PET: Proactive Demotion for Efficient Tiered Memory Management

Wanju Doh*
Seoul National University
South Korea
wj.doh@scale.snu.ac.kr

Yaebin Moon*
Samsung Electronics
South Korea
yaebin.moon@samsung.com

Seoyoung Ko
Seoul National University
South Korea
seoyoung@scale.snu.ac.kr

Seunghwan Chung
Seoul National University
South Korea
chung9868@scale.snu.ac.kr

Kwanhee Kyung
Seoul National University
South Korea
kwanhee.kyung@scale.snu.ac.kr

Eojin Lee
Inha University
South Korea
ejlee@inha.ac.kr

Jung Ho Ahn
Seoul National University
South Korea
gajh@snu.ac.kr

Abstract

Tiered memory is a promising approach for increasing main-memory capacity at a lower cost by using DRAM as the upper tier (fast memory) and slower-but-cheap byte-addressable memory as the lower tier (slow memory). A **proactive demotion**, one of the ways to use tiered memory efficiently, demotes cold data to slow memory even when fast memory has sufficient free space. Prior works have utilized proactive demotion to reduce the high cost of main memory by reducing applications' resident set size in fast memory. Further, proactive demotion helps mitigate severe performance degradation caused by fast memory shortages when there is a spike in demand for hot data. Still, we observe that leveraging memory access locality within the allocation units of applications enables larger fast-memory savings with lower system overhead.

We propose a new proactive demotion scheme, PET, which performs **p**roactive demotion for **e**fficient **t**iered memory management. PET proposes extending the unit of demotion and promotion from the OS page, adopted by prior works, to **PET-block** (**P-block**), which reflects the unit in which applications allocate memory. We also provide the mechanisms that carefully select the demotion target P-block and swiftly promote the demoted P-block when the access pattern changes. The prototype of PET on Linux kernel v6.1.44 reduces 39.8% (up to 80.4%) of fast-memory usage with only a 1.7% performance drop on average of the evaluated workloads. Also, it mitigates 31% performance slowdown compared to the default Linux kernel when the system's memory usage is larger than fast-memory capacity, which outperforms state-of-the-art schemes for tiered memory management.

CCS Concepts: • **Computer systems organization → n-tier architectures**; • **Software and its engineering → Memory management**.

Keywords: Memory Tiering, Operating Systems, Memory Management

ACM Reference Format:
Wanju Doh, Yaebin Moon, Seoyoung Ko, Seunghwan Chung, Kwanhee Kyung, Eojin Lee, and Jung Ho Ahn. 2025. PET: Proactive Demotion for Efficient Tiered Memory Management. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689031.3717471

1 Introduction

The demand for memory capacity in data centers has grown exponentially, but the slowdown of DRAM technology scaling makes it challenging to meet this demand solely with DRAM directly attached to the CPU's memory controller. Emerging coherent interfaces [12, 18], non-volatile memory technologies [39], or a combination of both can alleviate this demand by providing byte-addressable memory with lower, yet comparable, performance to DRAM at a lower cost. A promising usage model of these byte-addressable memories is *tiered memory*, which expands main-memory capacity by incorporating slower, but more cost-effective, byte-addressable memory (slow memory) alongside faster, but more expensive, DRAM (fast memory).

*Both authors contributed equally to the paper.

EuroSys '25, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/2025/03
https://doi.org/10.1145/3689031.3717471

To effectively utilize tiered memory in real-world systems, a robust data management technique is essential. It must intelligently place data to leverage the expanded memory capacity while mitigating performance degradation due to slower memory access. Various approaches to tiered memory management have been proposed at the hardware level [32, 33, 37, 38, 57, 59, 66–69], the application level [10, 15, 48, 55, 62], and the OS level [2, 5, 9, 16, 20, 31, 35, 36, 40, 41, 52, 54, 56, 60, 64]. This paper focuses on OS-level management techniques, offering application transparency without requiring hardware modification. To minimize access to slow memory, most existing OS-level techniques [5, 20, 31, 36, 40, 52, 54, 56, 64] employ a "lazy demotion" strategy. They migrate infrequently accessed pages (cold pages) to slow memory only when the fast-memory capacity falls below a critically low threshold, such as the Linux kernel's watermark (0.1–1% of the fast-memory capacity). However, this approach can lead to insufficient fast-memory capacity, forcing the demotion of cold pages to occur before the allocation of frequently accessed data (hot data) [41, 60, 63]. This can also limit the fast-memory available to other applications running concurrently on the system.

Proactive demotion schemes [2, 16, 35, 36, 41, 60, 63] demote cold pages even when fast-memory capacity is not deficient. By reducing the applications' resident set size in fast memory (up to 25%), proactive demotion lowers the requirement for expensive fast-memory capacity, allowing the same amount of tasks to be performed with less fast-memory capacity [2, 16, 35, 60]. Moreover, it provides free fast-memory space to allocate/promote hot data, mitigating the performance drop due to the deficiency of fast-memory capacity [36, 41, 63]. However, the abundance of cold pages in data-center applications (30–40% on average [16, 41, 60]) implies the feasibility of reducing a larger fast-memory usage with low-performance overhead through a more aggressive demotion, which can be an effective way to utilize tiered memory.

For aggressive and proactive demotion, establishing a demotion criterion and a management unit is essential to effectively identify cold data that will not affect performance. Based on the analysis using hardware performance monitoring units, we discovered that benchmarks typically exhibit the memory access locality within units by which processes and memory allocators allocate memory (e.g., memory object allocated by `malloc()`). To further explore the memory allocation unit, we manually modified the application to perform aggressive demotion and promotion using this memory allocation unit. These optimized binaries save from 13.2% to 84.7% of fast-memory usage while incurring only a slight performance drop (0.76%–2.82%), which showed the possibility of the management using a *memory allocation unit*.

We propose a new proactive demotion scheme named PET (**P**roactive demotion for **E**fficient **T**iered memory management) utilizing the memory access locality within the memory allocation unit. We extract the *PET block* (*P-block*), which approximates the memory allocation unit from the data structure OSs use to manage virtual address space. Then, we extend the management unit from the OS page to P-block. Based on the memory access locality within P-blocks, PET tracks accesses using only sampled pages in P-blocks. Also, PET demotes the P-blocks not accessed through multiple steps, including P-block splitting and pre-demotion (canary-demotion) of a portion of P-block. This P-block granularity demotion significantly reduces fast-memory usage with a low-performance overhead. We also propose a feature to demote cold file pages so that PET achieves more reduction in fast-memory usage. Finally, PET provides P-block granularity promotion to mitigate performance degradation due to slow access to demoted P-blocks. Our promotion mechanism introduces a dynamically adjusted threshold to promote the demoted P-block before slow-memory access degrades performance exceeding the tolerable degree.

In this paper, we make the following key contributions:

- We observe that benchmarks typically exhibit memory access locality within the memory allocation unit, and the optimized binaries with manually modified benchmarks that perform proactive demotion/promotion in this unit reduce large fast-memory usage with low-performance overhead (§3).
- We propose PET, a proactive demotion scheme capturing P-blocks that approximate the memory allocation unit (§4.1) and effectively identifies the data to be demoted in the P-block granularity (§4.2).
- We also propose a promotion mechanism that tracks accesses to demoted P-blocks and promotes them before they harm system performance (§4.3).
- PET reduces 39.8% (up to 80.4%) of fast-memory usage with only a 1.7% performance drop when the aggregate working set fits in fast memory. PET also reduces slowdown by 31% compared to the default Linux kernel when memory usage is larger than the fast-memory capacity (§5).

2 Background and Motivation

2.1 OS-level Management for Tiered Memory

OS-level management schemes for tiered memory utilize the page migration feature [49] to move hot data to the fast memory (promotion) and cold data to the slow memory (demotion) at runtime. They employ various methods to track the memory access of the system and determine hot/cold data to promote/demote.

The Least-Recently Used (LRU) list of OSs, originally implemented for page reclamation, is a popular method for access tracking. It ranks the pages in a system according to the recency of access to each page and classifies them into active/inactive lists. Prior works [20, 25, 40, 41, 60, 64] utilize the LRU list to identify the most suitable pages to be demoted under memory pressure.

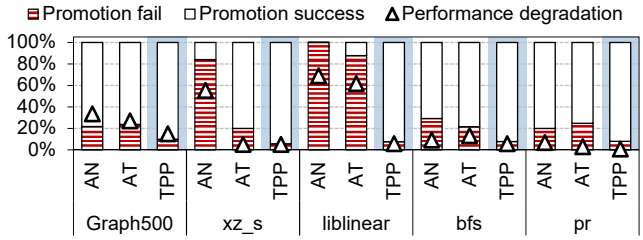

Figure 1. Promotion success ratio and performance drop (lower is better) compared to the system having sufficient fast memory capacity. AutoNUMA-Tiering (AN) [65] and AutoTiering (AT) [31] are two representative schemes demoting with a tight threshold, and TPP [41] is a representative scheme that performs proactive demotion.

Instead of using the LRU list, several works determine hot/cold pages based on their own policies. One example is periodically checking and clearing each page's access bit in the page table entry (PTE) [5, 9, 16, 35, 43, 54]. Others utilize a fake page fault by poisoning the PTE and track a page where the page fault occurs [2, 25, 31, 41, 63]. This method can perform a process for migration immediately after the page is accessed using the page fault handler. The rest utilize the performance monitoring unit (PMU) provided by processors [16, 36, 52, 56, 63]. They collect PMU profile data and map it to each page using user processes. Because each methodology has its own advantages, it is essential to employ the most suitable approach for one's mechanism.

2.2 Demotion with a Tight Threshold

Based on access tracking information, OS-level schemes need to accurately identify the hotness of data and appropriately trigger promotion or demotion. In this section, we classify existing works according to *how they trigger demotion.*

To minimize slow-memory access, most works [5, 20, 26, 31, 40, 52, 54, 56] trigger demotion using tight thresholds such as the Linux watermark, a criterion that restricts page allocation/promotion or makes promotion accompanied by demotion. Because the default value of the watermark is 10, which means the number of free pages in fast memory is lower than 0.1% of total pages, the demotion is only performed when free fast-memory capacity is deficient. When the aggregate working set fits in fast memory, these schemes show optimal performance because they use fast memory only. However, when the total working set size exceeds the fast-memory capacity, the lack of fast memory makes fast-memory allocation/promotion accompanied by demotion, which harms performance.

AutoNUMA-Tiering (AN) [65] and AutoTiering (AT) [31], two representative schemes triggering demotion with the Linux watermark, exhibit these characteristics. Figure 1 shows the ratio of promotion failure due to the deficiency of free fast memory and performance degradation of AN and AT where a working set size is 1.5 times the fast-memory capacity; the detailed experimental setup is explained in §5.1. AN and AT show a promotion failure ratio of 52.9% and 37.4% and a significant slowdown of 19.9% and 11.1% on average, respectively, compared to the system with a sufficient amount of fast memory capacity. liblinear [17], which fails most of the promotion, shows a large slowdown, whereas pr [4] with a small promotion failure ratio shows a small slowdown. This result shows that demotion with tight thresholds can constrain promotion and fast-memory allocation when the system's memory usage exceeds fast-memory capacity.

2.3 Proactive Demotion

Based on the abundance of cold pages in data-center applications (30–40% of the entire data) [16, 60], several works have suggested proactively demoting (or reclaiming) cold data even when fast memory is not deficient.

Far Memory [35] and TMO [60] propose to *reduce expensive main-memory costs* by reclaiming cold pages (up to 20%) to swap devices regardless of fast-memory usage. TMTS [16] and Thermostat [2] also focus on reducing memory costs through proactive demotion, but specifically target tiered memory systems. TMTS targets replacing 25% of DRAM with slower memory while keeping performance degradation below 5%. It achieves this by scanning the access bits of all pages every 30 seconds to identify the least frequently used (LFU) pages for demotion, maintaining a cold-age histogram to track pages not accessed within intervals.

Thermostat takes a different approach by tracking access counts for each huge page. To reduce overhead, it samples 5% of 2MB huge pages and re-samples 4KB pages (50 pages) within each. By inducing fake page faults through poisoning page table entries (PTEs) of sampled pages, it estimates slow-memory access counts during the sampling interval. Thermostat then compares the estimate with the pre-calculated threshold based on a user-defined slowdown (3%) and performs migration to satisfy the target slow-memory access count. While Thermostat achieves significant fast-memory savings, it can suffer from substantial performance degradation because of overusing fake page faults.

TPP [41], MEMTIS [36], and FlexMem [63] secure a small fraction of fast memory available for promotion and allocation *to overcome the limitations of the schemes that perform demotion with a tight threshold*, as shown in §2.2. TPP presents a new watermark to trigger demotion (`demotion_watermark`, 2% of fast-memory capacity), proactively demoting cold pages before reaching critical memory thresholds. As shown in Figure 1, TPP results in only 7.7% promotion failures and an average slowdown of 4.7%.

MEMTIS utilizes hardware-based sampling to manage page placement and size in tiered memory systems with Transparent Huge Pages enabled. It tracks the moving average of access counts for each page and maintains an access histogram of all pages to prioritize keeping frequently

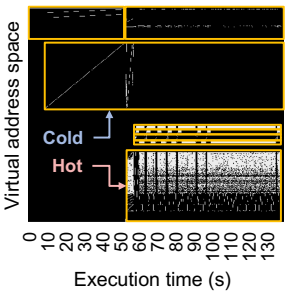

Figure 2. An analysis of memory access pattern and memory objects' region/lifetime.

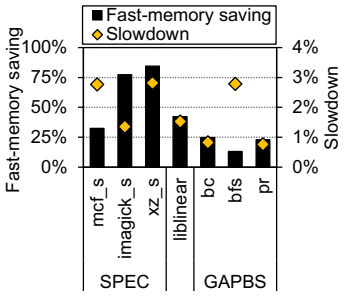

Figure 3. Fast-memory saving and slowdown of the optimized binaries compared to the original binaries.

accessed (hotter) pages in faster memory. Similar to TPP, MEMTIS triggers demotion when the amount of free space in fast memory drops below a predefined threshold (2%). FlexMem extends MEMTIS by adding fake page fault to access tracking to improve profiling accuracy and by deciding the number of pages to demote in an adaptive manner, which effectively reduces the promotion failure. However, because TPP, MEMTIS, and FlexMem proactively demote just to maintain a small fraction of free space in fast memory for performance improvement, they don't fully capitalize on the potential savings.

Prior works on proactive demotion focus on either reducing the requirement of fast-memory capacity or preventing performance drop due to the deficiency of fast-memory capacity. However, the presence of abundant cold pages in data-center applications implies that there is an opportunity to take full advantage of proactive demotion by demoting more cold pages with low-performance overhead. To enable more aggressive and proactive demotion, the access tracking method that tracks how long each data has not been accessed regardless of fast-memory usage is essential.

3 Exploring Aggressive and Proactive Demotion

In this section, we investigate new management units that are suitable for performing proactive demotion with minimal performance impact. Our analysis utilizing PMU reveals that benchmarks generally exhibit memory access locality within a memory allocation unit (e.g., memory object allocated using `malloc()`). Leveraging this, we manually modified the source code of several benchmarks for proactive demotion and promotion at *the memory object granularity*. These binaries reduced fast-memory usage significantly while maintaining low performance degradation, demonstrating the feasibility of hot/cold data classification based on memory allocation units.

3.1 Benchmark Analysis using PMU

Modern processors are equipped with a PMU, a specialized hardware unit that analyzes the system's performance characteristics. PMUs count the frequency of pre-defined hardware events (e.g., TLB misses, cache misses, and instruction cycles) using the sampling-based method at every sampling period defined by a user. Moreover, the PMUs of modern processors provide data address profiling feature [3, 22, 24] that aims to profile memory access. Data address profiling collects the runtime information of sampled load/store instructions, including hardware events caused by the instructions and the virtual address they accessed. These features are available through interfaces such as Linux Perf [14] and Intel VTune [23].

We analyzed the memory access characteristics of popular memory-intensive benchmarks (mcf_s, imagick_s, and xz_s in SPECspeed 2017, bfs, pr, and bc in GAP Benchmark Suite, and liblinear) in the system equipped with an Intel Xeon Platinum 8260 processor. We utilized Processor Event-Based Sampling (PEBS) and PEBS Data Linear Address (PEBS-DLA), the PMU and the data address profiling feature provided by the Intel processors, respectively. For each benchmark, we collected hardware events related to last-level cache (LLC) and second-level translation lookaside buffer (L2 TLB) misses (`MEM_LOAD_RETIRED.L3_MISS_PS` and `MEM_INST_RETIRED.STLB_MISS_LOAD|STORE_PS`). Then, we mapped each with the virtual address and time information of the load/store instruction.

Figure 2 shows the virtual address space where LLC and L2 TLB misses occurred over time during the execution of liblinear. The yellow boxes represent the regions and lifetimes of primary memory objects allocated using `malloc()`. The observed memory access pattern exhibits a high degree of spatial locality within individual memory objects (i.e., being hot or cold). Besides liblinear, other benchmarks under analysis also exhibited similar memory access behavior at the granularity of memory objects. These findings suggest that the unit for distinguishing hot and cold data can be extended beyond the OS-managed page level to memory allocation units, such as memory objects. Moreover, memory allocation units can be used as a criterion for determining the data size to demote proactively.

3.2 Proactive Demotion with Manually Modified Binaries

To verify the possibility of using the memory object as a unit for proactive demotion at an OS level, we performed an experiment using optimized binaries obtained by manually modifying the source code of each benchmark. We analyzed the source code and execution flow to identify the code line where primary memory objects, such as those shown in Figure 2, are allocated and accessed. Based on the source code analysis, we built an optimized binary that demotes memory

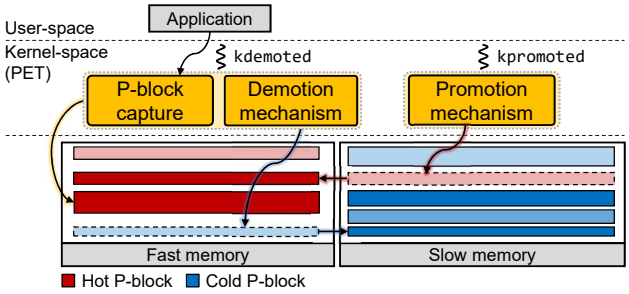

Figure 4. Overall process of PET with three mechanisms.

objects when each of them starts not to be accessed and promotes it right before it starts to be re-accessed, using the system call that performs page migration (e.g., `move_pages()`). We configured the optimized binary to have a working set that fits in fast memory. We compared the performance (inverse of execution time) and fast-memory usage of the optimized binaries with those of the original ones (i.e., binaries which allocate all data to fast memory without migration) in a tiered memory system described in §5.1.

As shown in Figure 3, optimized binaries reduce large amounts of fast memory usage (13.2–84.7%) with only slight performance degradation (0.76%–2.82%). This result indicates that performing proactive demotion/promotion in a memory object granularity reduces large fast-memory usage by enabling aggressive demotion with extended migration granularity compared to the OS page used by prior works. Identifying this memory allocation unit at the OS level and using it to perform proactive demotion/promotion can achieve comparable effects to optimized binaries without modifying source code, enabling an effective proactive demotion scheme. The memory allocation unit has been explored in various studies [10, 15, 19, 44, 48, 55, 61, 62], *but not considered in OS-level schemes for tiered memory yet.*

4 PET: Aggressive Demotion and Promotion

We propose a new OS-level proactive demotion scheme named PET, **P**roactive demotion for **E**fficient **T**iered memory management. To perform aggressive and proactive demotion, PET introduces a *PET block* (*P-block*) as a new unit for tiered memory management, utilizing memory access locality within the memory allocation unit at the OS level. PET's P-block-based mechanism effectively identifies target data to demote proactively compared to the page granularity demotion in previous studies. At the same time, PET achieves a larger reduction in fast-memory usage with less performance overhead by virtue of a multi-step demotion mechanism.

PET consists of three mechanisms (see Figure 4). First, the **P-block capturing mechanism** extracts the newly proposed P-block from the data structure that OSs use to manage virtual address space (§4.1). Second, the **demotion mechanism** performs proactive demotion through multiple steps

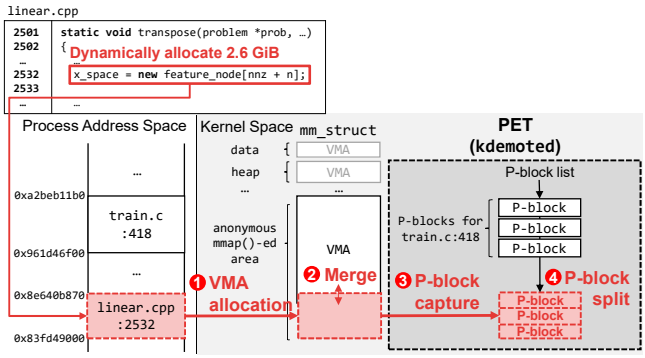

Figure 5. An example of P-block generation when liblinear dynamically allocates memory using `malloc()`.

based on P-block information to reduce fast-memory usage with low-performance overhead (§4.2). Finally, the **promotion mechanism** tracks access to demoted P-blocks and performs promotion after the number of accesses to slow memory exceeds a predetermined threshold to mitigate the performance drop caused by slow memory accesses (§4.3). We also propose the file page demotion feature in addition to the default PET (§4.4).

4.1 P-block: Management Unit for PET

Due to the difficulty of directly identifying the memory allocation units at the OS level, we approximate these units by utilizing the virtual address management unit of OSs. Most OSs manage each process's virtual address space by dividing it into several contiguous, non-overlapping spaces. Each contiguous virtual address space has its own properties (e.g., access permission), pointing to a specific area (e.g., heap and stack) of the process. As these spaces are created when applications or memory allocators allocate memory, they can be used to capture the units in which applications allocate memory. Various OSs such as Linux, FreeBSD, and Solaris have this unit, and in the rest of this paper, we will refer to it as VMA based on the `vm_area_struct` [7] of Linux.

Among the entire VMAs, PET focuses on the anonymous mmap()-ed area (anon_mmap), which is generated when applications or memory allocators allocate memory larger than the `MMAP_THRESHOLD` (128KB by default). We confirm that anon_mmap dominates the VMAs (more than 99%) in most applications, including memory objects explored in §3, and anon_mmap that contains larger memory allocations is suitable for exploiting memory access locality in memory allocation units. Therefore, PET captures anon_mmap VMAs created corresponding to each memory allocation and utilizes it as a data management unit for tiered memory. Figure 5 shows the P-block generation process when liblinear allocates a memory object. In response to the virtual memory allocation (`0x83fd49000` to `0x8e640b870`), the OS generates a new VMA for the requested memory object based on the page-aligned address (Figure 5-❶). After that, it merges the

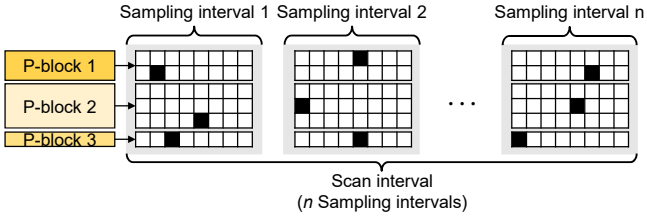

Figure 6. PET's sampling-based tracking to determine the demotion target. Within each scan interval, there are *n* sampling intervals that track accesses to sampled P-block pages.

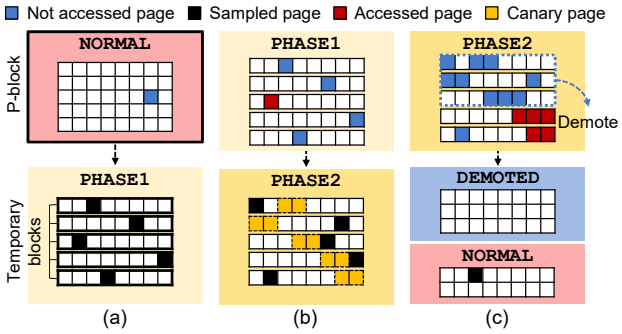

Figure 7. The operation of PET's demotion mechanism when the P-block of (a) NORMAL, (b) PHASE1, and (c) PHASE2 state is determined to be cold in each scan interval.

generated VMA with the neighboring VMA to minimize the number of VMAs (Figure 5-❷). PET captures the unit of the generated VMA into an additional structure before merging. We refer to this additional structure as *P-block* and utilize it as a data management unit for tiered memory (Figure 5-❸). We confirmed that P-blocks and memory objects allocated by `malloc()` overlap by more than 97% on the evaluated workloads. P-block information is periodically updated by a kernel thread named `kdemoted`, detailed in §4.2.

To prevent losing accuracy, PET performs initial splitting for the captured P-block when it is too large (e.g., GB scale) (Figure 5-❹). Large memory objects, such as the memory object shown in Figure 5, can include both hot/cold portions. Managing this memory object in large granularity reduces the accuracy of hot/cold data classification, which harms performance. Therefore, PET splits newly generated large P-blocks into specific sizes. We conducted a sensitivity study of the initial P-block splitting size, which is described in §5.3, and set the maximum P-block size to 1GB.

Each P-block requires 128 bytes of metadata (e.g., access counts and state), but PET shows negligible space overhead as the number of P-blocks is small. Graph500, which has the largest number of P-blocks per working-set size among the benchmarks used, allocates up to 217 P-blocks for 30GB of working set, showing only 0.0001% of space overhead.

4.2 Demotion Mechanism

Based on the P-block, PET performs demotion regardless of the total fast-memory usage, reducing fast-memory capacity requirement with a small performance impact. `kdemoted`, a kernel thread for performing the demotion mechanism, periodically updates captured P-block information and tracks memory access in P-block granularity rather than page granularity to identify demotion targets with only a small number of components. However, this sampling-based approach can often miss accesses occurring on pages that are not sampled, potentially leading to inaccurate identification of cold P-blocks. Therefore, PET performs a fine-grained multi-phase decision process for the P-blocks that are not accessed to identify the demotion target accurately.

To identify demotion targets, `kdemoted` utilizes a sampling-based method (see Figure 6). It tracks only one page per P-block by exploiting memory access locality within the P-block and identifies whether the access occurred in the P-blocks. `kdemoted` wakes up periodically using two different intervals, sampling/scan interval. For each sampling interval (e.g., hundreds of ms), it randomly samples one page per P-block and clears the sample page's PTE access bit, which is set if an access to that page occurs. After a sampling interval is passed, PET checks whether the access bit is set during the previous sampling interval, to verify access to P-blocks. This sampling-based method minimizes page table walk of access tracking using the PTE access bit. For each scan interval (e.g., several tens of times the sampling interval), `kdemoted` gathers the tracking information and identifies the demotion target P-blocks. Through memory access tracking, `kdemoted` considers the P-block with no access detected during the scan interval as ***cold***.

PET categorizes each P-block into one of four states based on its coldness: `NORMAL`, `PHASE1`, `PHASE2`, and `DEMOTED`. PET initially allocates P-blocks in fast memory and sets them to the `NORMAL` state (Figure 7). At each scan interval, it updates each P-block's state according to its coldness. If a P-block in the `NORMAL` state is determined to be cold by the access tracking mechanism, `kdemoted` switches the P-block to the `PHASE1` state and starts access tracking in a more fine-grained manner. Because P-blocks are typically larger than the size of an OS page, some P-blocks can contain multiple hot and cold blocks. Thus, determining the demotion target by tracking only sampled pages within a P-block can cause misidentification of the demotion target. To address this, PET splits the P-block into several temporary blocks when the P-block transitions to the `PHASE1` state (Figure 7-(a) and Figure 8-❶). The access tracking mechanism samples pages from each temporary block and determines coldness at the temporary block granularity. By applying detailed tracking of temporary blocks, PET enables finer-grained monitoring of memory access patterns and more accurate identification of the demotion target.

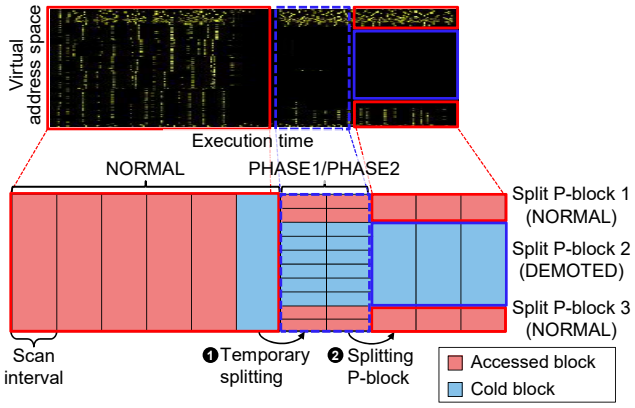

Figure 8. Example of a P-block in liblinear containing multiple hot/cold blocks.

For a P-block in the PHASE1 state, PET compares the total size of temporary blocks that have not been accessed (cold blocks) and those that have been accessed (accessed blocks) during the scan interval. If the total size of cold blocks is larger than that of accessed blocks, PET switches the P-block to the PHASE2 state, which is our final step before demotion.

For a P-block in the PHASE2 state, PET randomly selects and demotes a portion of the temporary blocks, named canary pages, to further increase access tracking accuracy. We set the protection flag of the canary pages to PROT_NONE so that the first access to them causes a fake page fault. When handling the fake page fault, PET increments the access counter of the temporary block to which the faulted page belongs. Because page fault handling is performed in the critical path of memory access, this fake page fault-based access tracking can cause high overhead. Thus, we fully utilize the canary pages in both determining the demotion target P-blocks in the PHASE2 state and promotion target P-blocks (we will explain the promotion mechanism in §4.3).

We also carefully select the ratio of the canary pages to accurately assess both coldness and hotness of P-blocks. A small ratio of canary pages can lower the demotion/promotion accuracy, while the bigger the ratio of canary pages, the larger the unnecessary demotion overhead can happen. We set the ratio of the canary pages to 10% of the temporary block to balance the accuracy and the overhead.

Finally, if the total size of cold blocks is greater than that of the accessed blocks within a P-block in the PHASE2 state, PET considers the P-block as a demotion target. For a P-block that exhibits different access patterns within it, PET splits the P-block based on the coldness of the temporary blocks. During this process, PET merges adjacent temporary blocks with the same property (i.e., accessed or cold) into one P-block (Figure 7-(c) and Figure 8-❷). As a result, PET switches the accessed P-blocks to the NORMAL state and promotes the canary pages within these P-blocks. For the P-blocks where cold blocks are merged, PET demotes them and converts them into the DEMOTED state, which prevents the lack of fast-memory capacity in a proactive manner.

PET's demotion mechanism involves a total of five user-configurable parameters (sampling/scan intervals, maximum P-block size, temporary block size, and canary page ratio). The default values for these parameters were chosen based on the settings that generally showed the highest performance in sensitivity studies (§5.3).

4.3 Promotion Mechanism

PET also performs promotion to swiftly detect and respond to changes in an application's memory access patterns, thereby reducing performance-harmful slow-memory accesses. Due to the locality of memory accesses within a P-block, accessing one page often leads to accessing others in the same P-block. Thus, PET identifies promotion targets at the P-block granularity.

Although previous studies have proposed effective promotion mechanisms, directly adopting their page-based methods are unsuitable for our P-block-based management. For example, promoting a P-block after only a few page accesses (e.g., 1–2 page faults [41] or 8 loads [52]) can overestimate the P-block's hotness, leading to unnecessary promotions and demotions. Scaling the promotion threshold to the P-block size can improve accuracy but result in excessive page faults and delayed promotion. Therefore, we propose a promotion mechanism tailored to our P-block-granularity approach.

We utilize fake page faults *only on canary pages*, enabling prompt detection of access pattern changes while reducing page fault handling costs. Further, we separate the decision for promotion from the actual promotion process to avoid performance penalties associated with promoting entire P-blocks during fault handling. We implemented kpromoted, a kernel thread that performs the actual promotion. When a fake page fault occurs, the page fault handler increments the access count of the P-block containing the faulted page. If the P-block meets the promotion criteria, the fault handler requests kpromoted to perform promotion.

For accurate promotion target selection, the first fake page fault on a demoted P-block does not immediately trigger promotion. PET allows continued slow-memory access to demoted P-blocks *until a target slow-memory access rate threshold is reached*, ensuring that P-blocks that become cold again after only a few accesses are not prematurely promoted. By avoiding premature promotion of large P-blocks, we prevent inefficient use of both fast- and slow-memory bandwidth and the dissipation of fast-memory capacity.

PET's promotion mechanism sets the system-wide target slow memory access rate, R_{promo}. It presents the allowed number of fake page faults per second and is calculated based on the following equation from Thermostat [2]. The equation calculates R_{promo} that satisfies the user-defined tolerable slowdown based on the mis-demotion penalty, which is an additional latency caused by a single fake page fault.

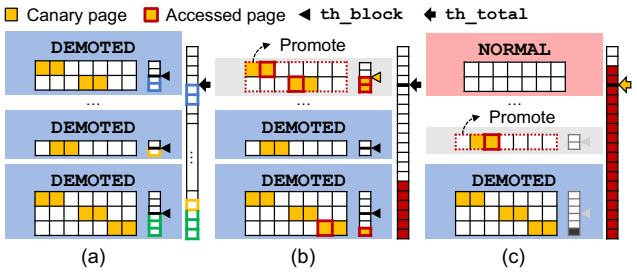

Figure 9. PET's promotion mechanism. (a) `th_total` is distributed to each demoted P-block in proportion to its size. (b) PET promotes the P-block if the number of fake page faults on it reaches `th_block` during a sampling interval. (c) After the aggregated number of fake page faults reaches `th_total`, the newly accessed P-block is promoted immediately, regardless of the `th_block`.

$$R_{promo} = \text{tolerable slowdown} \times \frac{\text{canary ratio}}{\text{mis-demotion penalty}}$$

We set the tolerable slowdown to 3% used by Thermostat. For the mis-demotion penalty, we conservatively set the worst-case additional latency caused by one hint fault, instead of the slow memory access latency (1μs) used by Thermostat. Through a microbenchmark that sequentially accesses a demoted unit at cache-line granularity prior to promotion, we measured an average additional latency of 8.3μs per page, reflecting overheads from slow-memory accesses and promotion. Using this mis-demotion penalty and 10% of canary ratio, we calculate R_{promo} as 360 faults/sec.

By multiplying R_{promo} with the sampling interval, we can get `th_total`, the target slow-memory access count for each sampling interval. PET also calculates `th_block`, the slow-memory access count threshold for each demoted P-block, by dividing `th_total` proportionally to each demoted P-block size (Figure 9-(a)). If the number of slow-memory accesses counted through fake page fault exceeds `th_block`, PET considers the P-block as a promotion target and wakes up `kpromoted` to perform promotion (Figure 9-(b)).

However, if promotion requests accumulate faster than the promotions that can be executed, delays occur, causing additional slow-memory accesses and potentially exceeding `th_total`. To maintain the total slow-memory access rate within R_{promo}, PET promotes all accessed P-blocks once the aggregated slow-memory accesses during the sampling interval reach `th_total`, even if individual units have not yet exceeded their `th_block` (Figure 9-(c)). This approach minimizes performance degradation by preventing excessive slow-memory accesses during promotion delays. At the end of each sampling interval, PET resets the slow-memory access counters, and `th_block` values are recalculated if the total size of demoted P-blocks changes. This dynamic adjustment allows the promotion mechanism to adapt to varying workloads and hence perform robustly.

4.4 File Page Demotion

Linux manages migratable pages by categorizing them into anonymous/file pages. Anonymous pages, which account for most virtual address space of a process, can be effectively managed by using the P-block. In contrast, file pages with cached data from storage exist independently of the process and require separate management. The stock kernel and previous works either prioritize file-backed pages as a demotion target [31, 41] or focus only on anonymous pages [16, 46].

Given that processes access files via `open()` and disconnect files via `close()`, PET regards the files that are not linked to any process (i.e., files called `close()` by all processes that opened them) as demotion targets. To identify demotion target files, PET has a counter (`open_count`) in each `inode`, which counts the number of processes currently using the file. PET adds the `inode` with the zero `open_count` to the linked list of cold files managed by the `kdemoted` thread and demotes the files that exist in the cold file list for each scan interval. This file page demotion is performed in the background by `kdemoted`, so it has minimal impact on the application's performance.

4.5 Comparison with Modern OS Features

The Multi-Generational LRU (MGLRU) [42], recently implemented in Linux, improves upon the traditional LRU by classifying pages into multiple generations—typically four—based on their coldness. This enables proactive demotion by periodically sorting pages by generation and demoting the oldest ones. Proactive demotion using MGLRU provides high access tracking accuracy since it tracks access at the page granularity. However, it requires periodic checking of the access bits for *all pages*, incurring a significant system overhead.

Data Access MONitoring (DAMON) [51] is a Linux kernel subsystem that tracks memory access patterns using region-based sampling. It divides the address space into `damon_regions` and dynamically adjusts their sizes based on observed access patterns, enabling efficient identification of hot and cold memory regions. Recently, DAMON-based tiered memory management [30] was added to the Linux kernel mainline, utilizing its physical address space monitoring feature. Two kernel threads are employed: one scans fast memory to demote cold `damon_regions`, while the other scans slow memory to promote hot `damon_regions`. This approach significantly reduces monitoring overhead and introduces the potential for region-based tiered memory management.

However, it also faces the following challenges. First, it monitors the physical address space, which exhibits less access locality compared to the virtual address space. This makes memory management at the region level less effective for capturing logical locality. Figure 10 shows the memory access patterns for liblinear [17] and Redis [53]. In the case of liblinear, the distinct locality is observed in the virtual address space, whereas the scattered access patterns are shown

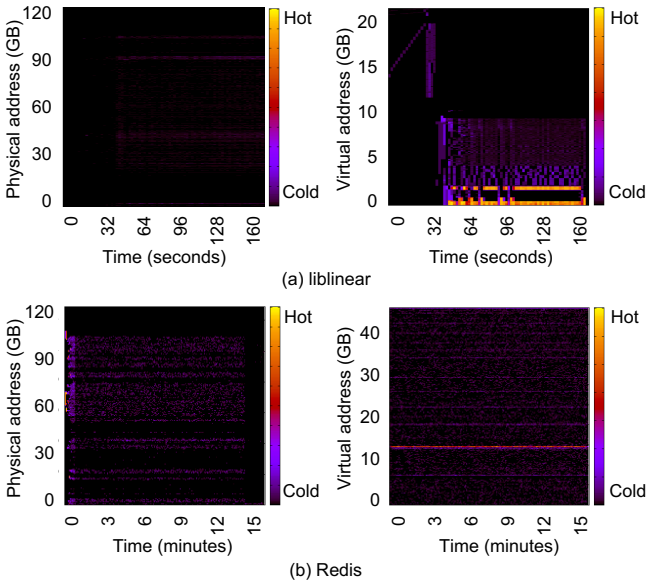

Figure 10. Memory access patterns of (a) XSBench and (b) Redis over time, depicted in both physical (left) and virtual (right) address spaces.

in the physical address space, highlighting the limitations of physical address-based memory management in capturing memory access patterns. Similarly, in Redis, where requests follow a Zipfian distribution, the virtual address space shows clear spatial and temporal locality, with frequently accessed objects grouped together. However, in physical address space, this locality is not evident due to fragmented mappings. Second, a uniform management policy is applied to all `damon_regions` without considering their hotness. As noted in [67], continuous scanning and frequent region adjustments can introduce runtime overhead, especially for applications with rapidly changing access patterns.

PET overcomes the limitations of physical memory space tracking inherent to DAMON-based methods [30] while achieving high accuracy with low overhead through region-based sampling methods. PET monitors the virtual address space of each process, allowing for an easy identification of appropriate monitoring region boundaries. To fully utilize access locality within allocation units, we implement a P-block capturing mechanism, which creates a P-block when a new VMA is generated in the `mmap()` path. Furthermore, PET differentiates tracking methods based on the hotness of the P-blocks: lower overhead, less accurate tracking for hotter P-blocks, and more precise tracking for colder P-blocks to prevent incorrect demotions. This strategy effectively balances tracking overhead and accuracy, optimizing performance while maintaining necessary precision.

PET and [30] are both implemented using kernel threads, but each assigns different roles to its threads. [30] assigns one kernel thread per memory tier, where each thread independently monitors its assigned tier, identifies hot/cold regions, and migrates the target regions accordingly. In contrast, PET utilizes two kernel threads that work cooperatively: `kdemoted` and `kpromoted`. `kdemoted` performs P-block creation, memory access tracking, and demotion, whereas `kpromoted` handles promotion. Typically, `kdemoted` manages P-blocks, but promotion candidate P-blocks are also accessible to `kpromoted`. Therefore, we employ mutex locks to prevent resource contention. In our evaluation, we compare PET with both MGLRU and DAMON-based proactive demotion to highlight these differences and demonstrate the effectiveness of our approach.

5 Evaluation

5.1 Experimental Setup and Methodology

System configuration: We implemented the prototype of PET on the Linux kernel v6.1.44. Our evaluation system is based on the Intel Xeon Platinum 8260 processor. To organize tiered memory, we used Intel's Optane DC Persistent Memory [39] (DCPMM) as slow memory and DRAM as fast memory. The tiered memory consists of 4×16GB of fast memory and 2×128GB of slow memory.

Methodology: We evaluated PET on memory-intensive benchmarks, Graph500 [45], SPECspeed 2017 [8], liblinear [17] with KDD2010/2012 dataset [28], GAP Benchmark Suite (GAPBS) [4] with Twitter graph, Redis [53] with YCSB [11] workload A (update heavy) and B (read heavy), and XSBench [58]. We also used DaCapo benchmark suite [6] written in Java to evaluate PET's performance in adversarial scenarios and developed a microbenchmark where the hot sets periodically shift to assess PET's responsiveness.

We plotted the average results after running each benchmark three times, where the variance was about 1.5% on average. We used the inverse of execution time as a performance metric and compared it to an ideal case, **Fast-only**, where the fast memory capacity is sufficient (>120GB) with Linux kernel v6.1.44. We also compared **PET** with three non-proactive demotion schemes (**Base**, **Base+**, and **AutoTiering** [31]) and five proactive demotion schemes (**Base+PD**, **TPP** [41], **Thermostat** [2], **DAMON** [30], and **MEMTIS** [36]).

Base is the default Linux kernel v6.1.44, which allocates newly created pages to the slow memory when the fast memory is insufficient. By enabling AutoNUMA Memory Tiering [65], we add a promotion feature on **Base** and named **Base+**. **Base+PD** is implemented on top of **Base+** to evaluate proactive demotion using MGLRU by periodically sorting LRU lists and demoting the oldest generation. For **DAMON**, we used the configuration-generating script provided in [21]. Because **MEMTIS** is designed for systems with THP enabled, we compared it separately from other works for fair comparison.

We set the canary page ratio to 10%, sampling/scan intervals to 500ms/10s, and the temporary block and maximum

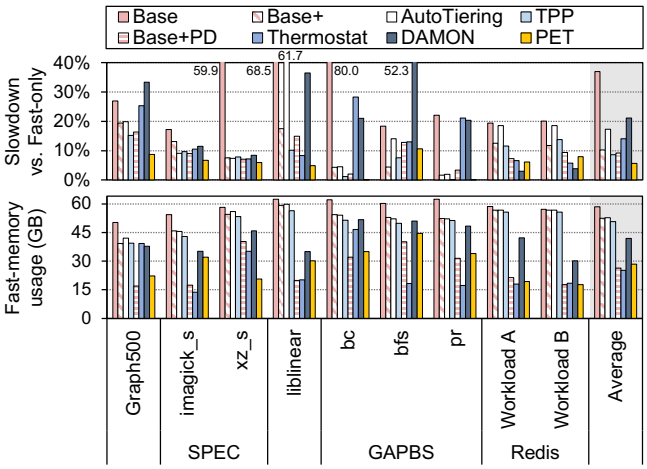

Figure 11. Slowdown and fast-memory usage (*lower is better for both*) compared to **Fast-only** for benchmarks whose working sets are larger than fast-memory capacity.

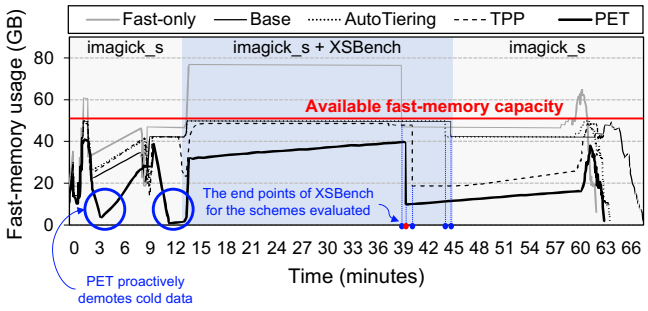

Figure 12. The trend of fast-memory usage over time when the phase of application execution changes for the case of imagick_s with XSBench. We marked the end of XSBench with a red dot for **PET** and blue dots for the other schemes.

P-block sizes to 128MB and 1,024MB. §5.3 describes the sensitivity analyses on these values. For a fair comparison with **PET**, **Base+PD**'s interval to sort the list and trigger demotion is set to 10s, equal to **PET**'s scan interval.

5.2 Performance of PET

5.2.1 When the aggregate working set is larger than the fast-memory capacity. **PET** mitigates performance drop by preventing memory allocation/promotion accompanied by demotion and responding to the sudden peak of fast-memory demand, outperforming prior OS-level schemes when the system's memory usage is larger than the fast-memory capacity. We verified it under the benchmarks that have lots of cold pages and are tuned to have a working set size of 1.5 times as much as the fast-memory capacity (80GB to 100GB), referring to Meta's target configuration for applying tiered memory to data centers [41]. Also, we executed the SPEC and GAPBS benchmarks in a multi-programmed manner, reflecting a scenario where multiple (five to ten) processes compete for fast memory.

PET achieves a light performance drop, 5.7% on average, compared to **Fast-only**, where all working sets are allocated to fast memory (see Figure 11). **AutoTiering**, an early study that proposes effective memory management techniques in multi-tiered memory, performs worse than the other schemes due to the non-proactive demotion, especially in benchmarks that experience sudden peaks in the demand for fast memory (e.g., liblinear). **PET**, **TPP**, and **Base+PD** achieve relatively high performance on average because they rarely fail in the promotion and fast-memory allocation by triggering demotion proactively. **Thermostat** suffers an average of 14.0% slowdown. As its access tracking mechanism causes excessive fake page faults, workloads with abrupt changes in access patterns, such as Graph500 and bc, significantly underperform. Thanks to proactive demotion, **Base+PD** performs slightly better than **Base+**. However, it performs worse over **PET** in benchmarks that exhibit high locality within the memory allocation unit (e.g., bc and xz_s) because **PET** effectively lowers the overhead for identifying memory access patterns and managing data placement. Although **DAMON** is also a proactive demotion scheme, it shows the lowest performance aside from **Base**. We attribute this performance drop to the inaccuracy of its access tracking method, which scans the physical address space. **PET** accurately identifies cold data with its multi-phase demotion and makes swift promotion decisions in the P-block granularity, achieving high performance on most of the benchmarks.

PET also runs benchmarks with a smaller fast-memory capacity by proactively demoting cold data regardless of fast-memory usage. The bottom graph in Figure 11 shows the average fast-memory usage by each scheme to run benchmarks. **PET** achieves its performance using only 28.4GB of fast-memory capacity on average, which is over 21GB less compared to **Base**, **Base+**, **AutoTiering**, and **TPP**. As shown in imagick_s or liblinear, **Base+PD** and **Thermostat** sometimes exhibit lower fast-memory usage than **PET**. However, this reduction comes at the cost of higher performance degradation. On average, **Base+PD** and **Thermostat** use 2.03GB and 3.17GB less fast memory than **PET**, respectively, but suffer from additional slowdowns of 8.36% and 3.49%, respectively.

The performance of **PET** can be attributed to its proactive demotion, which effectively reduces fast-memory usage, enabling it to handle sudden spikes in fast-memory demand. To verify this, we tracked fast-memory usage over time in our synthetic workload that mimics phase changes in application execution. We concurrently executed XSBench, a memory-bound benchmark, and imagick_s, a compute-bound benchmark having lots of cold data. We first execute imagick_s, which has a longer execution time, and execute XSbench when the execution time of imagick_s has passed

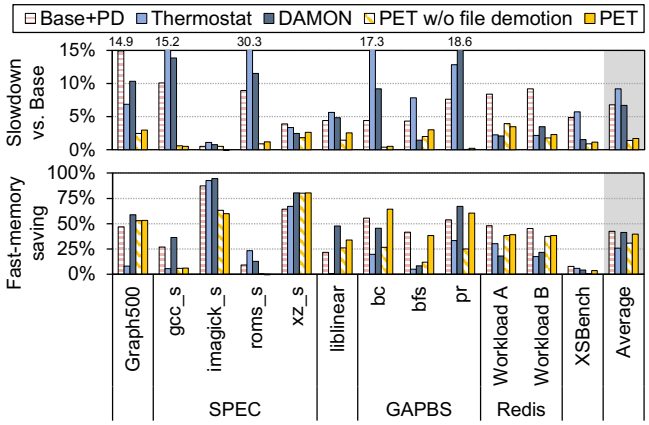

Figure 13. Slowdown and fast-memory saving (*lower/higher is better*) compared to **Base** when the aggregate working set fits in fast memory.

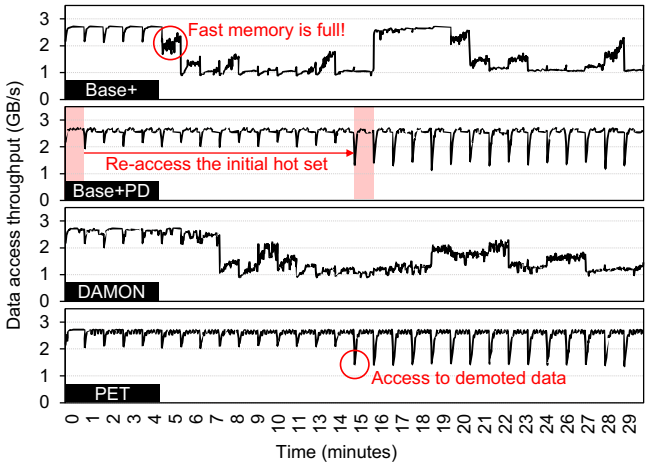

Figure 14. The amount of accessed data over time when hot sets are moved periodically (*higher is better*).

about 20% of its total execution time. Also, to verify the benefits of **PET**'s reduced fast-memory usage on performance, we limited the available fast memory to 50GB using `mlock` and compared **PET** with schemes that conservatively trigger demotion (**Base**, **AutoTiering**, and **TPP**).

Before the XSBench starts, **PET** saves 90.7% of fast-memory usage of imagick_s; thus, XSBench can allocate memory without a deficiency of fast-memory capacity (see Figure 12). Therefore, **PET** executes imagick_s and XSBench only with 2.3% and 1.9% slowdown, respectively, using just 50GB of fast memory. With the limited fast-memory capacity, **Base** and **AutoTiering** suffer from 17.2% on XSBench because the fast memory occupied by imagick_s limits fast-memory allocation for XSBench. **TPP** mitigates this situation, but it still suffers from a 4.80% slowdown on XSBench, which is worse than the result of **PET**.

5.2.2 For benchmarks with working sets that fit in fast memory.

Even when the aggregate working set fits in fast memory, **PET** saves significant fast memory with a low-performance drop compared to prior works [2, 16, 35, 60] focusing on RSS reduction (up to 25%). We verified it under the benchmarks configured to have a smaller memory footprint (~45GB) than fast-memory capacity. Thus, **Base**, **Base+**, **AutoTiering**, and **TPP** do not perform any demotion/promotion in this configuration. We used the slowdown compared to **Base** as a performance metric. We measured the fast-memory saving rate by comparing the DRAM usage over time to the **Base**.

PET without file demotion achieves 30.7% fast-memory savings with only a 1.40% average performance drop compared to **Base** (see Figure 13). **Thermostat** also shows large fast memory savings (25.7% on average), but it decreases performance by 9.22% on average due to repetitive page faults, especially when most data are hot (e.g., roms_s). **Base+PD** and **DAMON** achieve larger fast-memory savings and lower performance degradation than **Thermostat**. However, in workloads like gcc_s, which allocates numerous small objects, both **Base+PD** and **DAMON** aggressively reclaim fast memory, leading to significant performance degradation. While **PET** achieves a 5.8% fast-memory reduction in gcc_s, it avoids performance penalties by tracking only the anon_mmap region and refraining from managing areas where many small objects are allocated. **Base+PD** also experiences substantial performance drops in benchmarks like Graph500 and roms_s, where frequently accessed data that had been previously demoted is subsequently accessed again. These results suggest that **PET** effectively and proactively demotes cold data and rapidly promotes data that transitions from cold to hot, thereby preventing performance degradation.

PET with file demotion saves an additional fast memory on benchmarks with large file-page usages, such as liblinear and GAPBS, with negligible performance drop (39.8% fast memory saving with 1.7% performance drop on average). In particular, **PET** reduces fast-memory usage by 49.3% with an average 1.6% performance drop on liblinear and GAPBS that use large amounts of file pages.

5.2.3 Reactiveness of P-block granularity access tracking.

Proactive demotion must swiftly identify changes in memory access patterns and promote hot sets in the slow memory before they harm system performance. To evaluate the reactiveness of our P-block granularity access tracking, we developed a microbenchmark that periodically shifts its hot sets. In this benchmark, eight threads randomly access each 1GB hot set for one minute and relocate the hot set every minute over a 15-minute period, resulting in a total working set of 120GB. After completing this cycle, the hot sets returned to their initial regions, and the operations were repeated once more. We compared **PET** with the default Linux kernel's tiered memory management solution (**Base+**) as well as with the proactive demotion techniques

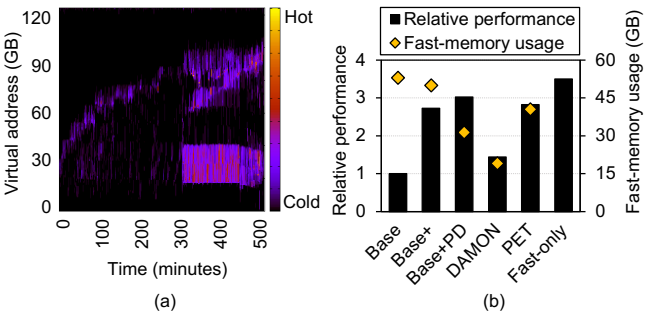

Figure 15. (a) Memory access pattern in Java heap. (b) Relative performance compared to **Base** and fast-memory usage.

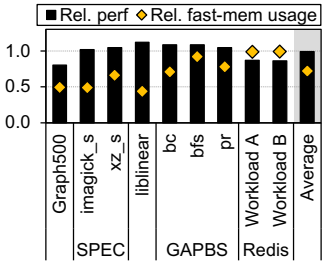

Figure 16. Relative performance and fast-memory usage of **PET** compared to **MEMTIS** with THP enabled.

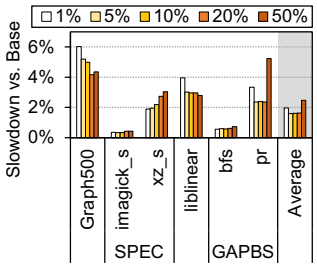

Figure 17. Slowdown compared to **Base** according to various canary ratios.

implemented using the modern OS features (**Base+PD** and **DAMON**). We define *reaction time* as the time it takes for a scheme to restore 95% of its maximum data access throughput after it begins accessing demoted data.

As shown in Figure 14, **Base+**, not performing proactive demotion, experiences a sharp performance decline once fast memory becomes insufficient and struggles to recover promptly despite using hint fault-based promotion. In contrast, **Base+PD**, which uses MGLRU for proactive demotion, maintains consistently high performance and shows an average of 12 seconds of reaction time. Another proactive demotion scheme, **DAMON**, initially maintains a higher performance than **Base+**. However, it eventually experiences a large performance drop, falling below **Base+** on the second cycle. We attribute this to the inaccuracy in its region-based access tracking over physical address space. **Base+** and **DAMON** fail to recover their data access throughput after performance degradation, so we do not evaluate reaction time for theses schemes. Despite performing P-block-granularity sampling-based access tracking, **PET** achieves the same maximum data access throughput as **Base+PD** with only a 1-second longer reaction time. Remarkably, **PET** scans only 0.87% of the pages compared to **Base+PD** while maintaining comparable performance. This highlights **PET**'s ability to deliver performance comparable to page-level tracking schemes while significantly reducing the amount of data that needs to be scanned.

5.2.4 Performance in adversarial scenarios for PET.

PET is designed under the assumption that access locality exists within a memory allocation unit. However, in managed languages such as Java, where Java Virtual Machine (JVM) allocates a large memory chunk and handles allocation and deallocation internally, the expected access locality might not exist within the P-block. To assess **PET**'s performance in this unfavorable scenario, we used the H2 workload from the DaCapo benchmark suite [6] written in Java. We scaled the working set size to over 100GB by increasing the scale factor to 1024 and setting the maximum Java heap size to 100GB. We used the time taken to process a total of five million transactions as the performance metric, excluding the time required to initialize or reset the database.

As shown in Figure 15-(a), the H2 benchmark exhibits limited access locality within the Java heap, making it difficult to accurately identify cold pages. Moreover, H2 workloads are highly sensitive to slower main memory speeds [13]. Thus, all schemes suffer more than 10% performance degradation compared to **Fast-only**. **DAMON**, a region-granularity proactive demotion scheme, performs the worst, as it misclassifies excessive regions as cold (Figure 15-(b)). **PET**, while also a region-based scheme, takes a more conservative approach by tracking coldness and performing demotion over multiple phases. This significantly reduces the number of false demotions compared to **DAMON** and delivers nearly twice the performance of **DAMON**. In contrast, **Base+PD**, which operates at page granularity, more accurately detects cold pages and thus outperforms **PET** by 7%. To address this, we plan to extend **PET** to intelligently detect the absence of memory access locality within an allocation unit and adaptively adjust the granularity of tracking and demotion.

5.2.5 Performance with Transparent Huge Page (THP).

We further evaluated **PET**'s performance with THP enabled and compared it to **MEMTIS**, a state-of-the-art scheme for managing page placement and size in tiered memory systems with THP enabled (Figure 16). When THP is enabled, a THP is considered accessed if an access occurs to any of the 512 base pages inside it. Thus, the management schemes that rely on the access bits like **PET** intermittently overestimate the hotness of pages (or P-block), performing sub-optimally in workloads with rapidly changing access patterns. However, **PET** performs nearly the same to **MEMTIS**, while using only 72% of the fast memory on average. **MEMTIS**, utilizing PEBS to track data hotness with fine granularity, has inherent limitations in detecting infrequently accessed cold data [36]. In contrast, **PET** effectively identifies cold data and proactively demotes them, outperforming **MEMTIS** when a substantial amount of cold data is present.

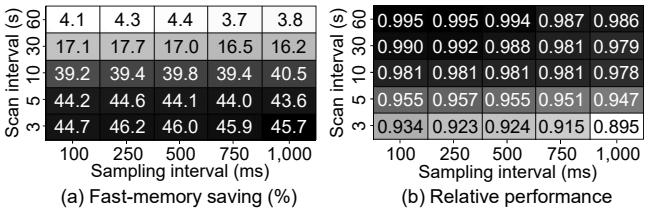

Figure 18. Heatmaps depicting the impacts of sampling/scan intervals on fast-memory saving and performance.

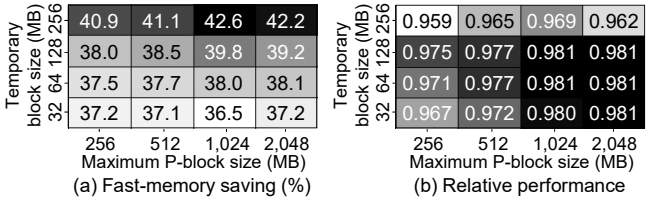

Figure 19. Heatmaps depicting the impacts of temporary block and max P-block sizes on fast-memory saving and performance.

5.3 Sensitivity Study

To check sensitivities to the canary page ratio, the interval values, and the P-block splitting sizes, we measured the performance and the fast-memory usage of the benchmarks (Graph500, imagick_s, xz_s, liblinear, bfs, and pr) configured to have a working set fit in the fast memory.

5.3.1 Sensitivity to the canary page ratio. The canary page plays a crucial role in verifying coldness for the last time before a P-block is demoted and in reflecting the hotness of the demoted P-blocks. While a higher canary page ratio improves tracking accuracy, an excessively high ratio can cause premature demotion, leading to unnecessary demotions and fake page faults. Due to this trade-off, as shown in Figure 17, the sensitivity to the canary ratio varies depending on the memory access pattern of individual benchmarks. For liblinear, which has complex memory access patterns, a lower canary page ratio leads to mis-demoting hot P-blocks, degrading performance. In contrast, for xz_s, which has a large amount of cold data and a simple access pattern, the overhead from fake page faults induced by a higher canary page ratio outweighs the benefits of obtaining better tracking precision. Experimental analysis across diverse workloads shows that a 10% canary page ratio provides stable and balanced performance.

5.3.2 Sensitivity to the interval values. It is noticeable primarily that the scan interval has a more significant impact on performance and fast-memory saving than the sampling interval (Figure 18). Decreasing the scan interval improves fast-memory savings but negatively impacts performance; intervals lower than 10 seconds achieve over 40% fast-memory savings but result in an average performance drop exceeding 4.3%. While a lower sampling interval enhances tracking accuracy, frequent sampling incurs overhead that leads to diminishing returns in performance. Therefore, we set the default sampling interval to 500ms and scan interval to 10s, balancing performance and fast-memory saving.

5.3.3 Sensitivity to the P-block splitting sizes. A larger temporary block increases fast-memory saving, but excessively large blocks (i.e., 256MB) can degrade performance due to increased mis-demotion risks (Figure 19). The maximum size of a P-block does not significantly impact fast-memory saving and performance, but a larger P-block performs slightly better on average due to a performance increase in benchmarks such as Graph500 and liblinear. Because they allocate large sizes of memory objects containing high access locality, sudden massive accesses to the slow memory could occur when these objects are demoted and accessed again soon. Using a larger P-block enables faster promotion decisions for these peaks of slow memory access compared to using numerous smaller P-blocks, improving performance. Consequently, we chose a temporary block size of 128MB and a maximum P-block size of 1,024MB that shows the highest performance with a reasonable fast-memory saving.

6 Discussion

6.1 Dynamic adjustment of the canary page ratio

In §5.3.1, we demonstrate that a 10% canary ratio delivers stable performance across diverse workloads. However, a static ratio may not be optimal for all workloads and system conditions. To address this issue, we plan to dynamically adjust the canary page ratio in response to real-time memory access patterns and workload intensity. For memory-intensive applications, a lower ratio can reduce the risk of page faults, whereas a higher ratio can more accurately capture coldness under lighter workloads. This adaptive mechanism aims to balance demotion accuracy and overhead, ensuring optimal performance across varying runtime conditions.

6.2 Handling memory thrashing in PET

PET assumes that the size of the fast memory is sufficient to accommodate the *hot working set*. However, when the hot working set size exceeds fast memory capacity, excessive hot/cold page swapping or memory thrashing can occur, undermining the benefits of proactive demotion. In such scenarios, PET's continuous tracking and migration increase overhead without improving performance. To mitigate thrashing, we plan to develop a feedback-based adjustment mechanism that detects rapid thrashing and temporarily suspends PET's memory management, including memory tracking and migration. This approach is similar to existing solutions like

HeMem [52] and TMTS [16], which manage thrashing by either halting page promotion and demotion or protecting frequently accessed pages from demotion. By temporarily suspending its operations, PET allows the system to stabilize without additional overhead. Once normal memory conditions are restored, PET seamlessly resumes its mechanisms, maintaining an optimal balance between performance and overhead.

7 Related Work

OS-level tiered memory management schemes: There have been various OS-level schemes to manage tiered memory. They suggest effective mechanisms to identify page hotness [20, 26, 40, 52, 64], track access [5, 36, 46, 52, 54], manage multi-tiered memory [9, 31, 54], optimize page migration operations [54, 64], and migrate in-kernel data [1, 27, 34, 50]. However, most rely on tight demotion thresholds, unlike PET. Several works [2, 16, 35, 36, 41, 60, 63] propose proactive demotion, but use OS-page granularity, which incurs high management overhead as the working set size increases. [43] presents a preliminary study that exploits the memory allocation unit for proactive demotion but uses static management units and belatedly performs promotion. PET proposes a proactive demotion mechanism that performs the management unit adjustment and a swift promotion, enhancing the effectiveness of using the memory allocation unit.

Schemes focusing on the memory allocation unit: There are other works that focus on the memory allocation unit for tiered memory management. They collect the memory access information and decide the optimal data placement either at compile-time [15, 19, 29, 44, 47, 48] or run-time [10, 55, 61, 62]. However, these application-level solutions are hard to consider the system's runtime behavior, especially when multiple workloads run concurrently, and require the application's allocation information in advance.

8 Conclusion

In this paper, we have introduced PET, an operating system-level solution designed to enhance tiered memory management. PET extends the granularity of memory management to the P-block level, presenting sophisticated mechanisms for demotion and promotion based on these P-blocks. PET reduces fast-memory capacity requirements while minimizing the performance degradation from slow memory accesses in tiered memory. The prototype of PET implemented on Linux kernel v6.1.44 shows a 39.8% reduction in fast-memory usage, with only a 1.7% performance drop compared to the default Linux kernel, in scenarios where the system's memory usage fits within fast memory. It also mitigates performance degradation by 31% compared to the default Linux kernel, surpassing other state-of-the-art solutions when memory usage exceeds fast-memory capacity.

Acknowledgments

We thank the anonymous reviewers for their constructive comments and suggestions. This paper was supported by SNU-SK hynix Solution Research Center (S3RC) and an Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (RS-2021-II211343, IITP-2023-RS-2023-00256081, and RS-2024-00469698). Wanju Doh and Seunghwan Chung are with the Department of Interdisciplinary Program in Artificial Intelligence (IPAI), Seoul National University (SNU). Seoyoung Ko and Kwanhee Kyung are with the Department of Intelligence and Information, SNU. This work was done when Yaebin Moon was at SNU. Jung Ho Ahn, the corresponding author, is with the Department of Intelligence and Information, Inter-University Semiconductor Research Center, and IPAI, SNU.

References

[1] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 283–300, 2020.

[2] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 631–644, 2017.

[3] AMD. AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://www.amd.com/system/files/TechDocs/24593.pdf, 2017.

[4] Scott Beamer, Krste Asanović, and David Patterson. The GAP Benchmark Suite, 2017.

[5] Shai Bergman, Priyank Faldu, Boris Grot, Lluís Vilanova, and Mark Silberstein. Reconsidering OS Memory Optimizations in the Presence of Disaggregated Memory. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, page 1–14, 2022.

[6] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The dacapo benchmarks: java benchmarking development and analysis. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems*, page 169–190, 2006.

[7] Daniel P. Bovet and Marco Cesati. Understanding the Linux Kernel. *San Francisco: Oreilly & Associates Inc*, pages 357–362, 2007.

[8] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. SPEC CPU2017: Next-Generation Compute Benchmark. In *Companion of the ACM/SPEC International Conference on Performance Engineering*, page 41–42, 2018.

[9] Juneseo Chang, Wanju Doh, Yaebin Moon, Eojin Lee, and Jung Ho Ahn. IDT: Intelligent Data Placement for Multi-tiered Main Memory with Reinforcement Learning. In *Proceedings of the 33rd International Symposium on High-Performance Parallel and Distributed Computing*, 2024.

[10] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. ATMem: Adaptive Data Placement in Graph Applications on Heterogeneous Memories. In *Proceedings of the ACM/IEEE International Symposium on Code Generation and Optimization*, 2020.

[11] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing*, 2010.

[12] Compute Express Link. https://www.computeexpresslink.org.

[13] DaCapo descriptions and statistics. https://github.com/dacapobench/dacapobench/blob/main/benchmarks/doc/dacapo-descriptions-and-statistics.pdf.

[14] Arnaldo Carvalho de Melo. Performance Counters on Linux. In *Linux Plumbers Conference*, 2009.

[15] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data Tiering in Heterogeneous Memory Systems. In *Proceedings of the 11th European Conference on Computer Systems*, 2016.

[16] Padmapriya Duraisamy, Wei Xu, Scott Hare, Ravi Rajwar, David 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. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, page 727–741, 2023.

[17] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A Library for Large Linear Classification. *J. Mach. Learn. Res.*, 9:1871–1874, jun 2008.

[18] Gen-Z. https://genzconsortium.org/.

[19] Derrick Greenspan. LLAMA - Automatic Memory Allocations: An LLVM Pass and Library for Automatically Determining Memory Allocations. In *Proceedings of the International Symposium on Memory Systems*, 2019.

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

[21] SK hynix. HMSDK. https://github.com/skhynix/hmsdk/blob/main/tools/gen_config.py, 2023.

[22] IBM. POWER9 Performance Monitor Unit User's Guide. https://wiki.raptorcs.com/w/images/6/6b/POWER9_PMU_UG_v12_28NOV2018_pub.pdf, 2018.

[23] Intel. Intel VTune Profiler User Guide. https://software.intel.com/en-us/vtune-help, 2020.

[24] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes 3B: System Programming Guide. https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-volume-3b-system-programming-guide-part-2, 2021.

[25] Intel. Tiering-0.72. https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git/, 2021.

[26] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, page 521–534, 2017.

[27] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. KLOCs: Kernel-Level Object Contexts for Heterogeneous Memory Systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 65–78, 2021.

[28] LIBSVM Data: Classification, Regression, and Multi-label. https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

[29] Dounia Khaldi and Barbara Chapman. Towards Automatic HBM Allocation Using LLVM: A Case Study with Knights Landing. In *3rd Workshop on the LLVM Compiler Infrastructure in HPC*, pages 12–20, 2016.

[30] Honggyu Kim. DAMON based tiered memory management for CXL memory. https://lwn.net/Articles/978313/.

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

[32] Kyungsan Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Junhyuk Im, Sungjoo Park, Jeonghyeon Cho, and Hoyoung Song. SMT: Software-Defined Memory Tiering for Heterogeneous Computing Systems With CXL Memory Expander . *IEEE Micro*, 43(02):20–29, March 2023.

[33] Apostolos Kokolis, Dimitrios Skarlatos, and Josep Torrellas. PageSeer: Using Page Walks to Trigger Page Swaps in Hybrid Memory Systems. In *IEEE International Symposium on High Performance Computer Architecture*, pages 596–608, 2019.

[34] Sandeep Kumar, Aravinda Prasad, Smruti R Sarangi, and Sreenivas Subramoney. Radiant: efficient page table management for tiered memory systems. In *Proceedings of the ACM SIGPLAN International Symposium on Memory Management*, page 66–79, 2021.

[35] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, Greg Thelen, Kamil Adam Yurtsever, Yu Zhao, and Parthasarathy Ranganathan. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 317–330, 2019.

[36] Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. MEMTIS: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *Proceedings of the Symposium on Operating Systems Principles*, page 17–34, New York, NY, USA, 2023. Association for Computing Machinery.

[37] Yongjun Lee, Jongwon Kim, Hakbeom Jang, Hyunggyun Yang, Jangwoo Kim, Jinkyu Jeong, and Jae W Lee. A Fully Associative, Tagless DRAM Cache. In *ACM/IEEE 42nd Annual International Symposium on Computer Architecture*, pages 211–222, 2015.

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

[39] Lily Looi and Jianping Jane Xu. Intel Optane Data Center Persistent Memory. In *IEEE Hot Chips 31 Symposium (HCS)*, pages i–xxv, 2019.

[40] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture*, pages 925–937, 2022.

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

[42] Multi-Generational LRU: The Next Generation. https://lwn.net/Articles/856931/.

[43] Yaebin Moon, Wanju Doh, Kwanhee Kyung, Eojin Lee, and Jung Ho Ahn. ADT: Aggressive Demotion and Promotion for Tiered Memory. *IEEE Computer Architecture Letters*, 22(1):21–24, 2023.

[44] Diego Moura, Daniel Mossé, and Vinicius Petrucci. Performance characterization of autonuma memory tiering on graph analytics. In *IEEE International Symposium on Workload Characterization*, pages 171–184, 2022.

[45] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the Graph500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[46] Alan Nair, Sandeep Kumar, Aravinda Prasad, Andy Rudoff, and Sreenivas Subramoney. Telescope: Telemetry at Terabyte Scale. *arXiv:2311.10275*, 2023.

[47] Aditya Narayan, Tiansheng Zhang, Shaizeen Aga, Satish Narayanasamy, and Ayse Coskun. MOCA: Memory Object Classification and Allocation in Heterogeneous Memory Systems.

In *IEEE International Parallel and Distributed Processing Symposium*, pages 326–335, 2018.

[48] Deok-Jae Oh, Yaebin Moon, Do Kyu Ham, Tae Jun Ham, Yongjun Park, Jae W. Lee, Jung Ho Ahn, and Eojin Lee. MaPHeA: A Framework for Lightweight Memory Hierarchy-Aware Profile-Guided Heap Allocation. *ACM Transactions on Embedded Computer Systems*, 22(1), 2022.

[49] Page Migration. https://lwn.net/Articles/157066/.

[50] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K Gopinath, and Jayneel Gandhi. Fast Local Page-Tables for Virtualized NUMA Servers with VMitosis. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 194–210, 2021.

[51] SeongJae Park. DAMON: Data Access Monitor. https://docs.kernel.org/mm/damon/index.html.

[52] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 392–407, 2021.

[53] Redis. redis.io. https://redis.io, 2020.

[54] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *Proceedings of the European Conference on Computer Systems*, page 803–817, 2024.

[55] Harald Servat, Antonio J. Peña, Germán Llort, Estanislao Mercadal, Hans-Christian Hoppe, and Jesús Labarta. Automating the Application Data Placement in Hybrid Memory Systems. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2017.

[56] Kevin Song, Jiacheng Yang, Sihang Liu, and Gennady Pekhimenko. Lightweight Frequency-Based Tiering for CXL Memory Systems. *arXiv:2312.04789*, 2023.

[57] Yan Sun, Jongyul Kim, Douglas Yu, Jiyuan Zhang, Siyuan Chai, Michael Jaemin Kim, Hwayong Nam, Jaehyun Park, Eojin Na, Yifan Yuan, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. M5: Mastering Page Migration and Memory Management for CXL-based Tiered Memory Systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2025.

[58] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. XSBench-The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. *The Role of Reactor Physics toward a Sustainable Future*, 2014.

[59] Evangelos Vasilakis, Vassilis Papaefstathiou, Pedro Trancoso, and Ioannis Sourdis. Hybrid2: Combining Caching and Migration in Hybrid Memory Systems. In *IEEE International Symposium on High Performance Computer Architecture*, 2020.

[60] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 609–621, 2022.

[61] Kai Wu, Yingchao Huang, and Dong Li. Unimem: Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Main Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[62] Kai Wu, Jie Ren, and Dong Li. Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Memory for Task-Parallel Programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2018.

[63] Dong Xu, Junhee Ryu, Kwangsik Shin, Pengfei Su, and Dong Li. FlexMem: Adaptive page profiling and migration for tiered memory. In *USENIX Annual Technical Conference*, pages 817–833, 2024.

[64] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 331–345, 2019.

[65] Huang Ying. NUMA balancing: optimize memory placement for memory tiering system. https://lwn.net/Articles/849095/.

[66] Xiangyao Yu, Christopher J Hughes, Nadathur Satish, Onur Mutlu, and Srinivas Devadas. Banshee: Bandwidth-Efficient DRAM Caching via Software/Hardware Cooperation. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, page 1–14, 2017.

[67] Yuhong Zhong, Daniel S Berger, Carl Waldspurger, Ishwar Agarwal, Rajat Agarwal, Frank Hady, Karthik Kumar, Mark D Hill, Mosharaf Chowdhury, and Asaf Cidon. Managing Memory Tiers with CXL in Virtualized Environments. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2024.

[68] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Yongqiang Xiong, and Guangyu Sun. Toward CXL-Native Memory Tiering via Device-Side Profiling. *arXiv:2403.18702v2*, 2024.

[69] Zhe Zhou, Yiqi Chen, Tao Zhang, Yang Wang, Ran Shu, Shuotao Xu, Peng Cheng, Lei Qu, Yongqiang Xiong, Jie Zhang, and Guangyu Sun. NeoMem: Hardware/Software Co-Design for CXL-Native Memory Tiering . In *IEEE/ACM International Symposium on Microarchitecture*, pages 1518–1531, Los Alamitos, CA, USA, November 2024. IEEE Computer Society.