi-workloads. More recently, ArtMem [42] introduces reinforcement learning for adaptive migration, yet mainly focuses on system-wide throughput and overlooks fairness and QoS in multi-tenant scenarios. In contrast, Vulcan addresses these challenges with workload-aware migration and QoS-driven fair resource partitioning, achieving both efficiency and fairness in multi-tenant environments. Moreover, Vulcan is compatible with reinforcement learning approaches like those in ArtMem, and can incorporate such techniques to further adapt migration policies for individual workloads.

QoS in Memory Tiering. MaxMem [30] is a tiered memory management system designed to optimize the performance of big data workloads in co-location scenarios. It requires manual QoS settings, which can be cumbersome for operations with various workloads. TMTS [8], on the other hand, focuses on reducing costs by limiting slow-tier capacity while maintaining strict performance bounds, which can restrict scalability. Recently, Soar and Alto [21] move beyond hotness-based tiering by introducing the Amortized Offcore Latency (AOL) metric, which serves as a fine-grained QoS indicator. However, AOL still requires manual threshold configuration. In contrast, Vulcan automates QoS configuration and incorporates fairness-aware mechanisms, providing a more scalable and practical solution for diverse workloads. Notably, Vulcan can be further enhanced by integrating AOL-based metrics.

7 Conclusion

In this paper, we identify the cold page dilemma and hidden costs of page migration as critical challenges faced by existing tiered memory systems within multi-tenant workloads on multi-core platforms. To address these issues, we propose Vulcan, a workload-aware tiered memory management framework that targets fair and efficient tiering in multi-tenant environments. Vulcan introduces four innovations: workload-dependent migration, QoS-aware fair resource partitioning, per-thread page table replication, and biased page migration policy. Evaluation shows that Vulcan significantly outperforms state-of-the-art solutions in both performance guarantee and fairness metrics.

References

[1] Wikipedia [n. d.]. *PageRank*. Wikipedia. https://en.wikipedia.org/wiki/PageRank
[2] 2024. Linux Kernel BPF Documentation. https://docs.kernel.org/bpf/index.html. Accessed: 2024-12-23.
[3] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently self-replicating page-tables for large-memory machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 283–300.
[4] Nadav Amit, Amy Tai, and Michael Wei. 2020. Don't shoot down TLB shootdowns!. In *Proceedings of the Fifteenth European Conference on Computer Systems* (Heraklion, Greece) *(EuroSys '20)*. Association for Computing Machinery, New York, NY, USA, Article 35, 14 pages.
[5] Yu Chen, Ivy Bo Peng, Zhen Peng, Xu Liu, and Bin Ren. 2020. ATMem: adaptive data placement in graph applications on heterogeneous memories. In *CGO '20: 18th ACM/IEEE International Symposium on Code Generation and Optimization, San Diego, CA, USA, February, 2020.* ACM, 293–304. doi:10.1145/3368826.3377922
[6] Austin T. Clements, M. Frans Kaashoek, and Nickolai Zeldovich. 2013. RadixVM: scalable address spaces for multithreaded applications. In *Eighth Eurosys Conference 2013, EuroSys '13, Prague, Czech Republic, April 14-17, 2013*, Zdenek Hanzálek, Hermann Härtig, Miguel Castro, and M. Frans Kaashoek (Eds.). ACM, 211–224.

[7] Fernando J. Corbató, Marjorie Merwin-Daggett, and Robert C. Daley. 1962. An experimental time-sharing system. In *Proceedings of the May 1-3, 1962, Spring Joint Computer Conference* (San Francisco, California) *(AIEE-IRE '62 (Spring))*. Association for Computing Machinery, New York, NY, USA, 335–344.

[8] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David E. Culler, Zhiyi Xu, Jianing Fan, Christopher 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 *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 727–741.

[9] Nima Elyasi, Changho Choi, and Anand Sivasubramaniam. 2019. Large-Scale Graph Processing on Emerging Storage Devices. In *17th USENIX Conference on File and Storage Technologies, FAST 2019, Boston, MA, February 25-28, 2019*. USENIX Association, 309–316.

[10] Brad Fitzpatrick. 2025. Memcached – a distributed memory object caching. https://memcached.org/. Accessed: 2025-04-23.

[11] Bin Gao, Qingxuan Kang, Hao-Wei Tee, Kyle Timothy Ng Chu, Alireza Sanaee, and Djordje Jevdjic. 2024. Scalable and Effective Page-table and TLB management on NUMA Systems. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. USENIX Association, Santa Clara, CA, 445–461.

[12] Taekyung Heo, Yang Wang, Wei Cui, Jaehyuk Huh, and Lintao Zhang. 2022. Adaptive Page Migration Policy With Huge Pages in Tiered Memory Systems. *IEEE Trans. Computers* 71, 1 (2022), 53–68.

[13] Intel. 2024. Intel 64 and IA-32 Architectures Software Developer's Manual. https://cdrdv2.intel.com/v1/dl/getContent/671200.

[14] Rajendra K Jain, Dah-Ming W Chiu, William R Hawe, et al. 1984. A quantitative measure of fairness and discrimination. *Eastern Research Laboratory, Digital Equipment Corporation, Hudson, MA* 21, 1 (1984).

[15] Sepehr Jalalian, Shaurya Patel, Milad Rezaei Hajidehi, Margo I. Seltzer, and Alexandra Fedorova. 2024. ExtMem: Enabling Application-Aware Virtual Memory Management for Data-Intensive Applications. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*, Saurabh Bagchi and Yiying Zhang (Eds.). USENIX Association, 397–408.

[16] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *Proceedings of the 2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*. USENIX Association, 715–728.

[17] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 705–721.

[18] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*. ACM, 17–34.

[19] Baptiste Lepers and Willy Zwaenepoel. 2023. Johnny Cache: the End of DRAM Cache Conflicts (in Tiered Main Memory Systems). In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 519–534.

[20] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, et al. 2023. Pond: Cxl-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 574–587.

[21] Jinshu Liu, Hamid Hadian, Hanchen Xu, and Huaicheng Li. 2025. Tiered Memory Management Beyond Hotness. In *19th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2025, Boston, MA, USA, July 7-9, 2025*, Lidong Zhou and Yuanyuan Zhou (Eds.). USENIX Association, 731–747.

[22] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. 2022. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems. In *IEEE International Symposium on High-Performance Computer Architecture, HPCA 2022, Seoul, South Korea, April 2-6, 2022*. IEEE, 925–937.

[23] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*. ACM, 742–755.

[24] Alan Nair, Sandeep Kumar, Aravinda Prasad, Ying Huang, Andy Rudoff, and Sreenivas Subramoney. 2024. Telescope: Telemetry for Gargantuan Memory Footprint Applications. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 409–424.

[25] Chris Petersen. 2021. Software Defined Memory: A Meta perspective. https://www.opencompute.org/events/past-events/2021-ocp-global-summit

[26] Pmem. 2024. Syscall_intercept. https://github.com/pmem/syscall_intercept.

[27] Zhenlin Qi, Shengan Zheng, Ying Huang, Yifeng Hui, Bowen Zhang, Linpeng Huang, and Hong Mei. 2025. Chrono: Meticulous Hotness Measurement and Flexible Page Migration for Memory Tiering. In *Proceedings of the Twentieth European Conference on Computer Systems*. 835–853.

[28] Hongliang Qu and Zhibin Yu. 2024. WASP: Workload-Aware Self-Replicating Page-Tables for NUMA Servers. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024*, Rajiv Gupta, Nael B. Abu-Ghazaleh, Madan Musuvathi, and Dan Tsafrir (Eds.). ACM, 1233–1249.

[29] 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 *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 392–407.

[30] Amanda Raybuck, Wei Zhang, Kayvan Mansoorshahi, Aditya K. Kamath, Mattan Erez, and Simon Peter. 2023. MaxMem: Colocation and Performance for Big Data Applications on Tiered Main Memory Servers. *CoRR* abs/2312.00647 (2023). doi:10.48550/ARXIV.2312.00647 arXiv:2312.00647

[31] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *Proceedings of the Nineteenth European Conference on Computer Systems, EuroSys 2024, Athens, Greece, April 22-25, 2024*. ACM, 803–817.

[32] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K Aguilera, and Adam Belay. 2020. AIFM:High-Performance,Application-Integrated far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 315–332.

[33] Michael Schwarz. 2024. PTEditor. https://github.com/misc0110/PTEditor. Accessed: 2024-12-23.

[34] Wenda Tang, Senbo Fu, Yutao Ke, Qian Peng, and Feng Gao. 2022. Themis: Fair Memory Subsystem Resource Sharing with Differentiated QoS in Public Clouds. In *Proceedings of the 51st International Conference on Parallel Processing, ICPP 2022, Bordeaux, France, 29 August 2022 - 1 September 2022*. ACM, 49:1–49:12. doi:10.1145/3545008.3545064

[35] National Taiwan University. [n. d.]. *A Library for Large Linear Classification*. https://www.csie.ntu.edu.tw/~cjlin/liblinear

[36] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 559–572.

[37] Midhul Vuppalapati and Rachit Agarwal. 2024. Tiered Memory Management: Access Latency is the Key!. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*. ACM, 79–94.

[38] Midhul Vuppalapati, Giannis Fikioris, Rachit Agarwal, Asaf Cidon, Anurag Khandelwal, and Éva Tardos. 2023. Karma: Resource Allocation for Dynamic Demands. In *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. USENIX Association, Boston, MA, 645–662.

[39] Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan, and Ren Wang. 2024. Nomad: Non-Exclusive Memory Tiering via Transactional Page Migration. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 19–35.

[40] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. 2024. FlexMem: Adaptive Page Profiling and Migration for Tiered Memory. In *Proceedings of the 2024 USENIX Annual Technical Conference, USENIX ATC 2024, Santa Clara, CA, USA, July 10-12, 2024*. 817–833.

[41] Zi Yan, Daniel Lustig, David W. Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. ACM, 331–345.

[42] Xinyue Yi, Hongchao Du, Yu Wang, Jie Zhang, Qiao Li, and Chun Jason Xue. 2025. ArtMem: Adaptive Migration in Reinforcement Learning-Enabled Tiered Memory. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture, ISCA 2025, Tokyo, Japan, June 21-25, 2025*. ACM, 405–418. doi:10.1145/3695053.3731001

[43] Kaiyang Zhao, Sishuai Gong, and Pedro Fonseca. 2021. On-demand-fork: a microsecond fork for memory-intensive and latency-sensitive applications. In *Proceedings of the Sixteenth European Conference on Computer Systems* (Online Event, United Kingdom) *(EuroSys '21)*. Association for Computing Machinery, New York, NY, USA, 540–555.

[44] Yuhong Zhong, Daniel S. Berger, Carl A. Waldspurger, Ryan Wee, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D. Hill, Mosharaf Chowdhury, and Asaf Cidon. 2024. Managing Memory Tiers with CXL in Virtualized Environments. In *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*. USENIX Association, 37–56.