NOMAD: Non-Exclusive Memory Tiering via Transactional Page Migration

Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, and Jia Rao,
The University of Texas at Arlington; Yifan Yuan and Ren Wang, *Intel Labs*

https://www.usenix.org/conference/osdi24/presentation/xiang

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

Open access to the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation is sponsored by

NOMAD: Non-Exclusive Memory Tiering via Transactional Page Migration

Lingfeng Xiang, Zhen Lin, Weishu Deng, Hui Lu, Jia Rao, Yifan Yuan†, Ren Wang†
The University of Texas at Arlington, †Intel Labs

Abstract

With the advent of byte-addressable memory devices, such as CXL memory, persistent memory, and storage-class memory, tiered memory systems have become a reality. Page migration is the *de facto* method within operating systems for managing tiered memory. It aims to bring hot data whenever possible into fast memory to optimize the performance of data accesses while using slow memory to accommodate data spilled from fast memory. While the existing research has demonstrated the effectiveness of various optimizations on page migration, it falls short of addressing a fundamental question: Is exclusive memory tiering, in which a page is either present in fast memory or slow memory, but not both simultaneously, the optimal strategy for tiered memory management?

We demonstrate that page migration-based exclusive memory tiering suffers significant performance degradation when fast memory is under pressure. In this paper, we propose *non-exclusive* memory tiering, a page management strategy that retains a copy of pages recently promoted from slow memory to fast memory to mitigate memory thrashing. To enable non-exclusive memory tiering, we develop NOMAD, a new page management mechanism for Linux that features *transactional page migration* and *page shadowing*. NOMAD helps remove page migration off the critical path of program execution and makes migration completely asynchronous. Evaluations with carefully crafted micro-benchmarks and real-world applications show that NOMAD is able to achieve up to 6x performance improvement over the state-of-the-art transparent page placement (TPP) approach in Linux when under memory pressure. We also compare NOMAD with a recently proposed hardware-assisted, access sampling-based page migration approach and demonstrate NOMAD's strengths and potential weaknesses in various scenarios.

1 Introduction

As new memory devices, such as high bandwidth memory (HBM) [4, 30], DRAM, persistent memory [7, 39], Compute Express Link (CXL)-based memory [1, 37, 44], and storage-class memory [53, 55] continue to emerge, future computer systems are anticipated to feature multiple tiers of memory with distinct characteristics, such as speed, size, power, and cost. Tiered memory management aims to leverage the strength of each memory tier to optimize the overall data access latency and bandwidth. Central to tiered memory management is *page management* within operating systems (OS), including page allocation, placement, and migration. Efficient page management in the OS is crucial for optimizing memory utilization and performance while maintaining transparency for user applications.

Traditionally, the memory hierarchy consists of storage media with at least one order of magnitude difference in performance. For example, in the two-level memory hierarchy assumed by commercial operating systems for decades, DRAM and disks differ in latency, bandwidth, and capacity by 2-3 orders of magnitude. Therefore, the sole goal of page management is to keep hot pages in, and maximize the hit rate of the "performance" tier (DRAM), and migrate (evict) cold pages to the "capacity" tier (disk) when needed. As new memory devices emerge, the performance gap in the memory hierarchy narrows. Evaluations on Intel's Optane persistent memory [56] and CXL memory [50] reveal that these new memory technologies can achieve comparable performance to DRAM in both latency and bandwidth, within a range of 2-3x. As a result, the assumption of the performance gap, which has guided the design of OS page management for decades, may not hold. It is no longer beneficial to promote a hot page to the performance tier if the migration cost is too high.

Furthermore, unlike disks which must be accessed through the file system as a block device, new memory devices are byte-addressable and can be directly accessed by the processor via ordinary `load` and `store` instructions. Therefore, for a warm page on the capacity tier, accessing the page directly and avoiding migration to the performance tier could be a better option. Most importantly, while the performance of tiered memory remains hierarchical, the hardware is no longer hierarchical. Both the Optane persistent memory and

CXL memory appear to the processor as a CPUless memory node and thus can be used by the OS as ordinary DRAM.

These unique challenges facing emerging tiered memory systems have inspired research on improving page management in the OS. Much focus has been on expediting page migrations between memory tiers. Nimble [54] improves page migration by utilizing transparent huge pages (THP), multi-threaded migration of a page, and concurrent migration of multiple pages. Transparent page placement (TPP) [44] extends the existing NUMA balancing scheme in Linux to support asynchronous page demotion and synchronous page promotion between fast and slow memory. Memtis [37] and TMTS [24] use hardware performance counters to mitigate the overhead of page access tracking and use background threads to periodically and asynchronously promote pages.

However, these approaches have two fundamental limitations. *First*, the existing page management for tiered memory assumes that memory tiers are *exclusive* to each other – hot pages are allocated or migrated to the performance tier while cold pages are demoted to the capacity tier. Therefore, each page is only present in one tier. As memory tiering seeks to explore the tradeoff between performance and capacity, the working set size of workloads that benefit most from tiered memory systems likely exceeds the capacity of the performance tier. Exclusive memory tiering inevitably leads to excessive hot-cold page swapping or memory thrashing when the performance tier is not large enough to hold hot data.

Second, there is a lack of an efficient page migration mechanism to support tiered memory management. As future memory tiers are expected to be addressable by the CPU, page migrations are similar to serving minor page faults and involve three steps: 1) unmap a page from the page table; 2) copy the page content to a different tier; 3) remap the page on the page table, pointing to the new memory address. Regardless of whether page migration is done synchronously upon accessing a hot page in the slower capacity tier or asynchronously in the background, the 3-step migration process is expensive. During migration, an unmapped page cannot be accessed by user programs. If page migration is done frequently, e.g., due to memory thrashing, user-perceived bandwidth, including accesses to the migrating pages, is significantly lower (up to 95% lower) than the peak memory bandwidth [54].

This paper advocates *non-exclusive* memory tiering that allows a subset of pages on the performance tier to have shadow copies on the capacity tier [1]. Note that non-exclusive tiering is different from *inclusive* tiering which strictly uses the performance tier as a cache of the capacity tier. The most important benefit is that under memory pressure, page demotion is made less expensive by simply remapping a page if it is not dirty and its shadow copy exists on the capacity tier. This allows for smooth performance transition when memory demand exceeds the capacity of the performance tier.

[1]We assume that page migrations only occur between two adjacent tiers if there are more than two memory tiers.

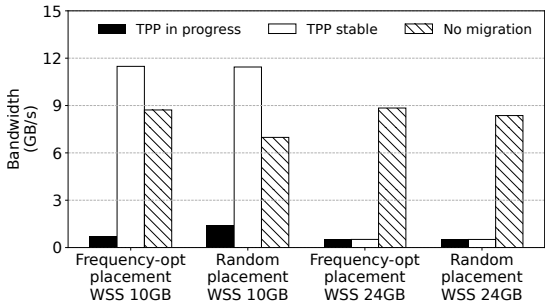


Figure 1: The comparison of achieved memory bandwidth in a micro-benchmark due to different phases in TPP and a baseline approach that disables page migration. Higher is better performance.

To reduce the cost of page migration, especially for promotion, this paper proposes *transactional page migration* (TPM), a novel mechanism to enable page access during migration. Unlike current page migrations, TPM starts page content copy without unmapping the page from the capacity tier so that the migrating page is still accessible by user programs. After page content is copied to a new page on the performance tier, TPM checks whether the page has been dirtied during the migration. If so, the page migration (i.e., the transaction) is invalidated and the copied page is discarded. Failed page migrations will be retried at a later time. If successful, the copied new page is mapped in the page table and the old page is unmapped, becoming a shadow copy of the new page.

We have developed NOMAD, a new page management framework for tired memory that integrates non-exclusive memory tiering and transactional page migration. NOMAD safeguards page allocation to prevent out-of-memory (OOM) errors due to page shadowing. When the capacity tier is under memory pressure, NOMAD prioritizes the reclamation of shadow pages before evicting ordinary pages. We have implemented a prototype of NOMAD in Linux and performed a thorough evaluation on four different platforms, including an FPGA-based CXL prototype, a persistent memory system, and a pre-market, commercial CXL system. Experimental results show that, compared to two representative page management schemes: TPP and Memtis, NOMAD achieves up to 6x performance improvement over TPP during memory thrashing and consistently outperforms Memtis by as much as 130% when the working set size fits into fast memory.

2 Motivation and Related Work

We introduce the background of page management in tiered memory systems and use TPP [44], a state-of-the-art page placement system designed for CXL-enabled tiered memory, as a motivating example to highlight the main limitations of current page management approaches.

2.1 Memory Tiering

Caching and tiering are two traditional software approaches to manage a memory, or storage hierarchy, consisting of various types of storage media (e.g., CPU caches, DRAM, and hard disks) differing in performance, capacity, and cost. Without loss of generality, we consider a two-level memory hierarchy with 1) a *performance* tier (i.e., the fast tier), backed with smaller, faster, but more expensive storage media; and 2) a *capacity* tier (i.e., the slow tier) with larger, slower, and cheaper storage media. For *caching*, data is stored in the capacity tier, and copies of frequently accessed or "hot" data are strategically replicated to the performance tier. For *tiering*, new data is first allocated to the performance tier and remains there if it is frequently accessed, while less accessed data may be relegated to the capacity tier when needed. At any moment, data resides exclusively in one of the tiers but not both. Essentially, caching operates in an *inclusive* page placement mode and retains pages in their original locations, only temporarily storing a copy in the performance tier for fast access. Conversely, tiering operates in an *exclusive mode*, actively relocating pages across various memory/storage mediums.

Diverse memory/storage devices, such as high bandwidth memory (HBM) [4], CXL-based memory [1], persistent memory (PM) [7], and fast, byte-addressable NVMe SSDs [31], have emerged recently. While they still make a tradeoff between speed, size, and cost, the gap between their performance narrows. For example, Intel Optane DC persistent memory (PM), available in a DIMM package on the memory bus enabling programs to directly access data from the CPU using `load` and `store` instructions, provides (almost) an order of magnitude higher capacity than DRAM (e.g., 8x) and offers performance within a range of 2-3x of DRAM, e.g., write latency as low as 80 *ns* and read latency around 170 *ns* [56]. More recently, compute express link (CXL), an open-standard interconnect technology based on PCI Express (PCIe) [1], provides a memory-like, byte-addressable interface (i.e., via the `CXL.mem` protocol) for connecting diverse memory devices (e.g., DRAM, PM, GPUs, and smartNICs). Real-world CXL memory offers comparable memory access latency (<2x) and throughput ($\sim$50%) to ordinary DRAM [50].

From the perspective of OS memory management, CXL memory or PM appears to be a remote, CPUless memory node, similar to a multi-socket non-uniform memory access (NUMA) node. State-of-the-art tiered memory systems, such as TPP [44], Memtis [37], Nimble [54], and AutoTiering [32], all adopt *tiering* to *exclusively* manage data on different memory tiers. Unlike the traditional two-level memory hierarchy involving DRAM and disks, in which DRAM acts as a cache for the much larger storage tier, current CXL memory tiering treats CXL memory as an extension of local DRAM. While exclusive memory tiering avoids data redundancy, it necessitates data movement between memory tiers to optimize the performance of data access, i.e., promoting hot data to the fast tier and demoting cold data to the slow tier. Given that all memory tiers are byte-addressable by the CPU and the performance gap between tiers narrows, it remains to be seen whether exclusive tiering is the optimal strategy considering the cost of data movement.

We evaluate the performance of transparent page placement (TPP) [44], a state-of-the-art and the default tiered memory management in Linux. Figure 1 shows the bandwidth of a micro-benchmark that accesses a configurable working set size (WSS) following a Zipfian distribution in a CXL-based tiered memory system. More details of the benchmark and the hardware configurations can be found in Section 4. We compare the performance of TPP while it actively migrates pages between tiers for promotion and demotion (denoted as *TPP in progress*) and when it has finished page relocation (*TPP stable*) with that of a baseline that disables page migration (*no migration*). The baseline does not optimize page placement and directly accesses hot pages from the slow tier. The tiered memory testbed is configured with 16GB fast memory (local DRAM) and 16GB slow memory (remote CXL memory). We vary the WSS to fit in (e.g., 10GB) and exceed (e.g., 24GB) fast memory capacity. Note that the latter requires continuous page migrations between tiers since hot data spills into slow memory. Additionally, we explore two initial data placement strategies in the benchmark. First, the benchmark pre-allocates 10GB of data in fast memory to emulate the existing memory usage from other applications. *Frequency-opt* is an allocation strategy that places pages according to the descending order of their access frequencies (hotness). Thus, the hottest pages are initially placed in fast memory until the WSS spills into slow memory. In contrast, *Random* employs a random allocation policy and may place cold pages initially in fast memory.

We have important observations from results in Figure 1. First, page migration in TPP incurs significant degradation in application performance. When WSS fits in fast memory, *TPP stable*, which has successfully migrated all hot pages to fast memory, achieves more than an order of magnitude higher bandwidth than *TPP in progress*. Most importantly, *no migration* is consistently and substantially better than *TPP in progress*, suggesting that the overhead of page migration outweighs its benefit until the migration is completed. Second, TPP never reaches a stable state and enters memory thrashing when WSS is larger than the capacity of fast memory. Third, page migration is crucial to achieving optimal performance if it is possible to move all hot data to fast memory and the initial placement is sub-optimal, as evidenced by the wide gap between *TPP stable* and *no migration* in the 10GB WSS and random placement test.

2.2 Page Management

In this section, we delve into the design of page management in Linux and analyze its overhead during page migration. We

focus our discussions on 1) how to effectively track memory accesses and identify hot pages, and 2) the mechanism to migrate a page between memory tiers.

Tracking memory access can be conducted by software (via the kernel) and/or with hardware assistance. Specifically, the kernel can keep track of page accesses via page faults [2, 32, 44], scanning page tables [2, 14, 19, 43, 54], or both. Capturing each memory access for precise tracking can be expensive. Page fault-based tracking traps memory accesses to selected pages (i.e., whose page table entry permissions are set to `no access`) via hint (minor) page faults. Thus, it allows the kernel to accurately measure the *recency* and *frequency* of these pages. However, invoking a page fault on every memory access incurs high overhead on the critical path of program execution. On the other hand, page table (PT) scanning periodically checks the `access` bit in all page table entries (PTE) to determine recently accessed pages since the last scanning. Compared to page fault-based tracking, which tracks every access on selected pages, PT scanning has to make a tradeoff between scanning overhead and tracking accuracy by choosing an appropriate scanning interval [37].

Linux adopts a *lazy* PT scanning mechanism to track hot pages, which lays the foundation for its tiered memory management. Linux maintains two LRU lists for a memory node: an *active* list to store hot pages and an *inactive* list for cold pages. By default, all new pages go to the inactive list and will be promoted to the active list according to two flags, `PG_referenced` and `PG_active`, in the per-page struct `page`. `PG_reference` is set when the `access` bit in the corresponding PTE is set upon a PTE check and `PG_active` is set after `PG_reference` is set for two consecutive times. A page is promoted to the active list when its `PG_active` flag is set. For file-backed pages, their accesses are handled by the OS through the file system interface, e.g., `read()` and `write()`. Therefore, their two flags are updated each time they are accessed. For anonymous pages, e.g., application memory allocated through `malloc`, since page accesses are directly handled by the MMU hardware and bypass the OS kernel, the updates to their reference flags and LRU list management are only performed during memory reclamation. Under memory pressure, the swapping daemon `kswapd` scans the inactive list and the corresponding PTEs to update inactive pages' flags, and reclaims/swaps out those with `PG_reference` unset. Additionally, `kswapd` promotes hot pages (i.e., those with `PG_active` set) to the active list. This lazy scanning mechanism delays access tracking until it is necessary to reduce the tracking overhead, but undermines tracking accuracy.

TPP [44] leverages Linux's PT scanning to track hot pages and employs page fault-based tracking to decide whether to promote pages from slow memory. Specifically, TPP sets all pages residing in slow memory (e.g., CXL memory) as `inaccessible`, and any user access to these pages will trigger a minor page fault, during which TPP decides whether to promote the faulting page. If the faulting page is on the active list, it is migrated (promoted) to the fast tier. Page demotion occurs when fast memory is under pressure and `kswapd` migrates pages from the inactive list to slow memory.

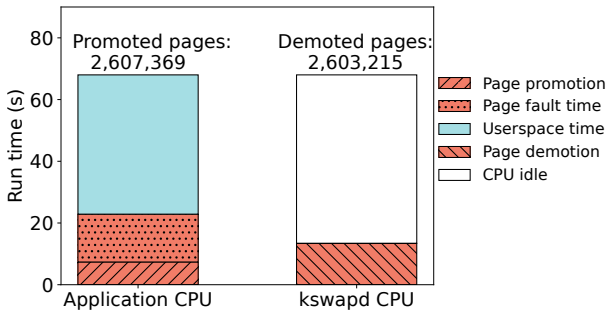


Figure 2: Time breakdown in the execution of *TPP in progress*: Synchronous page migration and page fault handling account for a significant portion of the runtime.

Accurate and lightweight memory access tracking can be achieved with hardware support, e.g., by adding a PTE *count field* in hardware that records the number of memory accesses [45]. However, hardware-based tracking can increase the complexity and require extensive hardware changes in mainstream architectures (e.g., x86). In practice, the hardware-assisted sampling, such as via Processor Event-Based Sampling (PEBS) [24, 37] on Intel platforms, has been employed to record page access (virtual address) information from sampled hardware events (e.g., LLC misses or store instructions). However, PEBS-based profiling also requires a careful balance between the frequency of sampling and the accuracy of profiling. We observed that the PEBS-based approach [37], with a sampling rate optimized for minimizing overhead, remains coarse-grained and fails to capture many hot pages. Further, the sampling-based approach may not accurately measure access recency, thus limiting its ability to make timely migration decisions.

Page migration between memory tiers involves a complex procedure: ① The system must trap to the kernel (e.g., via page faults) to handle migration; ② The PTE of a migrating page must be locked to prevent others from accessing the page during migration and be `unmapped` from the page table; ③ A translation lookaside buffer (TLB) shootdown must be issued to each processor (via inter-processor interrupts (IPIs)) that may have cached copies of the stale PTE; ④ The content of the page is copied between tiers; ⑤ Finally, the PTE must be `remapped` to point to the new location. Page migration can be done *synchronously* or *asynchronously*. Synchronous migration, e.g., page promotion in TPP, is on-demand triggered by user access to a page and on the critical path of program execution. During migration, the user program is blocked until migration is completed. Asynchronous migration, e.g., page demotion in TPP, is handled by a kernel thread (i.e., `kswapd`), oftentimes off programs' critical path, when certain criteria are met. Synchronous migration is costly not only because pages are inaccessible during migration but also may involve

a large number of page faults.

Figure 2 shows the run time breakdown of the aforementioned benchmark while TPP is actively relocating pages between the two memory tiers. Since page promotion is synchronous, page fault handling and page content copying (i.e., promotion) are executed on the same CPU as the application thread. Page demotion is done through `kswapd` and uses a different core. As shown in Figure 2, synchronous promotion together with page fault handling incurs significant overhead on the application core. In contrast, the demotion core remains largely idle and does not present a bottleneck. As will be discussed in Section 3.1, userspace run time can also be prolonged due to repeated minor page faults (as many as 15) to successfully promote one page. This overhead analysis explains the poor performance of TPP observed in Figure 1.

2.3 Related Work

A long line of pioneering work has explored a wide range of tiered storage/memory systems, built upon SSDs and HDDs [12, 15, 22, 26, 33, 41, 49, 52, 57], DRAM and disks [21, 25, 29, 46], HBM and DRAMs [23, 45, 48], NUMA memory [2, 3], PM and DRAM [13, 14, 40, 43], local and far memory [19, 27, 34, 35, 47], DRAM and CXL memory [37, 38, 44], and multiple tiers [32, 36, 40, 51, 54]. We focus on tiered memory systems consisting of DRAM and the emerging byte-addressable memory devices, e.g., CXL memory and PM. NOMAD also applies to other tiered memory systems such as HBM/DRAM and DRAM/PM.

Lightweight memory access tracking. To mitigate software overhead associated with memory access tracking, Hotbox [19] employs two separate scanners for fast and slow tiers to scan the slow tier at a fixed rate while the fast tier at an adaptive rate, configurable based on the local memory pressure. Memtis [37] adjusts its PEBS-based sampling rate to ensure its overhead is under control (e.g., < 3%). TMTS [24] also adopts a periodic scanning mechanism to detect frequency along with hardware sampling to more timely detect newly hot pages. While these approaches balance scanning/sampling overhead and tracking accuracy, an "always-on" profiling component does not seem practical, especially for high-pressure workloads. Instead, thermostat [14] samples a small fraction of pages, while DAMON [3] monitors memory access at a coarser-grained granularity (i.e., region). Although both can effectively reduce the scanning overhead, coarse granularity leads to lower accuracy regarding page access patterns. On the other hand, to reduce the overhead associated with frequent hint page faults like AutoNUMA [2], TPP [44] enables the page-fault based detection only for CXL memory (i.e., the slow tier) and tries to promote a page promptly via synchronous migration; prompt page promotion avoids subsequent page faults on the same page.

Inspired by existing lightweight tracking systems, such as Linux's active and inactive lists and hint page faults, NOMAD advances them by incorporating more recency information with *no* additional CPU overhead. Unlike hardware-assisted approaches [24, 37, 42], NOMAD does not require any additional hardware support.

Page migration optimizations. To hide reclamation overhead from applications, TPP [44] decouples page allocation and reclamation; however, page migration remains in the critical path, incurring significant slowdowns. Nimble [54] focuses on mitigating page migration overhead with new migration mechanisms, including transparent huge page migration and concurrent multi-page migration. Memtis [37] further moves page migration out of the critical path using a kernel thread to promote/demote pages in the background. TMTS [24] leverage a user/kernel collaborative approach to control page migration. In contrast, NOMAD aims to achieve prompt, on-demand page migration while moving page migration off the critical path. It is orthogonal to and can benefit from existing page migration optimizations. The most related work is [20], which leverages hardware support to pin data in caches, enabling access to pages during migration. Again, NOMAD does not need additional hardware support.

3 NOMAD Design and Implementation

NOMAD is a new page management mechanism for tiered memory that features *non-exclusive memory tiering* and *transactional page migration*. The goal of NOMAD design is to enable the processor to freely access pages from both fast and slow memory tiers and move the cost of page migration off the critical path of users' data access. Note that NOMAD does not make page migration decisions and relies on the existing memory access tracking in the OS to determine page temperature. Furthermore, NOMAD does not impact the initial memory allocation in the OS and assumes a standard page placement policy. Pages are allocated from the fast tier whenever possible and are placed in the slower tier only when there is an insufficient number of free pages in the fast tier, or attempts to reclaim memory in the fast tier have failed. After the initial page placement, NOMAD gradually migrates hot pages to the fast tier and cold pages to the slow tier. NOMAD seeks to address two key issues: 1) *how to minimize the cost of page migration?* 2) *how to minimize the number of migrations?*

Overview. Inspired by multi-level cache management in modern processors, which do not employ a purely inclusive or exclusive caching policy between tiers [16] to facilitate the sharing of or avoid the eviction of certain cache lines, NOMAD embraces a *non-exclusive* memory tiering policy to prevent memory thrashing when under memory pressure. Unlike the existing page management schemes that move pages between tiers and require that a page is only present in one tier, NOMAD instead copies pages from the slow tier to the fast tier and keeps a shadow copy of the migrated pages at the slow tier. The non-exclusive tiering policy maintains shadow copies

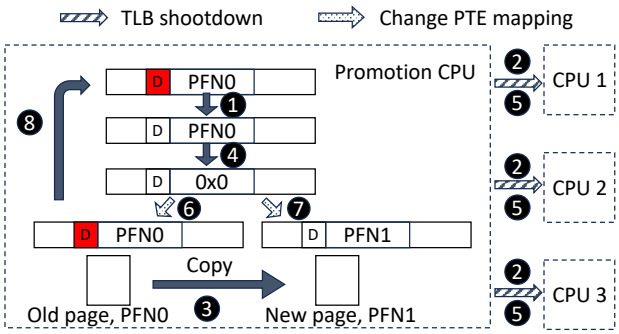


Figure 3: The workflow of transactional page migration. PFN is the page frame number and D is the dirty bit in PTE. The page is only inaccessible by user programs during step 4 when the page is remapped in the page table.

only for pages that have been promoted to the fast tier, thereby not an inclusive policy. The advantage of the non-exclusive policy is that the demotion of clean, cold pages can be simplified to remapping the page table entry (PTE) without the need to copy the cold page to the slower tier.

The building block of NOMAD is a new *transactional page migration* (TPM) mechanism to reduce the cost of page migrations. Unlike the existing unmap-copy-remap 3-step page migration, TPM opportunistically copies a page without unmapping it from the page table. During the page copy, the page is not locked and can be accessed by a user program. After the copy is completed, TPM checks if the page has been dirtied during the copy. If not, TPM locks the page and remaps it in the PTE to the faster tier. Otherwise, the migration is aborted and will be tried at a later time. TPM not only minimizes the duration during which a page is inaccessible but also makes page migration asynchronous, thereby removing it from the critical path of users' data access.

Without loss of generality, we describe NOMAD design in the context of Linux. We start with transactional page migration and then delve into page shadowing – an essential mechanism that enables non-exclusive memory tiering.

3.1 Transactional Page Migration

The motivation to develop TPM is to make page migration entirely asynchronous and decoupled from users' access to the page. As discussed in Section 2.2, the current page migration in Linux is synchronous and on the critical path of users' data access. For example, the default tiered memory management in Linux, TPP, attempts to migrate a page from the slow tier whenever a user program accesses the page. Since the page is in `inaccessible` mode, the access triggers a minor page fault, leading TPP to attempt the migration. The user program is blocked and makes no progress until the minor page fault is handled and the page is remapped to the fast tier, which can be a time-consuming process. Worse, if the migration fails, the OS remains in function `migrate_pages` and retries the aforementioned migration until it is successful or reaching a maximum of 10 attempts.

TPM decouples page migration from the critical path of user programs by making the migrating page accessible during migration. Therefore, users will access the migrating page from the slow tier before the migration is complete. While accessing a hot page from the slow tier may lead to sub-optimal memory performance, it avoids blocking user access due to the migration, thereby leading to superior user-perceived performance. Figure 3 shows the workflow of TPM. Before migration commences, TPM clears the protection bit of the page frame and adds the page to a migration pending queue. Since the page is no longer protected and not yet unmapped from the page table, following accesses to the page will not trigger additional page faults.

TPM starts a migration transaction by clearing the dirty bit of the page (step ❶) and checks the dirty bit after the page is copied to the fast tier to determine whether the transaction was successful. After changing the dirty bit in PTE, TPM issues a TLB shootdown to all cores that ever accessed this page (step ❷). This is to ensure that subsequent writes to the page can be recorded on the PTE. After the TLB shootdown is completed, TPM starts copying the page from the slow tier to the fast tier (step ❸). To commit the transaction, TPM checks the dirty bit by loading the entire PTE using atomic instruction `get_and_clear` (step ❹). Clearing the PTE is equivalent to unmapping the page and thus another TLB shootdown is needed (step ❺). Note that after unmapping the page from PTE, it becomes inaccessible by users. TPM checks whether the page was dirtied during the page copy (step ❻) and either commits the transaction by remapping the page to the fast tier if the page is clean (step ❼) or otherwise aborts the transaction (step ❽). If the migration is aborted, the original PTE is restored and waits for the next time when TPM is rescheduled to retry the migration. The duration in which the page is inaccessible is between ❹ and ❼/ ❽, significantly shorter than that in TPP (possibly multiple attempts between ❶ and ❼).

Page migration is a complex procedure that involves memory tracing and updates to the page table for page remapping. The state-of-the-art page fault-based migration approaches, e.g., TPP in Linux [44], employ synchronous page migration, a mechanism in the Linux kernel for moving pages between NUMA nodes. In addition to the extended migration time affecting the critical path of user programs, this mechanism causes excessive page faults when integrated with the existing LRU-based memory tracing. TPP makes *per-page* migration decisions based on whether the page is on the *active* LRU list. Nevertheless, in Linux, memory tracing adds pages from the inactive to the active LRU list in batches of 15 requests [2], aiming to minimize the queue management overhead. Due to synchronous page migration, TPP may submit multiple requests (up to 15 if the request queue is empty) for a page to

[2]The 15 requests could be repeated requests for promoting the same page to the active LRU list

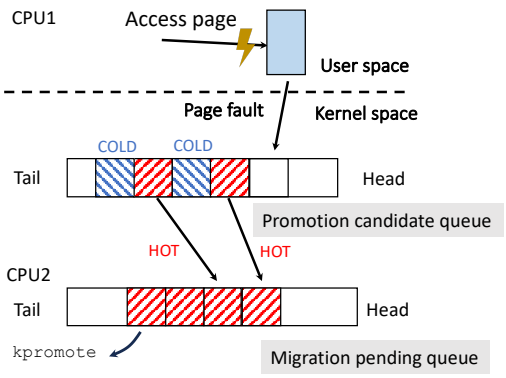


Figure 4: TPM uses a two-queue design to enable asynchronous page migration.

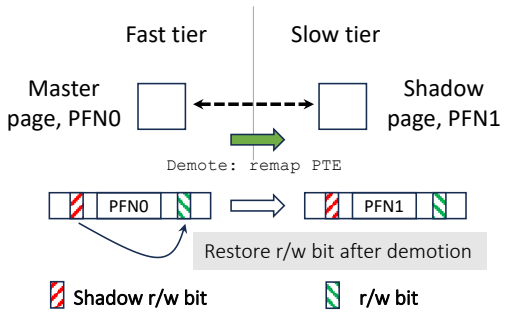


Figure 5: Shadow page management using `shadow r/w` bit.

be promoted to the active LRU list to initiate the migration process. In the worst case, migrating one page may generate as many as 15 minor page faults.

TPM provides a mechanism to enable **asynchronous page migration** but requires additional effort to interface with memory tracing in Linux to minimize the number of page faults needed for page migration. As shown in Figure 4, in addition to the inactive and active LRU lists in memory tracing, TPM maintains a separate *promotion candidate* queue (PCQ) for pages that 1) have been tried for migration but 2) not yet promoted to the active LRU list. Upon each time a minor (hint) page fault occurs and the faulting page is added to PCQ, TPM checks if there are any hot pages in PCQ that have both the `active` and `accessed` bits set. These hot pages are then inserted to a *migration pending queue*, from where they will be tried for asynchronous, transactional migration by a background kernel thread `kpromote`. Note that TPM does not change how Linux determines the temperature of a page. For example, in Linux, all pages in the active LRU list, which are eligible for migration, have the two memory tracing bits set. However, not all pages with these bits set are in the active list due to LRU list management. TPM bypasses the LRU list management and provides a more efficient method to initiate page migration. If all transactional migrations were successful, TPM guarantees that only one page fault is needed per migration in the presence of LRU list management.

3.2 Page Shadowing

To enable non-exclusive memory tiering, NOMAD introduces a one-way *page shadowing* mechanism to allow a subset of pages resident in the performance tier to have a shadow copy in the capacity tier. Only pages promoted from the slow tier have shadow copies in the slow tier. Shadow copies are the original pages residing on the slow tier before they are unmapped in the page table and migrated to the fast tier. Shadow pages play a crucial role in minimizing the overhead of page migration during periods of memory pressure. Instead of swapping hot and cold pages between memory tiers, page shadowing enables efficient page demotion through page table remapping. This would eliminate half of the page migration overhead, i.e., page demotion, during memory thrashing.

Indexing shadow pages. Inspired by the indexing of file-based data in the Linux page cache, NOMAD builds an XArray for indexing shadow pages. An XArray is a radix-tree like, cache-efficient data structure that acts as a key-value store, mapping from the physical address of a fast tier page to the physical address of its shadow copy on the slow tier. Upon successfully completing a page migration, NOMAD inserts the addresses of both the new and old pages into the XArray. Additionally, it adds a new `shadow` flag to the `struct page` of the new page, indicating that shadowing is on for this page.

Shadow page management. The purpose of maintaining shadow pages is to assist with page demotion. Fast or efficient page demotion is possible via page remapping if the master page, i.e., the one on the fast tier, is clean and consistent with the shadow copy. Otherwise, the shadow copy should be discarded. To track inconsistency between the master and shadow copies, NOMAD sets the master page as `read-only` and a write to the page causes a page fault. To simplify system design and avoid additional cross-tier traffic, NOMAD discards the shadow page if the master page is dirtied.

However, tracking updates to the master page poses a significant challenge. Page management in Linux relies heavily on the read-write permission to perform various operations on a page, such as copy-on-write (CoW). While setting master pages as `read-only` effectively captures all writes, it may affect how these master pages are managed in the kernel. To address this issue, NOMAD introduces a procedure called *shadow page fault*. It still designates all master pages as `read-only` but preserves the original read-write permission in an unused software bit on the page's PTE (as shown in Figure 5). We refer to this software bit as `shadow r/w`. Upon a write to a master page, a page fault occurs. Unlike an ordinary page fault that handles write violation, the shadow page fault, which is invoked if the page's `shadow` flag is set in its `struct page`, restores the read-write permission of the faulting page according to the `shadow r/w` bit and discards/frees the shadow page. The write may proceed once the shadow

page fault returns and reinstates the page to be writable. For `read-only` pages, tracking shadow pages does not impose additional overhead; for writable pages, it requires one additional shadow page fault to restore their write permission.

Reclaiming shadow pages. Non-exclusive memory tiering introduces space overhead due to the storage of shadow pages. If shadow pages are not timely reclaimed when the system is under memory pressure, applications may encounter out-of-memory (OOM) errors, which would not occur under exclusive memory tiering. There are two scenarios in which shadow pages should be reclaimed. First, the Linux kernel periodically checks the availability of free pages and if free memory falls below `low_water_mark`, kernel daemon `kswapd` is invoked to reclaim memory. NOMAD instructs `kswapd` to prioritize the reclamation of shadow pages. Second, upon a memory allocation failure, NOMAD also tries to free shadow pages. To avoid OOM errors, the number of freed shadow pages should exceed the number of requested pages. However, frequent memory allocation failures could negatively affect system performance. NOMAD employs a straightforward heuristic to reclaim shadow pages, targeting 10 times the number of requested pages or until all shadow pages are freed. While excessive reclamation may have a negative impact on NOMAD's performance, it is crucial to prevent Out-of-Memory (OOM) errors. Experiments in Section 4 demonstrate the robustness of NOMAD even under extreme circumstances.

3.3 Limitations

NOMAD relies on two rounds of TLB shootdown to effectively track updates to a migrating page during transactional page migration. When a page is used by multiple processes or mapped by multiple page tables, its migration involves multiple TLB shootdowns, per each mapping, that need to happen simultaneously. The overhead of handling multiple IPIs could outweigh the benefit of asynchronous page copy. Hence, NOMAD deactivates transactional page migration for multi-mapped pages and resorts to the default synchronous page migration mechanism in Linux. As high-latency TLB shootdowns based on IPIs continue to be a performance concern, modern processors, such as ARM, future AMD, and Intel x86 processors, are equipped with ISA extensions for faster broadcast-based [17, 18] or micro-coded RPC-like [28] TLB shootdowns. These emerging lightweight TLB shootdown methods will greatly reduce the overhead of TLB coherence in tiered memory systems with expanded memory capacity. NOMAD will also benefit from the emerging hardware and can be extended to scenarios where more intensive TLB shootdowns are necessary.

4 Evaluation

This section presents a thorough evaluation of NOMAD, focusing on its performance, overhead, and robustness. Our primary goal is to understand tiered memory management by comparing NOMAD with existing representative approaches to reveal the benefits and potential limitations of current page management approaches for emerging tiered memory.

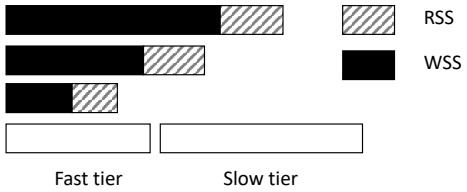


Figure 6: The three memory provisioning schemes used in the evaluation. From bottom to top concerning fast memory: over-provisioning, approaching capacity, and under-provisioning.

We analyze two types of memory footprints: 1) resident set size (RSS) – the total size of memory occupied by a program, and 2) working set size (WSS) – the amount of memory a program actively uses during execution. RSS determines the initial page placement, while WSS dictates the number of pages that should be migrated to the fast tier. Since we focus on in-memory computing, WSS is typically smaller than RSS. Figure 6 illustrates the three scenarios we study with the WSS size smaller than, close to, and larger than fast memory size.

Testbeds. We conducted experiments on *four* platforms with different configurations in CPU, local DRAM, CXL memory, and persistent memory, as detailed in Table 1.

- *Platform A* was built with commercial off-the-shelf (COTS) Intel Sapphire Rapids processors and a 16 GB Agilex-7 FPGA-based CXL memory device [6].
- *Platform B* featured an engineering sample of the Intel Sapphire Rapids processors with the same FPGA-based CXL memory device. The prototype processors have engineering tweaks that have the potential to enhance the performance of CXL memory, which were not available on platform A.
- *Platform C* included an Intel Cascade Lake processor and six 256 GB 100 series Intel Optane Persistent Memory. This platform enabled the full capability of PEBS-based memory tracking and allowed for a comprehensive comparison between page fault- and sampling-based page migration.
- *Platform D* had an AMD Genoa 9634 processor and four 256 GB Micron's (pre-market) CXL memory modules. This platform allowed us to evaluate NOMAD with more realistic CXL memory configurations.

Since the FPGA-based CXL memory device had only 16 GB of memory, we configured local DRAM to 16 GB for all platforms [3]. Note that platform C was equipped with DDR4 DRAM as fast memory while the other platforms used DDR5

[3] Although platform C and D have larger PM or CXL memory sizes, we configured them with 16 GB slow memory consistent with platform A and B for a fair comparison in micro-benchmarks.This limit was lifted when testing real applications.

	Platform A	Platform B (engineering sample)	Platform C	Platform D
CPU	4th Gen Xeon Gold 2.1GHz	4th Gen Xeon Platinum 3.5GHz	2nd Gen Xeon Gold 3.9GHz	AMD Genoa 3.7GHz
Performance tier (DRAM)	16 GB DDR5	16 GB DDR5	16 GB DDR4	16GB DDR5
Capacity tier (CXL or PM Memory)	Agilex 7 16 GB DDR4	Agilex 7 16 GB DDR4	Optane 100 256 GB DDR-T ×6	Micron CXL memory 256GB ×4
Performance tier read latency	316 cycles	226 cycles	249 cycles	391 cycles
Capacity tier read latency	854 cycles	737 cycles	1077 cycles	712 cycles
Performance tier bandwidth (GB/s) Single Thread / Peak performance	Read: 12/31.45 Write: 20.8/28.5	Read: 12/31.2 Write: 22.3/23.67	Read: 12.57/116 Write: 8.67/85	Read: 37.8/270 Write: 89.8/272
Capacity tier bandwidth (GB/s) Single Thread/Peak performance	Read: 4.5/21.7 Write: 20.7/21.3	Read: 4.45/22.3 Write: 22.3/22.4	Read: 4/40.1 Write: 8.1/13.6	Read: 20.25/83.2 Write: 57.7/84.3

Table 1: The configurations of four testbeds and performance characteristics of various memory devices.

Workload Type	In progress Promotion	In progress Demotion	Steady Promotion	Steady demotion
Small WSS	(1.2M|1M)/(15.9K|134K)/ (1.16M|781K)	(2.4M|2.2M)/(15.9K|140K)/ (2.7M|1.5M)	(0|3.3K)/(7.7K|104K)/ (82|74)	(424K|56K)/(0|104K)/ (48K|0)
Medium WSS	(4M|6M)/(0|0)/ (1.6M|5M)	(4.7M|6M)/(2|512)/ (2.5M|4.8M)	(1.8M|3.2M)/(17.4K|0)/ (417K|1.6M)	(1.9M|3.2M)/(16.9K|0)/ (293K|1.4M)
Large WSS	(7M|5.9M)/(0|0)/ (4.5M|7M)	(7.2M|6.5M)/(0|15)/ (4.1M|7.2M)	(7.1M|5.2M)/(0|143K)/ (6.8M|8.8M)	(7.1M|5.3M)/(0|143K)/ (6.8M|8.9M)

Table 2: The number of page promotions/demotions for read|write during the *migration in progress* and the *stable* phases for TPP/Memtis-Default/NOMAD. The data corresponds to Figure 7 for platform A.

DRAM. We evaluated both CXL memory and persistent memory (PM) as slow memory. Table 1 lists the performance characteristics of the four platforms for single-threaded and peak (multi-threaded) performance. While CXL memory and PM have distinct characteristics, including persistence, concurrent performance, and read/write asymmetry, they achieve comparable performance within 2-3x of DRAM and provide a similar programming interface as a CPUless memory node. To ensure a fair comparison, we only enabled one socket on each of the four platforms. Intel platforms were configured with 32 cores while the AMD platform had 84 cores.

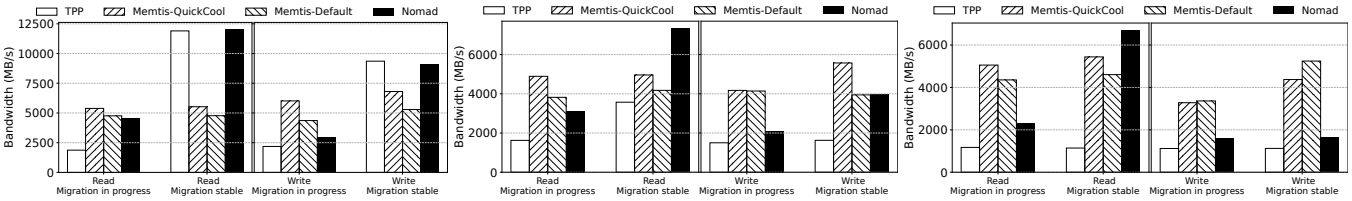
Baselines for comparison. We compared NOMAD with two state-of-the-art tired memory systems: TPP [44] and Memtis [37]. We evaluated both TPP and Nomad on Linux kernel v5.13-rc6 and ran Memtis on kernel v5.15.19, the kernel version upon which Memtis was built and released. We tested two versions of Memtis – Memtis-Default and Memtis-QuickCool – with different data cooling speeds (i.e., the number of samples collected before halving a page's access count). Specifically, Memtis-Default used the default cooling period of 2,000k samples, while Memtis-QuickCool used a period of 2k samples. A shorter cooling period encourages more frequent page migration between the memory tiers.

Memtis relies on Intel's Processor Event-Based Sampling (PEBS) to track memory access patterns. It samples various hardware events, including LLC misses, TLB misses, and retired store instructions, to infer accessed page addresses and build frequency-based histograms to aid in making migration decisions. Memtis currently only supports Intel-based systems, though it can be ported to AMD processors with Instruction-based Sampling (IBS). Thus, Memtis was not evaluated on platform D. Memtis works slightly differently on CXL-memory systems (platforms A and B) and the PM system (platform C). LLC misses to CXL memory are regarded as *uncore* events on Intel platforms and thus cannot be captured by PEBS. Therefore, Memtis relies solely on TLB misses and retired store instructions to infer page temperature on platforms A and B.

4.1 Micro-benchmarks

To evaluate the performance of NOMAD's transactional page migration and shadowing mechanisms, we developed a micro-benchmark to precisely assess NOMAD in a controlled manner. This micro-benchmark involves 1) allocating data to specific segments of the tiered memory; 2) running tests with various working set sizes (WSS) and resident set sizes (RSS); 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, as illustrated in Figure 6, to evaluate tiered memory management under different memory pressures. As platform B behaved similarly to platform A in micro-benchmarks, it is excluded from the discussion.

Small WSS. We began with a scenario with a small WSS of 10 GB and a total RSS of 20 GB. Initially, we filled the first 10 GB of local DRAM with the first half of the RSS data. Subsequently, we allocated 10 GB of WSS data as the



(a) Small working set size (b) Medium working set size (c) Large working set size

Figure 7: Performance comparison between TPP, Memtis-Default, Memtis-QuickCool, and NOMAD on platform A.

second half of the RSS – 6 GB on the local DRAM and 4 GB on CXL memory (platforms A and D) and the PM (platform C). The micro-benchmark continuously performed memory reads or writes (following a Zipfian distribution) to this 10 GB WSS data, spread across both the local DRAM and CXL memory or PM. The frequently accessed, or "hot" data, was uniformly distributed along the 10 GB WSS. In TPP and NOMAD, accessing data on CXL memory or PM triggered page migration to the local DRAM, with TPP performing this migration synchronously and NOMAD asynchronously. In contrast, Memtis used a background thread to migrate hot data from CXL memory or PM to the local DRAM. Due to page migration, the 4 GB WSS data, initially allocated to CXL memory or PM, was gradually moved to the local DRAM. Since the WSS was small (i.e., 10 GB), it could be completely stored in the fast tier (i.e., local DRAM) after the micro-benchmark reached a *stable* state.

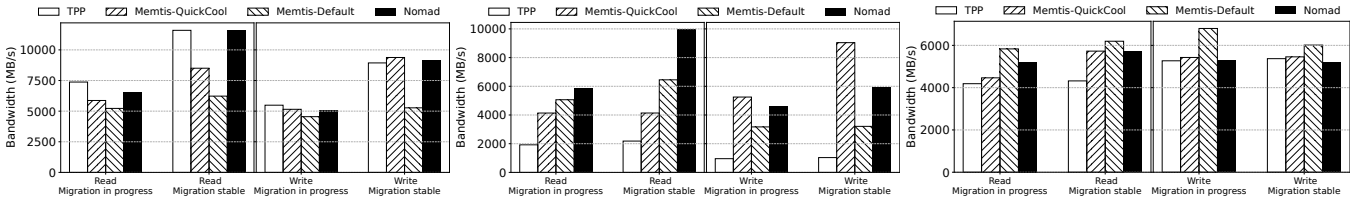
Figures 7 (a), 8 (a), and 9 (a) show that in the *transient* phase, during which page migration was conducted intensively (i.e., *migration in progress*), both NOMAD and Memtis demonstrated similar performance regarding memory bandwidth for reads. Although page-fault-based page migration in NOMAD could incur more overhead than the PEBS-based approach in Memtis, when the WSS can fit in fast memory and no memory thrashing occurs, the benefit of migration outweighs its overhead. For writes, e.g., on platform A, NOMAD incurred noticeable performance degradation compared to Memtis due to possibly aborted migrations and the maintenance of shadow pages. Note that NOMAD's overhead varies across platforms depending on the performance difference between fast and slow memory. In contrast, Nomad consistently outperformed TPP for both read and write, except for the slightly worse performance on platform C, highlighting the advantage of asynchronous page migration in NOMAD.

In the *stable* phase (i.e., *migration stable*), when most of the WSS data had been migrated from CXL memory or PM to the local DRAM, both NOMAD and TPP achieved similar read/write bandwidth. This was because memory accesses were primarily served by the local DRAM with few page migrations, as shown in Table 2. Memtis performed the worst, achieving as low as 40% of the performance of the other two approaches. We make two observations regarding Memtis's weaknesses. First, its stable phase performance is not drastically different from the transient phase. The migration statistics in Table 2 show that Memtis performed significantly fewer page migrations. This explains its sub-optimal performance in the stable phase as most memory accesses were still served from slow memory. Second, a shorter cooling period in Memtis, which incentivizes more frequent migrations, led to better performance. This also suggests that sampling-based memory access tracking may not accurately identify and timely migrate hot pages to fast memory.

Medium WSS. We increased the size of WSS and RSS to 13.5 GB and 27 GB, respectively. Similarly, we placed the first half of the RSS (13.5 GB) at the start of the local DRAM, followed by 2.5 GB of the WSS on the local DRAM, with the remaining 11 GB residing on CXL memory or PM. However, as the system (e.g., the OS kernel) required approximately 3-4 GB of memory, the WSS could barely fit in the fast tier, resulting in occasional and substantial migrations even during the stable phase. Accurately identifying hot pages and avoiding thrashing is crucial to achieving high performance for this medium-sized benchmark.

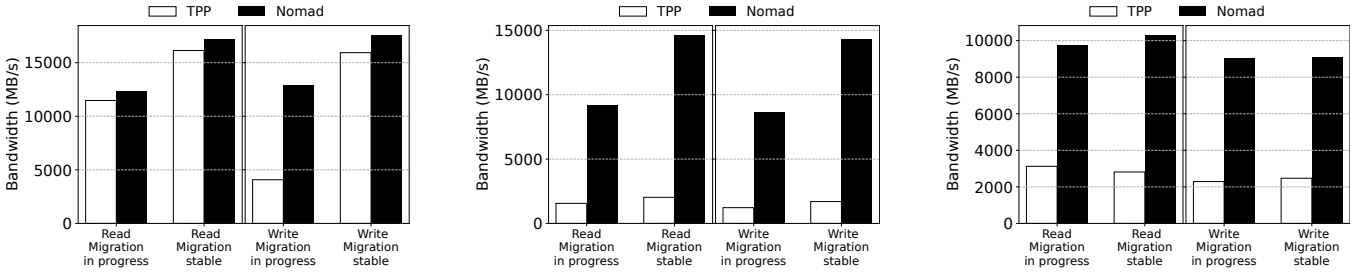
Unlike the small WSS case, Figures 7 (b), 8 (b), and 9 (b) show that during the transient phase, NOMAD and TPP generally achieved lower performance for both read and write compared to Memtis. This is because, under the medium WSS, the system experienced higher memory pressure than in the small WSS case, causing NOMAD and TPP to conduct more page migrations (2x - 6x) and incur higher overhead than Memtis, as shown in Table 2. Many of such migrations were futile during thrashing. Conversely, Memtis performed significantly fewer page migrations and avoided the waste. However, there was no evidence that Memtis effectively detected thrashing and throttled migration. The coarse-grained sampling was unable to accurately determine page temperature in a volatile situation and inadvertently sustained high performance under high memory pressure.

In the stable phase, NOMAD significantly outperformed TPP in all cases, especially on platform D. These results show the benefit of NOMAD's transactional page migration and non-exclusive memory tiering compared to TPP's synchronous page migration and exclusive tiering. On platform D, which was equipped with an application-specific integrated circuit (ASIC)-based CXL memory implementation, the performance gap between fast and slow memory narrows. Thus, the software overhead associated with synchronous page migration was exacerbated and NOMAD offered more pronounced per-



(a) Small working set size (b) Medium working set size. (c) Large working set size

Figure 8: Performance comparison between TPP, Memtis-Default, Memtis-QuickCool, and NOMAD on platform C.



(a) Small working set size (b) Medium working set size (c) Large working set size

Figure 9: Performance comparison between TPP and NOMAD on platform D with AMD Genoa processor. Memtis does not support AMD's instruction-based sampling (IBS) and thus was not tested.

formance gains. Similarly, NOMAD achieved substantially higher performance in reads than Memtis and comparable performance in writes. Unlike in the small WSS case, in which asynchronous and transactional page migration in NOMAD contributed most to its performance benefit, the advantage of page shadowing played a critical role in alleviating thrashing in the medium WSS case. Under memory thrashing, most demoted pages, which were recently promoted from the slow tier, can be simply discarded without migration. However, for write-intensive workloads, page shadowing requires one additional page fault for each write to restore a page's original read-write permission. This explains NOMAD's inferior write performance in the stable phase compared to Memtis.

Large WSS. We scaled up the WSS and RSS both to 27 GB and fully populated local DRAM with the first 16 GB of the WSS. The remaining WSS spilled onto CXL memory or PM. Unlike the medium WSS that incurred intermittent memory thrashing, this workload caused continuous and severe thrashing as the size of hot data greatly exceeded the capacity of fast memory. Figures 7 (c), 8 (c), and 9 (c) present the performance results in both the *transient* phase and the *stable* phase. Compared to the tests with the medium-sized workload in which NOMAD could outperform Memtis for read-only benchmarks, especially in the stable phase, both NOMAD and TPP performed worse than Memtis in almost all scenarios. It suggests that page fault-based tiered memory management, which makes per-page migration decisions upon access to a page, inevitably incurs high overhead during severe memory thrashing. Nevertheless, NOMAD consistently and significantly outperformed TPP thanks to asynchronous, transactional page migration and page shadowing.

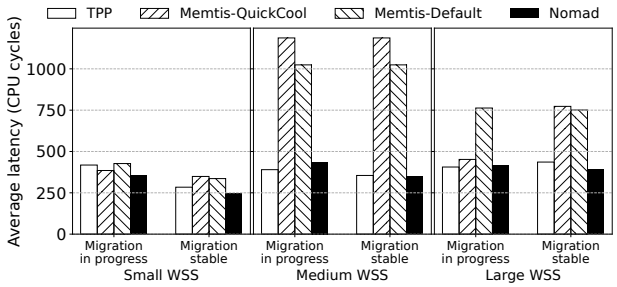


Figure 10: The average cache line access latency on platform C. The benchmark is a point-chasing workload optimized for PEBS-based memory access tracking.

Limitations of PEBS-based approaches. Our evaluation revealed several issues with PEBS-based memory access tracking. While Memtis prevented excessive migrations during thrashing, it achieved sub-optimal performance and failed to migrate all hot data to fast memory even when the WSS could fit in the fast tier. Due to the lack of hardware support for hot page tracking, PEBS-based approaches employ indirect metrics, such as LLC and TLB misses to sample recently accessed addresses to infer page temperature. Sampling-based memory tracking has two fundamental limitations. First, there is a difficult tradeoff between sampling rate and tracking accuracy. Second and most importantly, cache misses may not effectively capture hot pages. For most frequently accessed pages that always hit the caches, Memtis fails to collect enough (cache miss) samples to build the histogram. If such pages are evicted from the caches, e.g., due to conflict or coherence misses, they will be falsely regarded as "cold" pages.

To demonstrate these limitations, we created a favorable

RSS	23GB	25GB	27GB	29GB
Total shadow page size	3.93GB	2.68GB	2.2GB	0.58GB

Table 3: Shadow memory size as RSS changes on platform B. The size of tiered memory (DRAM+CXL) is 30.7 GB.

scenario where Memtis can capture every page access. We used a pointer-chasing benchmark that repeatedly accesses multiple fixed-sized (1 GB) memory blocks. Within each 1 GB block, the benchmark randomly accesses all cache lines belonging to a block while accesses across blocks follow a Zipfian distribution. The number of blocks determines the WSS. Since the block size exceeds the LLC size in our testbeds, every access generates an LLC miss that can be captured by Memtis. Effective memory access tracking should identify hot blocks and place them in fast memory.

Figure 10 shows the average latency to access a cache line in this benchmark on platform C. Note that platform C with PM was the only testbed on which Memtis has full tracking capability and can capture all the PEBS events. According to Table 1, a latency closer to DRAM performance (∼250 cycles) indicates more effective page placement. As shown in Figure 10, when the WSS exceeds fast tier capacity, Memtis achieved latency close to slow memory performance, suggesting that most hot pages still resided in the slow tier. In comparison, page fault-based approaches, e.g., NOMAD and TPP, can timely migrate hot pages and achieve low latency.

Robustness. Page shadowing can potentially increase memory usage and in the worst case can cause OOM errors if shadow pages are not timely reclaimed. In this test, we evaluated NOMAD's shadow page reclamation. We measured the total memory usage and the size of shadow memory using a micro-benchmark that sequentially scans a predefined RSS area. Table 3 shows the change of shadow pages as we varied the RSS. The results suggest that NOMAD effectively reclaimed shadow pages to reduce shadow memory usage as RSS increased and approached memory capacity.

4.2 Real-world Applications

We continued the evaluation of NOMAD using three representative real-world applications with unique memory access patterns: Redis [10], PageRank [9], and Liblinear [5]. We ran these three applications on four platforms (as shown in Table 1) with two configurations: 1) a small RSS (under 32 GB) working with all platforms and 2) a large RSS (over 32 GB) only on platform C and D with large PM or CXL memory. In addition, we include results from a "no migration" baseline which disables page migrations to show whether tiered memory management is necessary.

Key-value store. We first conducted experiments on a *latency-sensitive* key-value database, Redis [10]. The workload was generated from YCSB [11], using its *update-heavy* workload A, with a 50/50 distribution of read and write operations. We crafted three cases with different RSS and total operations.

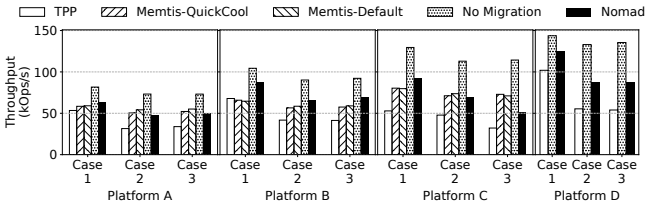


Figure 11: Performance comparisons using Redis and YCSB between TPP, Memtis-Default, Memtis-QuickCool, NOMAD, and "no migration".

Note that the parameters of YCSB were set as default unless otherwise specified. Case 1: We set *recordcount* to 6 million and *operationcount* to 8 million. After pre-loading the dataset, we used a customized tool to demote all memory pages to the slow tier before starting the experiment. The RSS of this case was 13GB. Case 2: We increased the RSS by setting *recordcount* to 10 million and *operationcount* to 12 million. We demoted all the memory pages to the slow tier in the same way. The RSS of this case was 24GB. Case 3: We kept the same total operations and RSS as Case 2. However, after pre-loading the dataset, we did *not* demote any memory pages.

Consistent with the micro-benchmarking results, Figure 11 shows that NOMAD delivered superior performance (in terms of operations per second) compared to TPP across all platforms in all cases. In addition, NOMAD outperformed Memtis when the WSS was small (i.e., in case 1), but suffered more performance degradation as the WSS increased (i.e., in case 2 and 3) due to an increased number of page migrations and additional overhead. Finally, all the page migration approaches underperformed compared to the "no migration" baseline. It is because the memory accesses generated by the YCSB workload were mostly "random", rendering migrating pages to the fast tier less effective, as those pages were unlikely to be accessed again. It indicates once again that page migration could incur nontrivial overhead, and a strategy to dynamically switch it on/off is needed.

We further increased the RSS of the database and operations of YCSB by setting the *recordcount* to 20 million and *operationcount* to 30 million. The RSS for this case was 36.5GB, exceeding the total size of the tiered memory on platforms A and B. Thus, the large RSS test was only performed on platforms C and D. We tested two initial memory placement strategies for the database – 1) *thrashing* that allocated all pages first to the slow tier and immediately invoked intensive page migrations, and 2) *normal* that prioritized page allocation to fast memory and triggered page migration only under memory pressure. As shown in Figure 14, NOMAD outperformed TPP due to its graceful performance degradation during thrashing but fell short of matching Memtis's performance. The initial placement strategy did not substantially affect the results and performance under different placements eventually converged.

Graph-based computation. We used PageRank [9], an ap-

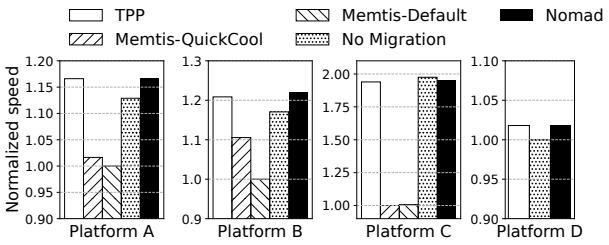


Figure 12: Performance comparisons of PageRank between non-migration, TPP, Memtis, and NOMAD. Performance is normalized to the approach with the lowest speed.

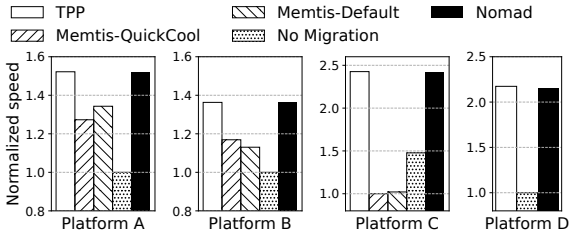


Figure 13: Performance comparisons of Liblinear between non-migration, TPP, Memtis, and NOMAD. Performance is normalized to the approach with the lowest speed.

plication used to rank web pages. It involves iterative computations to determine the rank of each page, based on the link structures of the entire web. As the size of the dataset increases, the computational complexity also increases, making it both memory-intensive and compute-intensive. We used a benchmark suite [8] to generate a synthetic uniform-random graph comprising 2^{26} vertices, each with an average of 20 edges. The RSS in this experiment was 22 GB, indicating that the memory pages were distributed at both the local DRAM and remote CXL memory or PM.

Figure 12 illustrates that there was negligible variance in performance between scenarios with page migrations (using NOMAD and TPP) and without page migrations (no migration). The results suggest that: 1) For *non-latency-sensitive* applications, such as PageRank, using CXL memory can significantly expand the local DRAM capacity without adversely impacting application-level performance. 2) In such scenarios, page migration appears to be unnecessary. These findings also reveal that the overhead associated with NOMAD's page migration minimally influences PageRank's performance. Additionally, it was observed that among all evaluated scenarios, Memtis exhibited the least efficient performance.

Figure 15 shows the case when we scaled the RSS to a very large scale on platforms C & D. When the PageRank program started, it first used up to 100GB memory, then its RSS size dropped to 45GB to 50GB. NOMAD achieved 2x the performance of TPP (both platforms) and slightly better than Memtis (platform C), due to more frequent page migrations – the local DRAM (16 GB) was not large enough to accommodate the WSS in this case.

Workload type	Success : Aborted
Liblinear (large RSS) on platform C	1:1.9
Liblinear (large RSS) on platform D	2.6:1
Redis (large RSS) on platform C	153:1
Redis (large RSS) on platform D	278.2:1

Table 4: The Success rate of transactional migration.

Machine learning. Our final evaluation of NOMAD involved using the machine learning library Liblinear [5], known for its large-scale linear classification capabilities. We executed Liblinear with an L1 regularized logistic regression workload with an RSS of 10 GB. Prior to each execution, we used our tool to demote all memory pages associated with the Liblinear workload to the slower memory tier.

Figure 13 demonstrates that both NOMAD and TPP significantly outperformed "no migration" and Memtis across all platforms, with performance improvement ranging from 20% to 150%. This result further illustrates that when the WSS is smaller than the local DRAM, NOMAD and TPP can substantially enhance application performance by timely migrating hot pages to the faster memory tier. Figure 16 shows that with a much larger model and RSS when running Liblinear, NOMAD consistently achieved high performance across all cases. In contrast, TPP's performance significantly declined, likely due to inefficiency issues, as frequent, high bursts in kernel CPU time were observed during TPP execution.

Migration success rate. As stated in Section 3.1, NOMAD's transactional page migration may be aborted due to updates to the migrating page, resulting in wasted memory bandwidth and CPU cycles. Subsequent retries could also fail. A low success rate could negatively affect application performance. Table 4 shows NOMAD's migration success rate for Liblinear and Redis on platforms C and D. We chose a large RSS for both applications and ensured there were sufficient cross-tier migrations. We observed a low success rate for Liblinear while Redis had a high success rate. Interestingly, this contrasted with NOMAD's performance – it was excellent with Liblinear but poor with Redis with large RSS. This suggests that a high success rate in page migrations does not necessarily lead to high performance. A low success rate indicates that the pages being migrated by NOMAD are also being modified by other processes, implying their "hotness". Timely migration of such pages can benefit ongoing and future accesses.

Summary. The results from micro-benchmarks and applications indicate that when the WSS was smaller than the performance tier, NOMAD enabled workloads to maintain higher performance than Memtis through asynchronous, transactional page migrations. However, when the WSS was comparable to or exceeded the performance tier capacity, leading to memory thrashing, the page-fault-based migration in NOMAD became detrimental to workload performance, underperforming Memtis in write operations. Notably, NOMAD's page shadowing feature preserved the efficiency of read opera-

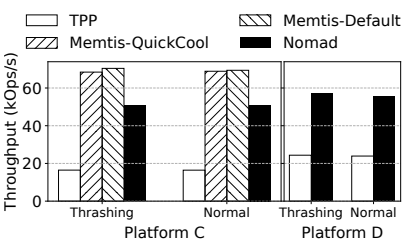


Figure 14: Redis (large RSS).

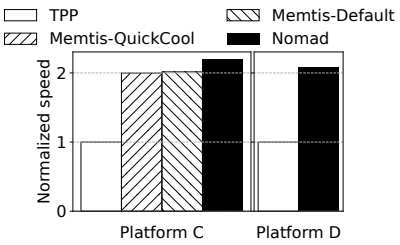


Figure 15: Page ranking (large RSS).

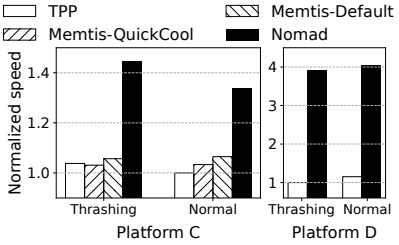


Figure 16: Liblinear (large RSS).

tions even under severe memory thrashing, often maintaining comparable or superior performance to Memtis. In all test scenarios, NOMAD significantly outperformed the state-of-the-art page-fault-based migration approach, TPP.

The evaluation results across four different platforms reveal the following observations: First, NOMAD generally performed better on platform D, which was equipped with faster and larger CXL memory, compared to other platforms. Additionally, the reduced performance gap between fast and slow memory on platform D allowed NOMAD to achieve greater performance gains than TPP, as the performance overhead from TPP's synchronous page migration was more pronounced. Second, while platforms A and B showed similar behavior in micro-benchmarks, their application-level performance varied (slightly) across different applications, suggesting that specific CPU features (differing between the off-the-shelf Intel Sapphire Rapids CPU for platform A and the engineering sample for platform B) may affect the performance of page migration under more realistic workloads.

5 Discussions and Future Work

The key insight from NOMAD's evaluation is that page migration, especially under memory pressure, has a detrimental impact on overall application performance. While NOMAD achieved graceful performance degradation and much higher performance than TPP, an approach based on synchronous page migration, its performance is sub-optimal compared to that without page migration. When the program's working set exceeds the capacity of the fast tier, the most effective strategy is to access pages directly from their initial placement, completely disabling page migration. It is straightforward to detect memory thrashing, e.g., frequent and equal number of page demotions and promotions, and disable page migrations. However, estimating the working set size to resume page migration becomes challenging, as the working set now spans multiple tiers. It requires global memory tracking, which could be prohibitively expensive, to identify the hot data set that can potentially be migrated to the fast tier. We plan to extend NOMAD to unilaterally throttle page promotions and monitor page demotions to effectively manage memory pressure on the fast tier. Note that this would require the development of a new page migration policy, which is orthogonal to the NOMAD page migration mechanisms proposed in this work.

Impact of Platform Characteristics: There exist difficult tradeoffs between page fault-based access tracking, such as TPP and NOMAD, and hardware performance counter-based memory access sampling like Memtis. While page fault-based tracking effectively captures access recency, it can be potentially expensive and on the critical path of program execution. In comparison, hardware-based access sampling is off the critical path and captures access frequency. However, it is not responsive to workload changes and its accuracy relies on the sampling rate. One advantage of NOMAD is that it is a page fault-based migration approach that is asynchronous and off the critical path. A potential future work is integrating NOMAD with hardware-based, access frequency tracking, such as Memtis, to enhance the current migration policy.

6 Conclusion

This paper introduces non-exclusive memory tiering as an alternative to the common exclusive memory tiering strategy, where each page is confined to either fast or slow memory. The proposed approach, implemented in NOMAD, leverages transactional page migration and page shadowing to enhance page management in Linux. Unlike traditional page migration, NOMAD ensures asynchronous migration and retains shadow copies of recently promoted pages. Through comprehensive evaluations, NOMAD demonstrates up to 6x performance improvement over existing methods, addressing critical performance degradation issues in exclusive memory tiering, especially under memory pressure. The paper calls for further research in tiered memory-aware memory allocation.

7 Acknowledgments

We thank our shepherd, Sudarsun Kannan, and the anonymous reviewers for their constructive feedback. This work was supported in part by NSF grants CCF-1845706, CNS-2415774, CCF-2415473, and the gift and equipment from Intel Labs and Micron Technology.

References

[1] https://www.computeexpresslink.org/.

[2] Autonuma: the other approach to numa scheduling. https://lwn.net/Articles/488709/.

[3] Damon-based reclamation. https://docs.kernel.org/admin-guide/mm/damon/reclaim.html#:~:text=DAMON%2Dbased%20Reclamation%20(DAMON_RECLAIM),of%20memory%20pressure%20and%20requirements.

[4] https://blocksandfiles.com/2023/11/20/accelerating-high-bandwidth-memory-to-light-speed/. https://blocksandfiles.com/2023/11/20/accelerating-high-bandwidth-memory-to-light-speed/.

[5] https://www.csie.ntu.edu.tw/ cjlin/libsvmtools/multicore-liblinear/.

[6] Intel agilex® 7 fpga and soc fpga. https://www.intel.com/content/www/us/en/products/details/fpga/agilex/7.html.

[7] Intel optane dimm. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[8] Pagerank. https://github.com/sbeamer/gapbs.

[9] Pagerank wiki. https://en.wikipedia.org/wiki/PageRank.

[10] Redis. https://redis.io/.

[11] Ycsb. https://github.com/brianfrankcooper/YCSB.

[12] Caching less for better performance: Balancing cache size and update cost of flash memory cache in hybrid storage systems. In *10th USENIX Conference on File and Storage Technologies (FAST 12)* (San Jose, CA, Feb. 2012), USENIX Association.

[13] ABULILA, A., MAILTHODY, V. S., QURESHI, Z., HUANG, J., KIM, N. S., XIONG, J., AND HWU, W.-M. Flatflash: Exploiting the byte-accessibility of ssds within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (2019), pp. 971–985.

[14] AGARWAL, N., AND WENISCH, T. F. Thermostat: Application-transparent page management for two-tiered main memory. *SIGPLAN Not. 52*, 4 (apr 2017), 631–644.

[15] AHMADIAN, S., SALKHORDEH, R., AND ASADI, H. Lbica: A load balancer for i/o cache architectures. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)* (2019), pp. 1196–1201.

[16] ALIAN, M., SHIN, J., KANG, K.-D., WANG, R., DAGLIS, A., KIM, D., AND KIM, N. S. Idio: Orchestrating inbound network data on server processors. *IEEE Computer Architecture Letters 20*, 1 (2021), 30–33.

[17] AMD. Zynq ultrascale+ device technical reference manual (ug1085), 2023. https://docs.amd.com/r/en-US/ug1085-zynq-ultrascale-trm/TLB-Maintenance-Operations.

[18] ARM. Learn the architecture - aarch64 memory management guide, 2024. https://developer.arm.com/documentation/101811/0103/Translation-Lookaside-Buffer-maintenance/Format-of-a-TLB-operation.

[19] BERGMAN, S., FALDU, P., GROT, B., VILANOVA, L., AND SILBERSTEIN, M. Reconsidering os memory optimizations in the presence of disaggregated memory. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management* (New York, NY, USA, 2022), ISMM 2022, Association for Computing Machinery, p. 1–14.

[20] BOCK, S., CHILDERS, B. R., MELHEM, R., AND MOSSÉ, D. Concurrent page migration for mobile systems with os-managed hybrid memory. In *Proceedings of the 11th ACM Conference on Computing Frontiers* (New York, NY, USA, 2014), CF '14, Association for Computing Machinery.

[21] BURNETT, N. C., BENT, J., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. Exploiting Gray-Box knowledge of Buffer-Cache management. In *2002 USENIX Annual Technical Conference (USENIX ATC 02)* (Monterey, CA, June 2002), USENIX Association.

[22] CHEN, F., KOUFATY, D. A., AND ZHANG, X. Hystor: Making the best use of solid state drives in high performance storage systems. In *Proceedings of the international conference on Supercomputing* (2011), pp. 22–32.

[23] CHOU, C., JALEEL, A., AND QURESHI, M. Batman: Techniques for maximizing system bandwidth of memory systems with stacked-dram. In *Proceedings of the International Symposium on Memory Systems* (New York, NY, USA, 2017), MEMSYS '17, Association for Computing Machinery, p. 268–280.

[24] DURAISAMY, P., XU, W., HARE, S., RAJWAR, R., CULLER, D., XU, Z., FAN, J., KENNELLY, C., MCCLOSKEY, B., MIJAILOVIC, D., MORRIS, B., MUKHERJEE, C., REN, J., THELEN, G., TURNER, P., VILLAVIEJA, C., RANGANATHAN, P., AND VAHDAT, A. 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* (New York, NY, USA, 2023), ASPLOS 2023, Association for Computing Machinery, p. 727–741.

[25] FORNEY, B. C., AND ARPACI-DUSSEAU, A. C. Storage-Aware caching: Revisiting caching for heterogeneous storage systems. In *Conference on File and Storage Technologies (FAST 02)* (Monterey, CA, Jan. 2002), USENIX Association.

[26] GUERRA, J., PUCHA, H., GLIDER, J., BELLUOMINI, W., AND RANGASWAMI, R. Cost effective storage using extent based dynamic tiering. In *9th USENIX Conference on File and Storage Technologies (FAST 11)* (San Jose, CA, Feb. 2011), USENIX Association.

[27] HOLLAND, D. A., ANGELINO, E., WALD, G., AND SELTZER, M. I. Flash caching on the storage client. In *2013 USENIX Annual Technical Conference (USENIX ATC 13)* (San Jose, CA, June 2013), USENIX Association, pp. 127–138.

[28] INTEL®. Remote action request –white paper. *revision 1.0* (2021). https://www.intel.com/content/dam/develop/external/us/en/documents/341431-remote-action-request-white-paper.pdf.

[29] JIANG, S., DING, X., CHEN, F., TAN, E., AND ZHANG, X. DULO: An effective buffer cache management scheme to exploit both temporal and spatial localities. In *4th USENIX Conference on File and Storage Technologies (FAST 05)* (San Francisco, CA, Dec. 2005), USENIX Association.

[30] JUN, H., CHO, J., LEE, K., SON, H.-Y., KIM, K., JIN, H., AND KIM, K. Hbm (high bandwidth memory) dram technology and architecture. In *2017 IEEE International Memory Workshop (IMW)* (2017), IEEE, pp. 1–4.

[31] JUNG, M. Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd). HotStorage '22, Association for Computing Machinery, p. 45–51.

[32] KIM, J., CHOE, W., AND AHN, J. Exploring the design space of page management for Multi-Tiered memory systems. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)* (July 2021), USENIX Association, pp. 715–728.

[33] KIM, Y., GUPTA, A., URGAONKAR, B., BERMAN, P., AND SIVASUBRAMANIAM, A. Hybridstore: A cost-efficient, high-performance storage system combining ssds and hdds. In *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems* (2011), pp. 227–236.

[34] KOLLER, R., MARMOL, L., RANGASWAMI, R., SUNDARARAMAN, S., TALAGALA, N., AND ZHAO, M. Write policies for host-side flash caches. In *11th USENIX Conference on File and Storage Technologies (FAST 13)* (San Jose, CA, Feb. 2013), USENIX Association, pp. 45–58.

[35] KOLLER, R., MASHTIZADEH, A. J., AND RANGASWAMI, R. Centaur: Host-side ssd caching for storage performance control. In *2015 IEEE International Conference on Autonomic Computing* (2015), pp. 51–60.

[36] KWON, Y., FINGLER, H., HUNT, T., PETER, S., WITCHEL, E., AND ANDERSON, T. Strata: A cross media file system. In *Proceedings of the 26th Symposium on Operating Systems Principles* (2017), pp. 460–477.

[37] LEE, T., MONGA, S. K., MIN, C., AND EOM, Y. I. Memtis: Efficient memory tiering with dynamic page classification and page size determination. In *Proceedings of the 29th Symposium on Operating Systems Principles* (New York, NY, USA, 2023), SOSP '23, Association for Computing Machinery, p. 17–34.

[38] LI, H., BERGER, D. S., HSU, L., ERNST, D., ZARDOSHTI, P., NOVAKOVIC, S., SHAH, M., RAJADNYA, S., LEE, S., AGARWAL, I., ET AL. 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* (2023), pp. 574–587.

[39] LIN, Z., XIANG, L., RAO, J., AND LU, H. P2CACHE: Exploring tiered memory for In-Kernel file systems caching. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)* (Boston, MA, July 2023), USENIX Association, pp. 801–815.

[40] LIN, Z., XIANG, L., RAO, J., AND LU, H. P2CACHE: Exploring tiered memory for In-Kernel file systems caching. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)* (Boston, MA, July 2023), USENIX Association, pp. 801–815.

[41] LIU, K., ZHANG, X., DAVIS, K., AND JIANG, S. Synergistic coupling of ssd and hard disk for qos-aware virtual memory. In *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)* (2013), pp. 24–33.

[42] LOH, G. H., JAYASENA, N., CHUNG, J., REINHARDT, S. K., O'CONNOR, J. M., AND MCGRATH, K. J. Challenges in heterogeneous die-stacked and off-chip memory systems.

[43] MARUF, A., GHOSH, A., BHIMANI, J., CAMPELLO, D., RUDOFF, A., AND RANGASWAMI, R. Multi-clock: Dynamic tiering for hybrid memory systems. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)* (Los Alamitos, CA, USA, apr 2022), IEEE Computer Society, pp. 925–937.

[44] MARUF, H. A., WANG, H., DHANOTIA, A., WEINER, J., AGARWAL, N., BHATTACHARYA, P., PETERSEN, C., CHOWDHURY, M., KANAUJIA, S., AND CHAUHAN, P. 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* (New York, NY, USA, 2023), ASPLOS 2023, Association for Computing Machinery, p. 742–755.

[45] MESWANI, M. R., BLAGODUROV, S., ROBERTS, D., SLICE, J., IGNATOWSKI, M., AND LOH, G. H. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)* (2015), pp. 126–136.

[46] PAPAGIANNIS, A., XANTHAKIS, G., SALOUSTROS, G., MARAZAKIS, M., AND BILAS, A. Optimizing memory-mapped {I/O} for fast storage devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (2020), pp. 813–827.

[47] RUAN, Z., SCHWARZKOPF, M., AGUILERA, M. K., AND BELAY, A. {AIFM}:{High-Performance},{Application-Integrated} far memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (2020), pp. 315–332.

[48] SIM, J., LOH, G. H., KIM, H., OCONNOR, M., AND THOTTETHODI, M. A mostly-clean dram cache for effective hit speculation and self-balancing dispatch. In *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture* (2012), pp. 247–257.

[49] SOUNDARARAJAN, G., PRABHAKARAN, V., BALAKRISHNAN, M., AND WOBBER, T. Extending ssd lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies* (USA, 2010), FAST'10, USENIX Association, p. 8.

[50] SUN, Y., YUAN, Y., YU, Z., KUPER, R., JEONG, I., WANG, R., AND KIM, N. S. Demystifying cxl memory with genuine cxl-ready systems and devices. *ArXiv abs/2303.15375* (2023).

[51] WU, K., GUO, Z., HU, G., TU, K., ALAGAPPAN, R., SEN, R., PARK, K., ARPACI-DUSSEAU, A. C., AND ARPACI-DUSSEAU, R. H. The storage hierarchy is not a hierarchy: Optimizing caching on modern storage devices with orthus. In *19th USENIX Conference on File and Storage Technologies (FAST 21)* (2021), pp. 307–323.

[52] WU, X., AND REDDY, A. N. Exploiting concurrency to improve latency and throughput in a hybrid storage system. In *2010 IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (2010), pp. 14–23.

[53] XIANG, L., ZHAO, X., RAO, J., JIANG, S., AND JIANG, H. Characterizing the performance of intel optane persistent memory: a close look at its on-dimm buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems* (2022), pp. 488–505.

[54] YAN, Z., LUSTIG, D., NELLANS, D., AND BHATTACHARJEE, A. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2019), ASPLOS '19, Association for Computing Machinery, p. 331–345.

[55] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies* (USA, 2020), FAST'20, USENIX Association, p. 169–182.

[56] YANG, J., KIM, J., HOSEINZADEH, M., IZRAELEVITZ, J., AND SWANSON, S. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)* (Santa Clara, CA, Feb. 2020), USENIX Association, pp. 169–182.

[57] YANG, Q., AND REN, J. I-cash: Intelligently coupled array of ssd and hdd. In *2011 IEEE 17th International Symposium on High Performance Computer Architecture* (2011), pp. 278–289.

A Artifact Appendix

Abstract

The artifact contains the source code of NOMAD, TPP, and Memtis for reproducing the results and graphs presented in the paper. The code works on platforms with Persistent Memory, Intel Agilex CXL memory, and/or Micron CXL memory. To facilitate the reproduction, we have provided a collection of scripts for compiling and installing these approaches, executing the experiments, collecting logs, and creating graphs. More details are available in the "README.md" file.

Scope

This artifact demonstrates NOMAD's strengths and weaknesses over TPP and Memtis across various scenarios and platforms, as elaborated in the Evaluation section.

It is open-source and can be used for further research, development, or other purposes by the community.

Contents

NOMAD, TPP and Memtis implementation. We provide two separate patches to enable NOMAD and TPP to work with the upstream kernel version v5.13-rc6. In particular, the TPP patch comes from the Linux community email discussions. Memtis, on the other hand, is directly incorporated from its original artifact, with a few minor bugs fixed.

Documentation The "Reproducing Paper Results" section of "README.md" provides a step-by-step guide for reproducing the results in the paper. This guide includes instructions for compiling the three implementations (i.e., NOMAD, TPP, and Memtis), running the experiments, and generating the graphs as presented in the paper.

Hosting

Artifact link: https://github.com/lingfenghsiang/Nomad
Artifact license: GNU GPL V3.0
Artifact version tag: v0.0

Requirements

To reproduce the results in the paper, the system under test requires one NUMA node with a CPU and another CPU-less NUMA node. If the system has more NUMA nodes, the operating system might encounter unexpected errors. Additionally, Memtis is only fully functional on platforms with Optane Persistent Memory. More details are included in the "Prerequisites" section of "README.md".