FlexMem: Adaptive Page Profiling and Migration for Tiered Memory

Dong Xu, *University of California, Merced;* Junhee Ryu, Jinho Baek, and Kwangsik Shin, *SK hynix;* Pengfei Su and Dong Li, *University of California, Merced*

https://www.usenix.org/conference/atc24/presentation/xu-dong

This paper is included in the Proceedings of the 2024 USENIX Annual Technical Conference.

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

978-1-939133-41-0

Open access to the Proceedings of the 2024 USENIX Annual Technical Conference is sponsored by

جامعة الملك عبدالله
للعلوم والتقنية
King Abdullah University of
Science and Technology

FlexMem: Adaptive Page Profiling and Migration for Tiered Memory

Dong Xu
University of California, Merced

Junhee Ryu
SK hynix

Jinho Baek
SK hynix

Kwangsik Shin
SK hynix

Pengfei Su
University of California, Merced

Dong Li
University of California, Merced

Abstract

Tiered memory, combining multiple memory components with different performance and capacity, provides a cost-effective solution to increase memory capacity and improve memory utilization. The existing system software to manage tiered memory often has limitations: (1) rigid memory profiling methods that cannot timely capture emerging memory access patterns or lose profiling quality, (2) rigid page demotion (i.e., the number of pages for demotion is driven by an invariant requirement on free memory space), and (3) rigid warm page range (i.e., emerging hot pages) that leads to unnecessary page demotion from fast to slow memory. To address the above limitations, we introduce FlexMem, a page profiling and migration system for tiered memory. FlexMem combines the performance counter-based and page hinting fault-based profiling methods to improve profiling quality, dynamically decides the number of pages for demotion based on the needs of accommodating hot pages (i.e., frequently accessed pages), and dynamically decides the warm page range based on how often the pages in the range is promoted to hot pages. We evaluate FlexMem with common memory-intensive benchmarks. Compared to the state-of-the-art (Tiering-0.8, TPP, and MEMTIS), FlexMem improves performance by 32%, 23%, and 27% on average respectively.

1 Introduction

Motivation. The surge of memory cost and memory consumption creates challenges for building efficient data centers while reducing total cost of ownership (TCO). It has been reported that memory accounts for 50% of Microsoft Azure's server costs [68] and 37% of Meta's server costs [43]. On the other hand, the memory consumption of many applications (e.g., quantum mechanical methods [8,41], in-memory database [24,25,31], graph analysis [6,7,67,81,87], and large AI model inference and training [44,54,61,69]) easily reach the TB scale and call for the support of larger memory.

Recently memory tiering appears as a cost-effective solution to reconcile the conflict between the increase of memory cost and increase of the demands for larger memory capacity. Memory tiering connects multiple memory components with different capacities and performance (in terms of latency and bandwidth). In a tiered memory system, the fast-memory tier is characterized with higher performance but smaller capacity, while the slow-memory tier is characterized with higher capacity but lower performance. As a result, the tiered memory can bring larger memory capacity with lower cost. Furthermore, the recent emergence of advanced memory technologies (such as Compute Express Link (CXL) [1,27,85] and non-volatile Optane memory [19]) makes the tiered memory feasible and performance-beneficial [13,20–23,51,62,72].

The tiered memory demands a memory management system because of performance asymmetry across memory components. Such a system should make the best use of the fast-memory tier for high performance while minimizing the page management overhead. The overhead can come from memory profiling to determine page access frequency and recency, page migration, and page formation (forming and splitting huge pages). The overhead can also come from efforts to mitigate the translation cost of the address in the high capacity memory tier [40,49,50], classify pages [22,26,43,51,71], and avoid page ping pong migration [26,43,56].

Problems. There are recent solutions to manage the tiered memory from the perspective of system software [22,26,43,51,56,71,82]. These efforts often inherit traditional designs in memory systems to build a unified, NUMA-based memory abstract and page migration (e.g., AutoNUMA [10], AutoTiering [22], and Tiering-0.8 [71]). These efforts also customize the traditional designs to support the tiered memory systems (e.g., providing flexibility for cross-tier page migration [22], supporting rich memory tiers [22,56,71], and decoupling memory allocation and reclaim [43]). However, the inherent limitations in the traditional designs prevent the effectiveness of the recent solutions from handling the tiered memory. We summarize these limitations as follows.

First, the page profiling method lacks the flexibility to timely capture time-changing access patterns. The common page-profiling methods can be broadly classified into perfor-

mance counter-based and NUMA hint fault-based (or *fault-based* for short). The performance counter-based profiling method typically uses multiple time intervals to accumulate memory accesses to pages to confirm the page hotness and prevent false positive detection of hot pages. This method can accurately capture the invariant hot-page set, but be slow to adapt to time-changing access patterns. The fault-based profiling method decides the page hotness based on the records of sporadic page accesses, which can opportunistically capture the emerging hot pages, but can mistakenly identify a cold page as hot. In conclusion, there is no profiling method that is sensitive to the change of the hot-page set while remaining accurate.

Second, the page demotion (from fast memory to slow memory) lacks the flexibility to accommodate upcoming page promotion (from slow memory to fast memory). The page demotion migrates less frequently accessed pages (cold pages) to slow memory, which is inevitable to save fast memory space for hot pages to promote. Page demotion is critical for an application with varying execution phases and dynamic hot page sets, which demands frequent page exchanges between fast and slow memories. However, we observe that the current page demotion in the traditional designs is highly ineffective, leading to a large number of page promotion failures (detailed in Section 3). The existing solutions for the tiered memory largely inherit this limitation, and cannot timely promote hot pages and hence lose performance, no matter how page promotion is optimized.

Our solution. We introduce a transparent page management system for the tiered memory, FlexMem, aiming to address the above problems.

To address the page profiling problem, we combine the performance counter-based and fault-based profiling methods. The major challenges of combining them come from how to coordinate the two methods in terms of page-promotion timing and reconcile the disagreement between the two methods regarding the decision of hot pages. To address the challenges, FlexMem uses a unified promotion interval and gives the two profiling methods equal opportunities to contribute hot pages; also, the hot pages identified by the fault-based method, although recognized as cold by the performance counter-based method, are not immediately demoted until the two methods reach agreement on the page hotness.

To address the page demotion problem, we realize that the current page demotion is essentially driven by the availability of fast memory space, not by the urgency of page promotion. As a result, the constant pace of page demotion employed in the existing solutions may not be able to match the pace of page migration, hence leading to frequent page promotion failures. We introduce a mechanism to dynamically adapt the demotion rate (i.e., how fast to demote pages) guided by the frequency of page promotion failures and effectiveness of recent page promotion.

Beyond the above contributions, we identify the ineffectiveness of the warm page mechanism in the state-of-the-art [26, 56]. This mechanism is used to prevent unnecessary demotion: some pages are becoming less frequently accessed but are not demoted as cold pages; instead, they are classified as warm pages without demotion unless the free fast memory space is really scarce. The existing mechanism classifies pages as warm when their position in the memory access histogram is only *one* bin away from hot pages. This classification method is rigid because those warm pages whose positions are farther can be demoted and promoted later on, which is ineffective. To address this problem, FlexMem dynamically changes the range of warm bins based on the estimation of the bin's potential to become hot.

We summarize the major contributions of this paper as follows.

- We identify three performance problems based on a thorough analysis of the state-of-the-art page management systems for the tiered memory: (1) differences in guiding page promotion between different memory profiling methods, (2) frequent promotion failures, and (3) ineffective identification of warm pages.
- We introduce FlexMem, a page management system for the tiered memory, aiming to address the three problems based on adaptive page profiling (using two profiling methods to get the best of two worlds), adaptive page demotion to reduce page promotion failures, and adaptive warm bins.
- We evaluate FlexMem with representative memory-intensive applications and compare FlexMem with three state-of-the-art systems (Tiering-0.8 [71], TPP [43], and MEMTIS [26]). FlexMem outperforms them by 28% on average (geomean). FlexMem reduces the page migration failure by 25% and improves the fast memory usage by 21%.

2 Background

We review the background information in this section.

2.1 Page Migration in Tiered Memory

A page, once recognized as hot, will be promoted from slow memory to fast memory. However, the promotion can fail when there is not enough space in fast memory. As a result, the hot page stays in slow memory, leading to performance loss. Linux and existing efforts (e.g., TPP and AutoTiering) proactively reclaim fast memory: when the free space in a fast memory node is lower than `low_watermark` (i.e., a threshold), the node is considered under memory pressure and page reclamation for that node is initiated until the free space is right above the watermark. The page reclamation is not in the critical path of page migration or allocation. However, when the memory node is under pressure, the freed memory space is constrained and up to `low_watermark`.

To avoid promotion failures, existing solutions, such as MEMTIS, limit the pages to promote within the available free space in fast memory without relying on page reclamation. This method, however, makes less effective use of fast memory. Other solutions, such as MTM [55] and Unimem [74], demote pages as demanded when the promotion failure is about to happen. This extends the critical path of page promotion, leading to higher page-migration overhead.

2.2 Memory Profiling

Technically, there are two common memory profiling methods in the state-of-the-art tiered memory systems. We discuss them as follows.

Triggering NUMA hinting faults. A kernel task routinely samples a range of a process's memory (256MB of pages by default [10]) on each NUMA node. To sample a page, a specific reserved bit (the `_PAGE_PROTNONE` bit) in the PTE is set. When a sampled page is accessed, a minor page fault (a.k.a. a NUMA hinting fault) is generated, and the reserved bit is reset. Therefore, by counting the number of NUMA hinting faults, we can estimate page access frequency. This method is traditionally used for NUMA balancing (AutoNUMA [10]), but used by multiple tiered memory systems (e.g., AutoTiering [22], Tiering-0.8 [71], and TPP [43]).

Using performance counters. Performance counters are special registers in the CPU that can be configured to count specific architecture events such as last-level cache (LLC) misses and branch mispredictions. If configured to use processor event-based sampling (PEBS), the processor can write an event record to a preallocated memory buffer after a performance counter overflows. The existing work commonly takes samples on retired LLC load misses and retired store instructions, which record the virtual memory address target for the sampled events. This method is used in recent work [14,26,29,34,51]. Using the performance counter-based method, there is usually *page metadata* associated with each page to record the number of accesses to that page.

The NUMA hinting faults-based method, relying on soft faults can be expensive. Because of the high performance overhead, this profiling method must limit the number of pages to be sampled (i.e., profiled) in a time interval. For example, in AutoNUMA, within a profiling time interval (1000 *ms*, by default), only 256MB of pages are sampled for profiling. The whole address space is segmented into a series of 256MB-sized memory regions, and these regions are profiled one by one. As a result, the pages chosen for sampling are not driven by the *previous* profiling results or page hotness.

In addition, the fault-based method does not use many records of memory accesses to decide the promotion of a page in slow memory. For example, AutoNUMA and TPP use an active list and an inactive list to organize pages. The pages on the active list are subject to promotion. To be on the active list, a page only needs to be accessed twice. Tracking such a small number of memory accesses is used to reduce performance overhead. However, this compromise can lead to substantial false positives in the detection of hot pages, i.e., a page is recognized as hot for promotion, but is not accessed often.

In contrast to the fault-based method, the performance counter-based method uses a large number of memory accesses to decide the promotion of a page. This is because of the relatively small overhead of using performance counters. For example, in MEMTIS, the profiled pages are organized into a 16-bin histogram based on page accesses. The n-th bin has the range of page accesses $[2^n, 2^{n+1})$. The pages falling into the high bins (e.g., the 15-th or 16-th bins) are subject to promotion. As a result, the emerging hot pages may take a while to be recognized by the performance counters as promotion candidates. This creates a profiling problem for those applications with time-changing memory access patterns. However, with performance counters, a page, once recognized as hot, is highly likely to be hot.

3 Motivation

We evaluate three state-of-the-art tiered memory systems: TPP, AutoNUMA, and MEMTIS. We use a server equipped with Optane and DRAM as slow and fast memories. Detailed hardware specifications can be found in Section 8.1. Table 1 summarizes the workloads we use for evaluation. We have the following observations that drive our design in FlexMem.

Observation 1: no single profiling method is a permanent winner. We compare the performance of TPP (using the fault-based profiling) and MEMTIS (using the performance counter-based profiling), as shown in Figure 1. We can see that no method is a permanent winner and different workloads require different profiling methods: in some cases (e.g., FT and SP), the fault-based profiling wins while in other cases (e.g., Btree, Silo, Graph500, and LU), the performance counter-based profiling wins.

To identify the root causes, we further collect the number of fast memory accesses [1], as shown in Figure 2. Comparing Figures 1 and 2, we find there is a clear positive association between the performance of profiling methods and the number of fast memory accesses, i.e., the larger the number of fast memory accesses, the better the performance of profiling methods. It motivates us to employ the fault-based profiling in conjunction with the performance counter-based profiling in an adaptive manner to enjoy the best of both worlds.

Observation 2: page promotion failures happen often. Figure 3 reports the rates of page promotion failures with TPP (Figure 3a) and AutoNUMA (Figure 3b) respectively. The failure rate is computed as the rate of the number of pages that need to be promoted but fail (due to the limited fast

[1] We exclude fast memory accesses due to page migration and only collect those from the workloads.

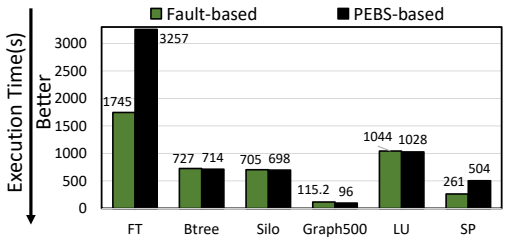


Figure 1: Performance comparison between the fault-based and performance counter-based (PEBS-based) profiling methods. The numbers on top of each bar are the performance in seconds.

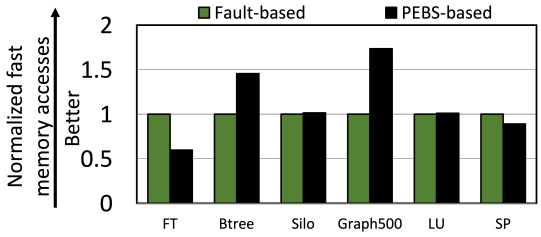
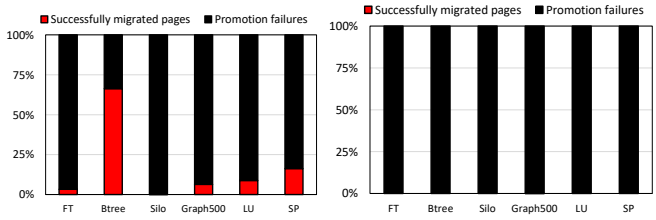


Figure 2: The number of fast memory accesses, excluding those from page migrations.

memory space) to the total number of pages promoted to fast memory successfully. We can see that TPP and AutoNUMA both suffer from significant page promotion failures, with an average 83% failure rate for TPP and an almost 100% failure rate for AutoNUMA. The reasons are as follows.

For TPP, it uses a threshold-based mechanism to proactively determine when page demotion should happen to reclaim fast memory (detailed in Section 2.1). This mechanism is constrained by the predefined threshold and thus may not reclaim sufficient space in fast memory for future page promotion. For AutoNUMA, it leverages NUMA balancing to allocate pages in fast memory, which does not provide a page demotion mechanism for CPU-less memory nodes (slow memory in our evaluation platform). This is because the page demotion based on NUMA balancing works only when memory accesses can be attributed to CPUs. TPP is better than AutoNUMA because TPP triggers demotion to proactively reclaim fast memory. We also evaluate MEMTIS with the same workloads (not shown in Figure 3). With MEMTIS, the rate of page promotion failures is less than 1%. However, it makes less effective use of fast memory, as discussed in Section 2.1.

Observation 3: warm-page set changes over time. We take Silo as an example to study the change of warm-page set. We use PEBS to collect page access frequency, based on which, we categorize pages into different bins. Bin_i has the range of page accesses $[2^n, 2^{n+1})$. For every 0.1-million sampled PEBS records, we update each bin size based on page accesses and for every two-million sampled PEBS records, we



(a) TPP (b) AutoNUMA

Figure 3: The rate of the number of pages failed to be promoted to the total number of pages promoted to fast memory successfully.

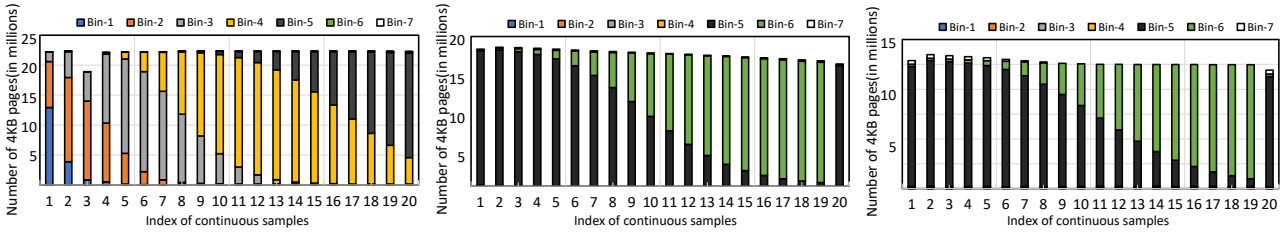
halve page accesses to decay the impact of old accesses and give more weight to recent accesses, which is aligned with MEMTIS. In MEMTIS, this is called "cooling operation". Figure 4 draws an overview of the change of bin sizes in three phases of Silo: beginning, middle, and end [2]. The three phases evenly divide the execution time of Silo. We make the following observations.

Beginning phase. Cold pages are getting accesses and these pages are moving towards hot bins. At $sample_1$ in Figure 4a, 95% of pages are located at bin_1 and bin_2. In this phase, the pages in low bins are quickly shifting-up to higher bins because of the accumulation of memory accesses. For bin_1, 96% of pages are promoted to higher bins (i.e., bin_2 and bin_3) within two sample intervals. However, for bin_2 and bin_3, at least 50% of pages are promoted to higher bins within four intervals. Compared to bin_1, bin_2, and bin_3, bin_4 needs a longer duration, spanning seven intervals, to ascend to higher bins. This extended interval is attributed to the additional time required for accumulating a sufficient number of memory accesses.

Middle and end phases. Figures 4b and 4c show the change of bin sizes in the middle and end phases. The hot page threshold is 7 (i.e., bin_7). This means all the pages above bin_7 are hot. All the pages below bin_7 are cold. However, We find that the majority of the pages in cold bins are moving between bin_5 and bin_6. In Figure 4b, pages in bin_5 shift to bin_6, shown from $sample_3$ to $sample_{19}$. At $sample_{20}$, we see that the size of bin_5 suddenly increases. That is because of the cooling operation, which makes the pages shifting to bin_6 fall back to bin_5. The page shifting indicates that pages in the cold bins are being frequently accessed by the application.

In conclusion, in the beginning phase, the warm pages are in bin_1 to bin_5, while in the middle and end phases, the warm pages are in bin_5 and bin_6. No single bin can effectively capture warm pages.

[2] Because of the page limitation, we only show 20 continuous samples for each phase.



(a) Begining phase (b) Middle phase (c) End phase

Figure 4: Bin sizes in three phases of Silo.

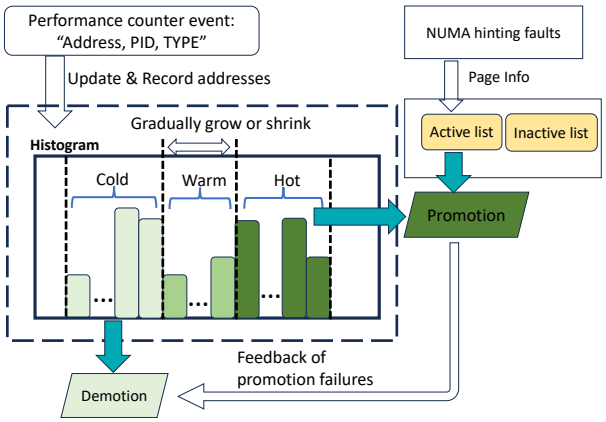


Figure 5: Overview of FlexMem.

4 Overview

FlexMem has three major designs.

(1) Adaptive profiling. This design combines the performance counter-based and fault-based profiling methods such that FlexMem can use the fault-based profiling method to opportunistically and timely identify emerging hot pages (solving the limitation of the performance counter-based method), and use the performance counter-based method to reduce the possibility of falsely identifying hot pages (solving the limitation of the fault-based method).

To coordinate the two profilers for page promotion, FlexMem suspends the irregular page promotions in the fault-based profiling method until the performance counter-based profiling method is ready to promote pages on a regular basis. To coordinate the two profilers for page demotion, a hot page identified by the fault-based profiling method, even though recognized as cold by the performance counter-based profiling method, is not demoted, until FlexMem finds that the page has few accesses in the next few time intervals. This method prevents unnecessary page demotion for emerging hot pages. FlexMem also enriches the histogram that tracks memory accesses across pages by adding page address information, which leads to simplification of data structure (i.e., the page promotion list). In addition, FlexMem couples the determination of the hot page threshold with page migration instead of decoupling them as in the existing work. This coupling method ensures the correct migration of hot and cold pages.

(2) Adaptive demotion. In FlexMem, the number of pages to demote is dynamically determined by the number of cold pages in fast memory and the number of pages to promote but fails. Promoting some pages identified by the fault-based profiling method may not be effective, when the pages do not become hot in the near future as expected. The page demotion considers this kind of ineffectiveness: FlexMem considers if the most recent page promotion from the fault-based profiling method leads to the discovery of emerging hot pages. FlexMem quantifies the successful discovery and dynamically changes the number of page demotions in proportion to the increase (or decrease) of successful discovery.

(3) Adaptive warm bins. FlexMem does not always use a single bin for warm pages (i.e., the warm bin) as the existing work. Instead, it dynamically extends the lower bound of warm bins towards the cold bins after the hot page threshold is selected. A cold bin can be counted as a warm bin when a certain percentage of pages in the cold bin were promoted to higher bins recently.

Figure 5 overviews the design of FlexMem.

5 Adaptive Profiling

The adaptive profiling method in FlexMem combines a performance counter-based profiler and a fault-based profiler.

Performance counter-based profiler is built upon the profiler in MEMTIS and MTM [55], but with significant extension to improve accuracy and enable easy interaction with the fault-based profiler.

Existing design in MEMTIS and MTM is employed in FlexMem, discussed as follows. FlexMem counts memory accesses by sampling LLC load misses and store instructions using PEBS. Each memory access is attributed to a page. The number of accesses to a page is accumulated using the expo-

nential moving average (EMA). In other words, assuming that the number of accesses to a page in the i^{th} profiling interval is x, and the EMA at the $(i-1)^{th}$ interval is EMA_{i-1}, then the EMA in the i^{th} interval is as follows.

$$EMA_i = k \times x + (1-k) \times EMA_{i-1} \quad (1)$$

where k is a parameter to balance the contribution of history profiling records and the current record.

Using EMAs collected for memory access samples, FlexMem builds a page access histogram. In particular, the histogram consists of 16 bins, and each bin has a range of EMA following an exponential scale. For example, n^{th} bin has the range of EMA $[2^n, 2^{n+1})$. The value of each bin is the number of distinct base pages in the EMA range. Using the histogram, we decide the *hot page threshold*. This threshold is a bin index, h. The accumulated size of those pages in the bin h and above is just smaller than the fast memory capacity. Such a threshold is periodically updated because the histogram is updated all the time. MEMTIS introduces a warm page threshold w and a cold page threshold c ($w = h-1$ and $c = w-1$).

As MEMTIS, FlexMem uses two kernel threads per memory node, `kimgrated` and `kprof`, to implement the performance counter-based profiler. When a page access is sampled, `kprof` updates the EMA of the page and updates the histogram. If the page becomes hot, the page is moved to a promotion list. `kmigrated` awakes periodically to examine the promotion list and move hot pages in slow memory to fast memory. When the free space in fast memory is below a threshold (2% of fast memory capacity), `kmigrated` performs page demotion. The cold pages (and warm pages if there is not enough space in fast memory) are demoted.

New design. Different from the existing work, the uniqueness of our performance counter-based profiler comes from two perspectives. (1) Couple the determination of the hot page threshold with page migration. In the existing work MEMTIS, there is no synchronization between the adaptation of the hot page threshold and page migration: the threshold adaptation happens every 100,000 sampled events and the page migration happens every 500ms. This creates a problem: some warm/cold pages become hot since the last threshold-adaption, but the page migration cannot timely migrate these pages because of using a stale hot-page threshold.

To address this problem, FlexMem couples the determination of the hot page threshold with page migration. In particular, whenever `kmigrated` awakes, it adapts the threshold using the most recent histogram, decides hot pages, and then migrates them. This method can reduce the delay of migrating recent hot pages.

(2) Embed page addresses into the histogram. Given the design (1), the design (2) is used to reduce the runtime overhead. The traditional histogram does not have information on page addresses: for each bin, the histogram only records the number of pages. Such a histogram is enough to determine the hot page threshold, and the page migration only needs the promotion list to determine which pages should be migrated. However, with the design (1), the hot page threshold is determined right before page migration. As a result, the promotion list must be re-examined to determine hot pages, which can be time-consuming.

To address this problem, FlexMem extends the histogram by adding page addresses. Each bin not only has the number of pages but also the addresses of those pages. The pages are classified into slow memory pages and fast memory pages. As a result, once the hot page threshold is determined, `kmigrated` can immediately know which pages should be migrated without using the promotion list. This also saves the overhead of running the promotion list.

Maintaining the new histogram is lightweight. When a page access is sampled, `kprof` updates the page EMA and the histogram as usual. However, the histogram update may include adding the page address into a bin when the page needs to be moved from one bin to another, which is lightweight.

Fault-based profiler uses Linux's active list and inactive list. The two lists are LRU-based and serve as a building block to track page hotness. A page, once found accessed by counting a page fault, is moved to the active list; once accessed again, it is immediately promoted from slow memory to fast memory and moved from the inactive list to the active list. The above design is also employed in AutoNUMA, TPP, and Tiering-0.8.

However, the above design creates difficulty in coordinating the two profilers. In particular, the fault-based profiler immediately promotes a page, once finding it hot. In contrast, the performance counter-based profiler promotes pages by `kmigrated` at a constant time interval. There is no synchronization between the two profilers. As a result, there is no good control over how the hot pages decided by the different profilers should be migrated to fast memory.

Coordination of the two profilers. To address the above problem, FlexMem does not immediately promote the hot page decided by the fault-based profiler. Instead, the page is only moved to the LRU-based active list without page promotion. When `kmigrated` promotes hot pages, it checks both the active list and histogram, and promotes pages in the active list and hot bins in the histogram. In the case that there is no enough free space in fast memory to host those pages, FlexMem chooses pages from the active list and hot bins in a round-robin fashion until reaching the maximum capacity of fast memory. Using this method, the active list and hot bins have equal opportunities to contribute hot pages.

Using the above method, there is no need to change the method of deciding the hot threshold in the performance counter-based profiler. FlexMem uses the hot threshold only for this profiler, and relies on the fault-based profiler to opportunistically find hot pages.

A hot page in the active list, once promoted, can be immediately subject for demotion by the histogram mechanism,

because this page may fall into a cold bin. To prevent this ping-pong effect, we add a countdown timer to the "hot page" in the cold bin. The countdown timer is initially set to 5, but counts down by one whenever this page is selected for demotion. This page is not demoted until the countdown timer reaches 0. The countdown timer is disabled when the page is promoted to a hot bin.

Using this countdown timer, the hot page identified by the fault-based profiler can stay in fast memory and be promoted to a hot bin, which means that the two profilers reach an agreement on the page hotness. In addition, if this hot page stays in the cold bin and is falsely detected by the fault-based profiler, the countdown timer allows us to demote the page and correct the page-promotion mistake.

Besides the above change to the page demotion in the performance counter-based profiler, we remove the page demotion mechanism in the fault-based profiler, because the performance counter-based profiler is more reliable to decide page demotion.

Thread management for the profilers. The two profilers share a kernel thread, `kmigrated`, for page migration. Besides this thread, the performance counter-based profiler has `kprof`, and the fault-based profiler has a page-fault handler to update the (in)active list. `kmigrated` is also used for adaptive demotion (Section 6) and `kprof` is also used to implement adaptive warm bins (Section 7).

Discussion on the profiling overhead. The profiling time overhead is small and remains at the same level as MEMTIS (i.e., 2% on a single CPU). This is because memory access tracking and page migration happen asynchronously in the background. In addition, compared with MEMTIS, FlexMem removes the promotion list, but adds an overhead of adding page addresses into the bins and add/update countdown timers in a few pages, which is small.

Time intervals. Like existing tiered memory systems, FlexMem has multiple time intervals. (1) The hot-page threshold adaptation interval is the same as the one in MEMTIS which takes a sample every 100K PEBS events; (2) there is no page migration interval. Once the hot page threshold is updated, FlexMem migrates pages (including page promotion and demotion); (3) the profiling interval (such as the time duration used to collect performance events in the performance counter-based profiler or the time duration used to scan a memory address range in the fault-based profiler) remains the same as the original profilers.

6 Adaptive Demotion

The demotion rate is defined as the number of pages to demote from fast memory to slow memory in a time interval (the hot-page threshold adaptation interval). Most existing solutions (e.g., TPP and MEMTIS) use a threshold to trigger page demotion: when the free fast-memory space is below a threshold (e.g., 2% of the fast memory size), the page demotion (page reclamation) happens.

FlexMem dynamically decides the demotion rate at runtime. The demotion rate (DR) is calculated in Equation 2. DR is controlled by ① the number of cold pages in the fast memory, ② the number of hot pages identified by the histogram ($promo\_fails_{histo}$) to promote but fails, and ③ the number of hot pages identified by the fault-based profiler to promote but fails ($promo\_fails_{pf}$).

$$DR = cold\_pages + promo\_fails_{histo} + \alpha \times promo\_fails_{pf} \quad (2)$$

The promotion failures in ② are caused by real hot pages identified by the history information. The promotion of those pages is necessary, and hence the demotion must reclaim enough pages to accommodate those hot pages. The promotion failures in ③ are caused by hot pages opportunistically identified (called *promising hot pages*). The promotion of the promising hot pages is subject to the effectiveness of promoting them. The effectiveness of promoting them is quantified by a variable, α. $\alpha \in [0,1]$. A larger α indicates that more promising hot pages have been correctly identified recently and FlexMem should demote more pages to allow the promotion of promising hot pages. Hence α can work as a metric to guide the promotion of promising hot pages (or page reclaim). We describe the above method in more detail as follows. Algorithm 1 depicts details.

Counting cold pages in fast memory. The histogram maintains page information for bins: for each bin, there are page addresses classified into slow memory pages and fast memory pages (see Section 5). FlexMem counts the number of pages in fast memory and stores them in an array BS_fc for all bins, shown in Algorithm 1.

Adaptive α. α is initially set as 1 to maximize the promotion of promising hot pages, and then dynamically changed according to how many promising hot pages are mistakenly promoted. We leverage the countdown timer mechanism to determine whether a promising hot page is mistakenly promoted: if the page is not promoted to any bin before the timer reaches 0, it is mistakenly promoted. We count such pages since the last page demotion.

The change of α is aligned with the change of the promotion mistakes. In particular, FlexMem records the number of mistakenly promoted pages (denoted by m), and the total number of promoted promising-hot-pages (denoted by tp). The new α is calculated as follows.

$$\alpha = 1 - m/tp \quad (3)$$

m/tp quantifies the degree of promotion mistakes. When m/tp is large, it means that the promoted pages are not actually hot. As a result, α should decrease to discourage the opportunistic promotion caused by the fault-based profiling.

Algorithm 1: Adaptive demotion rate.

BS_{fc} : *# cold pages in fast memory for each bin*;
N_{fc} : *total number of cold pages in fast memory*;
m : *# of mistakenly promoted pages*;
tp : *# promoted pages by fault−based profiling*;
DR : *demotion rate*;
T_{warm} : *the lower boundary of the warm bins*;

```
1  i ← 0;
2  N_fc ← 0;
   // calculate #cold pages in fast memory.
3  while i < T_warm do
4  |   N_fc ← N_fc + BS_fc[i];
5  |   i ← i + 1;
6  end
   // update α
7  α ← 1 − m/tp;
   // calculate the demotion rate.
8  DR ← N_fc + promo_fail_histo + α ∗ promo_fail_pf;
   // reset the statistics.
9  promo_fail_histo ← 0;
10 promo_fail_pf ← 0;
11 m ← 0;
12 tp ← 0
```

Algorithm 2: Adaptive warm bins

T_{hot} : *the current hotness threshold*;
T_{warm} : *the current lower bound of warm bins*;
BS : *the size of each bin*;
MS : *the number of shifting−up pages in each bin*;

```
1  w ← 0;
2  i ← T_hot − 1;
   // calculate the lower bound of warm bins.
3  while i >= 0 and MS[i] >= β ∗ BS[i] do
4  |   i = i − 1;
5  |   w = w + 1;
6  end
7  T_warm ← T_hot − w;
   // Reset the MS
8  for i ← 0 to b_hot − 1 do
9  |   MS[i] ← 0;
10 end
```

Overhead analysis. We allocate an array BS_{fc} to store the statistics for cold pages. This array only consumes 64 bytes (4 bytes for each bin). FlexMem only needs to run Algorithm 1 to determine DR, which is lightweight, because only T_{warm} of bins need to be examined to count cold pages (T_{warm} is the minimum index of warm bins), and T_{warm} is bounded by 16 (the total number of bins).

7 Adaptive Warm Bins

Definition of warm pages. The warm pages in FlexMem is a category of pages between "hot" and "cold" in terms of access frequency. The warm pages are about to become hot but can be unnecessarily demoted when page reclaim happens. Demoting warm pages that will become hot increases slow memory accesses and adds extra page migration when back-promotion.

MEMTIS designates the bin whose index is just one less than the minimum index of the hot bins (i.e., bin_{hot}) as the warm bin. All the other bins are either hot or cold. However, as illustrated in Figures 4a-4c, the range of warm bins can vary across execution phases.

Deciding the range of warm bins. FlexMem dynamically adjusts the lower bound of warm bins, and the upper bound is $bin_{hot} - 1$. The adjustment happens right after the adjustment of the hotness threshold bin_{hot}. The determination of hot pages is not impacted by the determination of warm bins.

FlexMem uses a greedy strategy to decide the lower bound, shown in Algorithm 2. The algorithm expands the warm page range from $T_{hot} - 1$ as far as possible under certain conditions. The condition to count a bin as a warm bin is whether there are many pages in that bin shifting to higher bins. If yes, that indicates that the bin is becoming hot and should not be used for page demotion. We introduce a parameter β as a threshold to determine if a bin is warm. β is the ratio of the number of shifting-up pages to the total number of pages in a bin. A larger β leads to a smaller range of warm bins, and vice versa.

Algorithm 2 gives more details. The algorithm takes four variables as input: the current hotness threshold, the variable for recording the upper bound of warm bins, the number of pages in each bin, and the number of shifting-up pages in each bin. At Lines 3-6, the algorithm attempts to find a bin that meets the warm-bin condition. Line 7 updates the lower bound of warm bins.

Building *BS* and *MS*. Algorithm 2 uses two arrays (*BS* and *MS*) to record the number of pages in each bin, and the number of shift-up pages in each bin within a timer interval (the hot-page threshold adaptation interval). Whenever there is a PEBS performance event, the two arrays are updated accordingly.

Overhead analysis. We allocate an extra array *BS* and *MS* to record the moving page size and the size of each bin. These two arrays only consume 128 bytes (8 bytes for each bin). FlexMem runs Algorithm 2 after changing the hotness threshold. The algorithm needs to check at most T_{hot} bins to determine the lower bound of warm bins and T_{hot} is bounded by 16 (the total number of bins), which is lightweight.

8 Evaluation

We implement FlexMem [3] in Linux Kernel v5.15. We extend the implementation of MEMTIS: we maintain a page-address list for each bin in the histogram using page frame number (PFN) [32]; we record the number of page promotion failures and the number of pages moved between bins, and feed these numbers to the adaptive demotion and adaptive warm bin algorithms; we also change the demotion mechanism to demote cold pages till the warm bins.

We evaluate FlexMem by answering the following questions:

- How does FlexMem perform with memory-intensive applications (such as HPC workloads and in-memory database engines), compared with the state-of-the-art memory tiering solutions?
- Can FlexMem use fast memory effectively?
- Can FlexMem address the existing memory management problems effectively?

8.1 Evaluation Methodology

Evaluation platform. We evaluate FlexMem on a dual-socket server equipped with Intel Xeon Gold 6252 @2.10 GHz processors (24 cores per socket), where each socket has 6×16GB DDR4 DRAM (i.e., fast memory), and 6×128GB Intel Optane DCPMM (i.e., slow memory). Similar to prior works [26, 43], we use a single socket for our evaluation to avoid NUMA effects. We use Intel Memory Latency Checker [11] to get the memory acccess latency of DRAM(98ns) and Optane memory(348ns).

Benchmarks. We use six representative memory-intensive applications, including a graph processing benchmark (Graph500) [45], an in-memory database engine (Silo) [70], an in-memory index lookup benchmark (Btree) [58], and three NAS parallel benchmarks (HPC workloads) [3, 46]: discrete 3D fast Fourier Transform (FT), Lower-Upper Gauss-Seidel solver (LU), and Scalar Penta-diaagonal solver (SP). Table 1 details these benchmarks.

Comparison targets. We compare FlexMem to three state-of-the-art tiered memory systems: TPP, MEMTIS, and Tiering-0.8. We enable transparent huge page (THP) for huge page allocation. We run each benchmark with 24 threads and report its relative performance normalized to the performance of running it entirely in slow memory (i.e., the all-NVM case).

Memory tiering configurations. Due to the relatively small working set size of the benchmarks for evaluation, we reduce the fast memory size to 30GB to evaluate the effectiveness of page migration. More specifically, for MEMTIS, we use `cgroups` to limit the free fast memory size, and for TPP and Tiering-0.8, we use the `memmap` kernel parameter to limit the free fast memory size. For other configurations, we use the default values in MEMTIS and TPP. For Tiering-0.8, we use the recommended values.

[3] https://github.com/PASAUCMerced/FlexMem

Table 1: Benchmarks for evaluation.

Benchmark	Description	Working set size
FT	Discrete 3D fast Fourier Transform.	80.4GB
Graph500	Generation and search of large graphs	120GB
Btree	In-memory index lookup benchmark	64GB
Silo	In-memory database engine	89.5GB
LU	Lower-Upper Gauss-Seidel solver.	134GB
SP	Scalar Penta-diagonal solver	80GB

8.2 Overall Performance

Figure 6 shows the performance improvement of Tiering-0.8, TPP, MEMTIS, and FlexMem over all-NVM. FlexMem performs best in all the benchmarks, and outperforms Tiering-0.8, TPP, and MEMTIS by 32%, 23% and 27% on average respectively. Tiering-0.8 performs worst, because it sets an upper bound on the promotion and demotion rates to limit the utilization of memory bandwidth and the promotion of hot pages. We build a heatmap for each benchmark, as shown in Figure 7, to visualize the change of page hotness during program execution. We also collect the numbers of fast memory and slow memory accesses using performance counters and calculate their ratio, as shown in Figure 9, to quantify how effectively the benchmarks use fast memory after page migration.

8.2.1 NPB benchmarks: FT, SP and LU

FT. FlexMem outperforms TPP and MEMTIS by 32% and 57% respectively. Figure 7a shows its heatmap.

In the beginning phase, the entire address space in FT is evenly accessed. Hence, MEMTIS successfully marks hot pages. After a short while, however, most memory accesses are attributed to the first half of the address space, and the hot page set keeps changing over time. In this case, MEMTIS fails to timely and accurately detect and promote emerging hot pages. Because those "old" hot pages with low access recency remain in fast memory due to their higher cumulative access frequency than the emerging hot pages. This is a fundamental drawback of performance counter-based profiling. Unlike MEMTIS, TPP and FlexMem employ fault-based profiling to quickly promote the emerging hot pages and thus deliver better performance.

FlexMem outperforms TPP because of adaptive demotion. FlexMem demotes cold pages more quickly for the sake of better use of fast memory. This conclusion is backed up by Figure 9. The figure shows the ratio of fast memory accesses to slow memory accesses. For FT, the ratio is 2.17 and 3.75 for TPP and FlexMem respectively, indicating FlexMem makes a better use of fast memory.

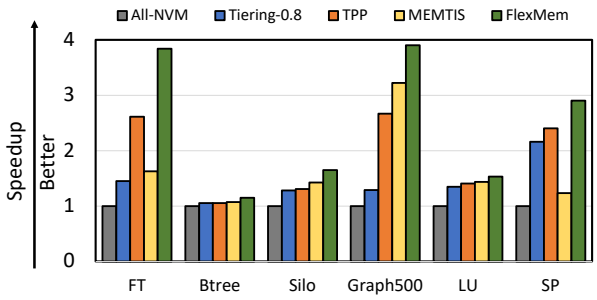


Figure 6: Performance improvement over all-NVM.

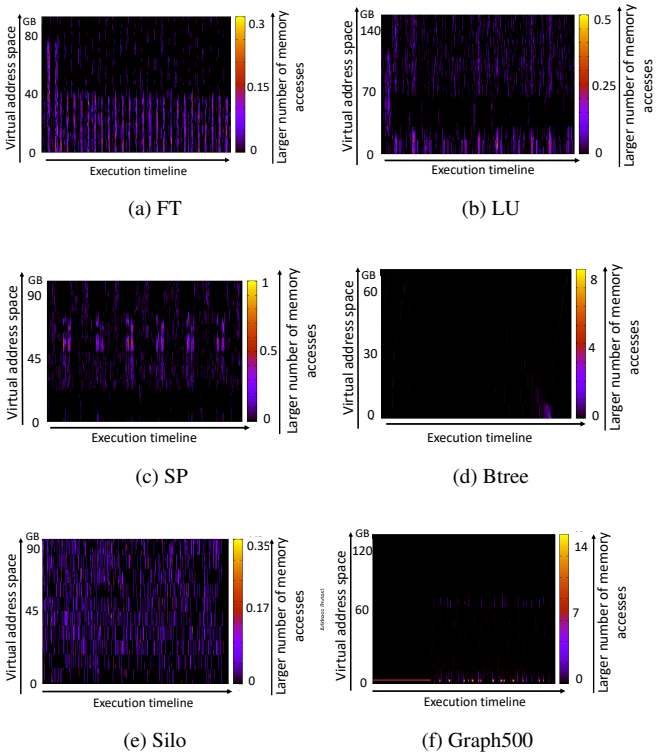


Figure 7: Memory access patterns of different benchmarks.

LU and SP. FlexMem outperforms TPP by 8% and 27% and MEMTIS by 6% and 57% for LU and SP respectively. Figures 7b and 7c show their respective heatmaps. We can see that the hot page set changes over time although a small number of pages periodically become hot. Therefore, timely and accurately capturing the time-changing hot page set is the key to success, which TPP and MEMTIS lack. Figure 9 further backs up our conclusion: FlexMem enjoys a higher ratio (52%) of fast memory to slow memory accesses than TPP (46%) and MEMTIS (36%).

8.2.2 Graph Processing: Graph500

Graph500 uses a Breadth-First Search (BFS) algorithm to search for 64 keys within a generated graph. The default generator in Graph500 is a stochastic Kronecker graph generator [28]. It is a power-law graph, where a small number of nodes have the majority of edges (i.e., these nodes require multiple pages to store their neighbors). During BFS, the pages containing neighbor information are frequently accessed.

FlexMem outperforms TPP and MEMTIS by 32% and 17% respectively. TPP performs the worst because it demotes pages without considering their future accesses. We manually examine its page demotion and find that 25% pages containing neighbor information are demoted and then promoted again within a mere two migration intervals, indicating many unnecessary page migrations. Different from TPP, MEMTIS categorizes old hot pages as warm to avoid unnecessary page migration and hence performs better. However, MEMTIS only considers the pages in a single bin as warm, diminishing the effectiveness of the warm bin mechanism. FlexMem circumvents the above limitations by adopting adaptive demotion and multiple warm bins.

8.2.3 In-Memory Database Engine: Silo

We use YCSB-C [9] as the substrate to benchmark Silo. YCSB-C, a minimal C++ version of YCSB, performs 300 million lookup operations per thread. FlexMem outperforms TPP and MEMTIS by 26% and 13% respectively.

FlexMem outperforms MEMTIS because of its adaptive warm-bin mechanism. From Figure 4, we can see that memory pages in Silo get hot progressively instead of abruptly. Therefore circumventing the demotion of warm pages that are getting hot is critical, which MEMTIS's single warm-bin mechanism fails to address. Figure 10 further backs up our conclusion. We can see that MEMTIS increases the number of page migrations (including page demotion and promotion) by 56% compared to FlexMem. Because MEMTIS mistakenly demotes warm pages and then promotes them soon. FlexMem outperforms TPP because of its higher accuracy in detecting hot pages.

8.2.4 In-Memory Index Lookup Benchmark: Btree

We populate Btree with 315-million key-value pairs and perform 20-billion random lookup operations. Figure 7d shows its heatmap. FlexMem outperforms TPP and MEMTIS by 9% and 6% respectively. Further investigation reveals that the memory accesses in Btree are scattered all over the program's address space. As a result, the performance counter-based MEMTIS needs a longer duration to identify hot pages and the fault-based TPP is more likely to mistakenly identify hot pages.

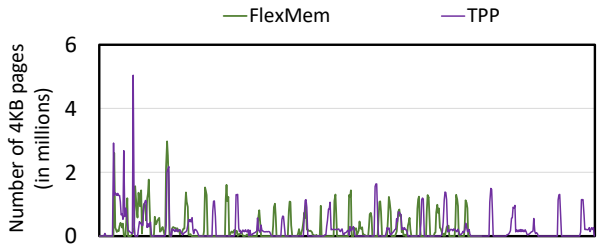


Figure 8: The number of migration failures in FT.

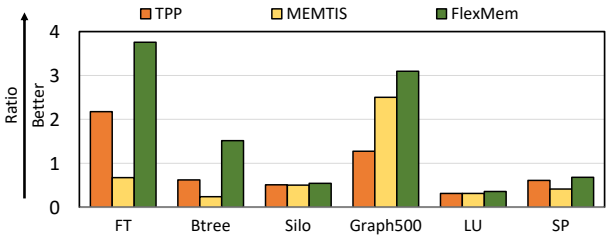


Figure 9: Ratio of the fast memory accesses to slow memory accesses.

8.3 Analysis on Page Management

We perform deeper analysis to evaluate FlexMem.

Effectiveness of adaptive demotion. Figure 8 shows the number of promotion failures at 5-second time intervals when we run FT with TPP and FlexMem. We exclude MEMTIS because it has few promotion failures. The figure shows that compared to TPP, FlexMem cuts promotion failures by 26% thanks to the adaptive demotion. For all the benchmarks, FlexMem reduces the promotion failures by 18% on average.

Fast memory and slow memory accesses. We use the Linux perf tool [33] to collect fast memory and slow memory accesses and present their ratio in Figure 9. The figure reveals that for all the benchmarks, FlexMem has a higher ratio than TPP and MEMTIS, demonstrating that FlexMem makes better use of fast memory for high performance.

Effectiveness of adaptive warm bins. Figure 11a and 11b show the number of warm pages in Silo and Graph500 identified by FlexMem and MEMTIS respectively. We can see that FlexMem identifies more pages as warm, leading to more fast memory accesses (Figure 9) and fewer page migrations (Figure 10). We observe similar results in the other benchmarks.

Migration volume. Figure 10 shows the page migration volume in terms of the number of 4KB pages. For Btree, Silo, and Graph500, FlexMem has (nearly) the lowest migration volume because it uses multiple bins for warm pages, avoiding unnecessary migration of warm pages. For FT, LU, and SP, MEMTIS' migration volume is 57% and 51% lower than TPP and FlexMem respectively. This is because of MEMTIS' inability to timely promote hot pages (due to the slow accumulation of PEBS records) and demote cold pages (due to the single warm-bin mechanism).

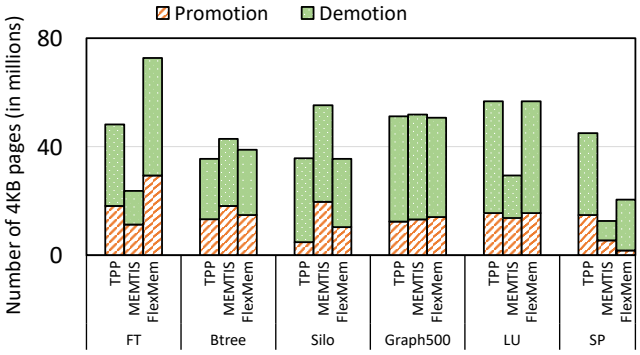
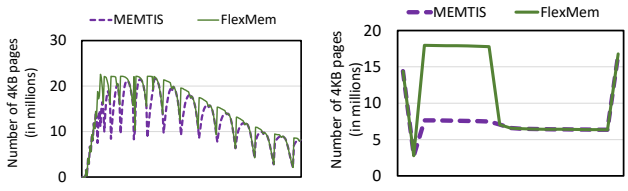


Figure 10: Page migration volume.



(a) Silo (b) Graph500

Figure 11: The number of warm pages identified by MEMTIS and FlexMem. The workloads are Silo and Graph500.

8.4 Sensitivity Analysis

α variance. Figure 12 shows the variance of α at runtime. We use FT as an example to demonstrate the adaptiveness of α. α continuously changes between 0 and 1. At some time, the value drops from 1 to 0 or goes up to 1 from 0 because m is close to tp and vice versa (see Equation 3).

Sensitivity to the setting of β. Figure 13 shows the sensitivity of Silo and Graph500 to β. We exclude the other benchmarks because their performance is insensitive to the adaptive warm bins. We change β from 10% to 40% and observe that the default β value enjoys the best performance in all cases. When β is smaller than the default value, it becomes difficult to add more warm bins, leading to more unnecessary page demotion.

Sensitivity to the fast memory size. Figure 14 shows the performance variance of Silo under different fast memory sizes. We use Silo as an example, but the other benchmarks show a similar trend. The figure shows that FlexMem consistently outperforms TPP and MEMTIS. With the increase of the fast memory size, however, the performance difference between them becomes smaller. This is because the larger the fast memory is, the less performance improvement the page migration delivers.

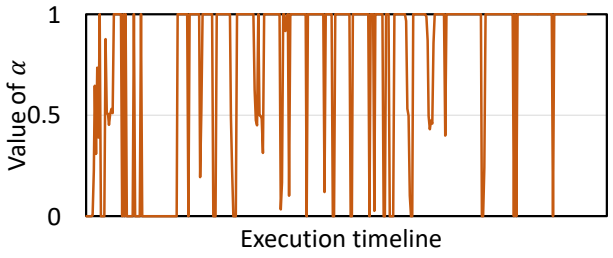


Figure 12: The variance of α value along with the workload execution. X-axis is the execution timeline. The workload is FT.

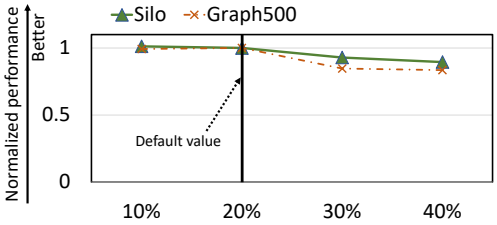


Figure 13: Performance variance when changing β. The performance is normalized by the performance of the default β value.

8.5 Overhead Analysis

Maintaining histogram. FlexMem maintains a page list for each bin. The page list is a collection of physical PFN. Recording PFNs incurs at most 0.19% memory overhead in our evaluation. The histogram is maintained by the `kprof` kernel thread and the runtime overhead is less than 1% in our evaluation.

Maintaining the page list. We use PFN(page frame number) as the element in the list. There are only adding and deleting operations in a linked list. Only when a page's accumulated access exceeds the range of the current bin, an adding operation will happen. Only when the page list is going to be migrated, FlexMem will delete the pages from the head of the list. Both operations have O(1) time complexity.

9 Related Work

Page management in tiered memory. Recent works [2, 12, 16–18, 26, 29, 35, 36, 39, 42, 43, 51–55, 57, 60, 71, 74–80, 82–84] explored the design of tiered memory systems. AutoNUMA [39] is designed to improve the performance of applications running on NUMA hardware systems. Tiering-0.8 [71], MTM [55] and Nimble [82] target multi-tier memory systems. Tiering-0.8 leverages runtime memory-bandwidth usage to decide migration volume. Nimble applies multi-threading to accelerate huge page migration. HeMem [51], TPP [43], MEMTIS [26], Pond [29], TMTS [15], MTM [55] and Thermostat [2] are designed for two-tiered memory systems. HeMem uses performance counters and an experience-based hotness threshold to identify hot pages. Thermostat uses random poisoning to identify hot huge pages. TPP sets a watermark to trigger page demotion. MEMTIS proposes a dynamic hotness threshold based on runtime profiling. Pond creates machine learning models that can accurately predict how much local and pool memory to allocate to a virtual machine. TMTS implements an adaptive, hardware-guided architecture to dynamically optimize access to the various directly-addressed memory tiers without faults. MTM [55] uses PEBS to profile the hot pages and reduce the memory profiling overhead.

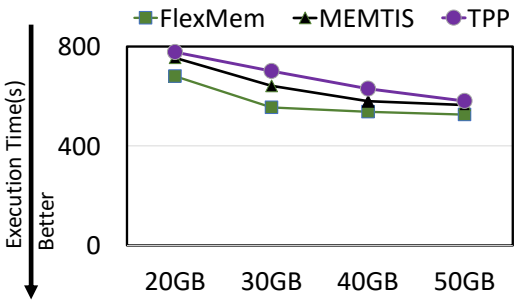


Figure 14: Performance variance when changing the fast memory size. The results are for Silo.

Memory profiling. Memory profiling plays a key role in measuring, analyzing, and optimizing various forms of inefficiencies in memory subsystems, including high memory latency [5, 30, 37, 66], redundant memory accesses [48, 63–65, 73], poor locality of reference [38, 59, 86], memory leaks [4, 47], to name a few. Technically, memory profiling can be classified into hardware-assisted and software-based approaches. The former leverages performance counters available in almost (if not) all CPU processors to collect memory metrics (e.g., access frequencies of individual pages) at a low cost (typically less than 5% time and memory overhead). In contrast, the latter leverages source, intermediate, or binary code instrumentation to collect memory metrics from program execution at a high cost (typically greater than 10× time and memory overhead), thus significantly perturbing program behavior. As an online approach, FlexMem uses performance counters in conjunction with NUMA hinting faults to perform non-intrusive, lightweight profiling so as to minimize interference with the monitored program.

10 Conclusions

Tiered memory system, combining fast memory and slow memory components, is promising, but brings challenges to page profiling and migration to decide page allocation and placement on the memory components in the tiered memory. Although there are many progresses in the system software to

manage the tiered memory, their designs often lack flexibility to capture changing memory access patterns and make the best use of fast memory. In this paper, we introduce FlexMem to address the above limitation. By adapting the system designs based on the performance feedback of page demotion rate and warm bins, and combining two profiling methods to improve profiling quality, FlexMem outperforms the state-of-the-art by at least 20% on average.

Acknowledgments

This paper is a result of a research project sponsored by SK hynix Inc. This project was partly supported by the NSF grants 2104116 and 2316202. We would like to thank the anonymous reviewers for their feedback on the paper.

References

[1] Compute Express Link Industry Members. https://www.computeexpresslink.org/members, 2021.

[2] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, 2017.

[3] D.H. Bailey, E. Barszcz, L. Dagum, and H.D. Simon. Nas parallel benchmark results. In *Supercomputing '92:Proceedings of the 1992 ACM/IEEE Conference on Supercomputing*, pages 386–393, 1992.

[4] Derek Bruening and Qin Zhao. Practical memory checking with dr. memory. In *International Symposium on Code Generation and Optimization (CGO 2011)*, pages 213–223, 2011.

[5] Milind Chabbi, Shasha Wen, and Xu Liu. Featherlight on-the-fly false-sharing detection. *SIGPLAN Not.*, 53(1):152–167, feb 2018.

[6] Rong Chen, Jiaxin Shi, Yanzhe Chen, Binyu Zang, Haibing Guan, and Haibo Chen. Powerlyra: Differentiated graph computation and partitioning on skewed graphs. *ACM Transactions on Parallel Computing (TOPC)*, 5(3):1–39, 2019.

[7] Zheng Chen, Feng Zhang, JiaWei Guan, Jidong Zhai, Xipeng Shen, Huanchen Zhang, Wentong Shu, and Xiaoyong Du. Compressgraph: Efficient parallel graph analytics with rule-based compression. *Proceedings of the ACM on Management of Data*, 1(1):1–31, 2023.

[8] Claudio Conti. *Quantum Machine Learning: Thinking and Exploration in Neural Network Models for Quantum Science and Quantum Computing*. Springer Nature, 2024.

[9] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, 2010.

[10] J. Corbet. AutoNUMA: the Other Approach to NUMA Scheduling. http://lwn.net/Articles/488709.

[11] Intel Corporation. Intel® memory latency checker v3.10. 2023.

[12] Dong Xu, Yuan Feng, Kwangsik Shin, Daewoo Kim, Hyeran Jeon, and Dong Li. Efficient tensor offloading for large deep-learning model training based on compute express link., 2024.

[13] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeffrey R. Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. *Proceedings of the Eleventh European Conference on Computer Systems*, 2016.

[14] 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 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.

[15] 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 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, ASPLOS 2023, page 727–741, New York, NY, USA, 2023. Association for Computing Machinery.

[16] Yuan Feng, Hyeran Jeon, Filip Blagojevic, Cyril Guyot, Qing Li, and Dong Li. Attmemo : Accelerating transformers with memoization on big memory systems, 2023.

[17] 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.

[18] Yingchao Huang and Dong Li. Performance Modeling for Optimal Data Placement on GPU with Heterogeneous Memory Systems. In *IEEE International Conference on Cluster Computing*, 2017.

[19] Intel. Intel Optane Memory. https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-memory.html.

[20] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. Heteroos — os design for heterogeneous memory management in datacenter. *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 521–534, 2017.

[21] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. Klocs: kernel-level object contexts for heterogeneous memory systems. *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[22] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. Exploring the design space of page management for multi-tiered memory systems. In *USENIX Annual Technical Conference*, 2021.

[23] Kyung Duk Kim, Hyunseok Kim, Jinin So, Wonjae Lee, Jun-Hyeok Im, Sung-Rok YoonandSin-Chong Park, Jeonghyeon Cho, and Ho Uk Song. Smt: Software-defined memory tiering for heterogeneous computing systems with cxl memory expander. *IEEE Micro*, 43:20–29, 2023.

[24] Tirthankar Lahiri, Marie-Anne Neimat, and Steve Folkman. Oracle timesten: An in-memory database for enterprise applications. *IEEE Data Eng. Bull.*, 36(2):6–13, 2013.

[25] Sangjin Lee, Alberto Lerner, Philippe Bonnet, and Philippe Cudré-Mauroux. Database kernels: Seamless integration of database systems and fast storage via cxl. In *14th Conference on Innovative Data Systems Research, CIDR*, pages 9–12, 2024.

[26] 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 29th Symposium on Operating Systems Principles*, SOSP '23, page 17–34, New York, NY, USA, 2023. Association for Computing Machinery.

[27] Alberto Lerner and Gustavo Alonso. Cxl and the return of scale-up database engines. *arXiv preprint arXiv:2401.01150*, 2024.

[28] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *Journal of Machine Learning Research*, 11(33):985–1042, 2010.

[29] Huaicheng Li, Daniel S Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, 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*, pages 574–587, 2023.

[30] Pengcheng Li, Yixin Guo, Yingwei Luo, Xiaolin Wang, Zhenlin Wang, and Xu Liu. Graph neural networks based memory inefficiency detection using selective sampling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22. IEEE Press, 2022.

[31] Jan Lindström, Vilho Raatikka, Jarmo Ruuth, Petri Soini, and Katriina Vakkila. Ibm soliddb: In-memory database optimized for extreme speed and availability. *IEEE Data Eng. Bull.*, 36(2):14–20, 2013.

[32] Linux kernel labs. Memory mapping. "https://linux-kernel-labs.github.io/refs/heads/master/labs/memory_mapping.html.

[33] Linux tool. https://en.wikipedia.org/wiki/Perf_(Linux).

[34] David Lion, Adrian Chiu, and Ding Yuan. M3: End-to-end memory management in elastic system software stacks. In *Proceedings of the Sixteenth European Conference on Computer Systems*, EuroSys '21, page 507–522, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Jiawen Liu, Dong Li, and Jiajia Li. Athena: High-Performance Sparse Tensor Contraction Sequence on Heterogeneous Memory. In *International Conference on Supercomputing (ICS)*, 2021.

[36] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. Sparta: high-performance, element-wise sparse tensor contraction on heterogeneous memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 318–333, New York, NY, USA, 2021. Association for Computing Machinery.

[37] Tongping Liu and Xu Liu. Cheetah: Detecting false sharing efficiently and effectively. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 1–11, New York, NY, USA, 2016. Association for Computing Machinery.

[38] Xu Liu and John Mellor-Crummey. Pinpointing data locality problems using data-centric analysis. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '11, page 171–180, USA, 2011. IEEE Computer Society.

[39] LWN.net. Autonuma balancing. "https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_tuning_and_optimization_guide/sect-virtualization_tuning_optimization_guide-numa-auto_numa_balancing".

[40] Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. Learning-based memory allocation for c++ server workloads. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 541–556, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Luca Magri, Peter J Schmid, and Jonas P Moeck. Linear flow analysis inspired by mathematical methods from quantum mechanics. *Annual Review of Fluid Mechanics*, 55:541–574, 2023.

[42] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniela Campello, Andy Rudoff, and Raju Rangaswami. Multi-clock: Dynamic tiering for hybrid memory systems. *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 925–937, 2022.

[43] 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*, ASPLOS 2023, page 742–755, New York, NY, USA, 2023. Association for Computing Machinery.

[44] Agarwal Megha, Qureshi Asfandyar, Sardana Nikhil, Li Linden, Quevedo Julian, and Khudia Daya. Llm inference performance engineering: Best practices. Available at https://www.databricks.com/blog/ (2023/10/14).

[45] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray Users Group (CUG)*, 19:45–74, 2010.

[46] NASA Advanced Supercomputing (NAS) Division. https://www.nas.nasa.gov/software/npb.html.

[47] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *SIGPLAN Not.*, 42(6):89–100, jun 2007.

[48] Adrian Nistor, Linhai Song, Darko Marinov, and Shan Lu. Toddler: Detecting performance problems via similar memory-access patterns. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, page 562–571. IEEE Press, 2013.

[49] Ashish Panwar, Sorav Bansal, and K Gopinath. Hawkeye: Efficient fine-grained os support for huge pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 347–360, 2019.

[50] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making huge pages actually useful. *SIGPLAN Not.*, 53(2):679–692, mar 2018.

[51] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. Hemem: Scalable tiered memory management for big data applications and real nvm. *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021.

[52] Jie Ren, Jiaolin Luo, Ivy Peng, Kai Wu, and Dong Li. Optimizing large-scale plasma simulations on persistent memory-based heterogeneous memory with effective data placement across memory hierarchy. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 203–214, New York, NY, USA, 2021. Association for Computing Machinery.

[53] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, and Hyeran Jeon. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 598–611, 2021.

[54] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. {ZeRO-Offload}: Democratizing {Billion-Scale} model training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, pages 551–564, 2021.

[55] 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 Nineteenth European Conference on Computer Systems*, EuroSys '24, page 803–817, New York, NY, USA, 2024. Association for Computing Machinery.

[56] Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory Systems. In *European Conference on Computer Systems*, 2024.

[57] Jie Ren, Minjia Zhang, and Dong Li. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Conference on Neural Information Processing Systems (NeurIPS)*, 2020.

[58] Reto Achermann and Ashish Panwar. 2019. mitosis workload btree.

[59] Probir Roy and Xu Liu. Structslim: A lightweight profiler to guide structure splitting. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, page 36–46, New York, NY, USA, 2016. Association for Computing Machinery.

[60] Jee Ho Ryoo, Lizy K. John, and Arkaprava Basu. A case for granularity aware page migration. In *Proceedings of the 2018 International Conference on Supercomputing*, ICS '18, page 352–362, New York, NY, USA, 2018. Association for Computing Machinery.

[61] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[62] Joonseop Sim, Soohong Ahn, Taeyoung Ahn, Seungyong Lee, Myunghyun Rhee, Jooyoung Kim, Kwangsik Shin, Donguk Moon, Euiseok Kim, and Kyoung Park. Computational cxl-memory solution for accelerating memory-intensive applications. *IEEE Computer Architecture Letters*, 22:5–8, 2023.

[63] Linhai Song and Shan Lu. Performance diagnosis for inefficient loops. In *Proceedings of the 39th International Conference on Software Engineering*, ICSE '17, page 370–380. IEEE Press, 2017.

[64] Pengfei Su, Qingsen Wang, Milind Chabbi, and Xu Liu. Pinpointing performance inefficiencies in java. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 818–829, New York, NY, USA, 2019. Association for Computing Machinery.

[65] Pengfei Su, Shasha Wen, Hailong Yang, Milind Chabbi, and Xu Liu. Redundant loads: A software inefficiency indicator. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 982–993. IEEE Press, 2019.

[66] Nathan R. Tallent, John M. Mellor-Crummey, and Michael W. Fagan. Binary analysis for measurement and attribution of program performance. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, page 441–452, New York, NY, USA, 2009. Association for Computing Machinery.

[67] Ali TehraniJamsaz, Alok Mishra, Akash Dutta, Abid M Malik, Barbara Chapman, and Ali Jannesari. Paragraph: Weighted graph representation for performance optimization of hpc kernels. *arXiv preprint arXiv:2304.03487*, 2023.

[68] The Next Platform.CXL and Gen-Z Iron Out a Coherent Inter- connect Strategy. https://www.nextplatform.com/2020/04/03/cxl-and-gen-z-iron-out-a-coherent-interconnect-strategy/.

[69] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *ArXiv*, abs/2302.13971, 2023.

[70] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 18–32, New York, NY, USA, 2013. Association for Computing Machinery.

[71] Vishal Verma. Tiering-0.8. 2022. https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git/log/?h=tiering-0.8.

[72] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, M. Jain, Chunqiang Tang, and Dimitrios Skarlatos. Tmo: transparent memory offloading in datacenters. *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022.

[73] Shasha Wen, Milind Chabbi, and Xu Liu. Redspy: Exploring value locality in software. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, page 47–61, New York, NY, USA, 2017. Association for Computing Machinery.

[74] Kai Wu, Yingchao Huang, and Dong Li. Unimem: runtime data managementon non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '17, New York, NY, USA, 2017. Association for Computing Machinery.

[75] Kai Wu, Ivy B. Peng, Jie Ren, and Dong Li. Ribbon: High Performance Cache Line Flushing for Persistent Memory. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2020.

[76] Kai Wu, Jie Ren Ivy Peng, and Dong Li. ArchTM: Architecture-Aware, High Performance Transaction for Persistent Memory. In *USENIX Conference on File and Storage Technologies*, 2021.

[77] Kai Wu, Jie Ren, and Dong Li. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 401–413, 2018.

[78] Panruo Wu, Dong Li, Zizhong Chen, Jeffrey Vetter, and Sparsh Mittal. Algorithm-directed data placement in explicitly managed non-volatile memory. In *ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)*, 2016.

[79] Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. Md-hm: memoization-based molecular dynamics simulations on big memory system. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 215–226, New York, NY, USA, 2021. Association for Computing Machinery.

[80] Zhen Xie, Jie Liu, Jiajia Li, and Dong Li. Merchandiser: Data placement on heterogeneous memory for task-parallel hpc applications with load-balance awareness. In *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, page 204–217, New York, NY, USA, 2023. Association for Computing Machinery.

[81] Zihui Xue, Yuedong Yang, and Radu Marculescu. Sugar: Efficient subgraph-level training via resource-aware graph partitioning. *IEEE Transactions on Computers*, 2023.

[82] Zi Yan, Daniel Lustig, David W. Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[83] Shuangyan Yang, Minjia Zhang, Wenqian Dong, and Dong Li. Betty: Enabling large-scale gnn training with batch-level graph partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 103–117, New York, NY, USA, 2023. Association for Computing Machinery.

[84] Shuo Yang, Kai Wu, Yifan Qiao, Dong Li, and Jidong Zhai. Algorithm-Directed Crash Consistence in Non-Volatile Memory for HPC. In *IEEE International Conference on Cluster Computing*, 2017.

[85] Yiwei Yang, Pooneh Safayenikoo, Jiacheng Ma, Tanvir Ahmed Khan, and Andrew Quinn. Cxlmemsim: A pure software simulated cxl. mem for performance characterization. *arXiv preprint arXiv:2303.06153*, 2023.

[86] Xin Zhao, Jin Zhou, Hui Guan, Wei Wang, Xu Liu, and Tongping Liu. Numaperf: Predictive numa profiling. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '21, page 52–62, New York, NY, USA, 2021. Association for Computing Machinery.

[87] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph: Large-Scale graph processing on a single machine using 2-level hierarchical partitioning. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 375–386, Santa Clara, CA, July 2015. USENIX Association.