MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory

Jie Ren
jren03@wm.edu
William & Mary

Dong Xu
dxu17@ucmerced.edu
University of California, Merced

Junhee Ryu
junhee.ryu@sk.com
SK hynix

Kwangsik Shin
kwangsik.shin@sk.com
SK hynix

Daewoo Kim
daewoo0220.kim@sk.com
SK hynix

Dong Li
dli35@ucmerced.edu
University of California, Merced

Abstract

Multi-terabyte large memory systems are often characterized by more than two memory tiers with different latency and bandwidth. Multi-tiered large memory systems call for rethinking of memory profiling and migration because of the unique problems unseen in the traditional memory systems with smaller capacity and fewer tiers. We develop **MTM**, an application-transparent **M**ulti-**T**iered **M**emory management framework, based on three principles: (1) connecting the control of profiling overhead with the profiling mechanism for high-quality profiling; (2) building a universal page migration policy on the complex multi-tiered memory for high performance; and (3) introducing huge page awareness. We evaluate MTM using common big-data applications with realistic working sets (hundreds of GB to 1 TB). MTM outperforms seven solutions by up to 42% (17% on average).

ACM Reference Format:
Jie Ren, Dong Xu, Junhee Ryu, Kwangsik Shin, Daewoo Kim, and Dong Li. 2024. MTM: Rethinking Memory Profiling and Migration for Multi-Tiered Large Memory. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627703.3650075

1 Introduction

The memory hierarchy is adding more tiers and becoming more heterogeneous to cope with performance and capacity demands from applications. Multi-tier memory systems that started from multi-socket non-uniform memory access (NUMA) architecture is now a de-facto solution for building scalable and cost-effective memory systems. For instance, the Amazon EC2 High Memory Instance has three DRAM-based memory tiers built upon eight NUMA nodes [26]. The commercial availability of some memory technologies, such as high-bandwidth memory (HBM) and compute express link (CXL) [2], is adding a new dimension to memory systems. As a result, a multi-tier memory system can easily exceed two memory tiers. Top tiers feature lower memory latency or higher bandwidth but smaller capacity, while bottom tiers feature higher capacity but lower bandwidth and longer latency. When high-density memory is in use, e.g., Intel's Optane DC PM [52], a multi-tier large memory system enables high-performance, terabyte-scale graph analysis [14, 20, 47], in-memory database services [7, 10, 50, 57], and scientific simulations [39, 44, 49, 54, 55] on a single machine in a cost-effective way.

Most of the page management systems for multi-tier heterogeneous memory (HM) [5, 27, 28, 33, 38, 43, 48] consist of three components – a profiling mechanism, a migration policy, and a migration mechanism. A profiling mechanism is critical for identifying performance-critical data in applications and is often realized through tracking page accesses. A migration policy chooses candidate pages to be moved to top tiers. Finally, the effectiveness of a page management solution depends on whether its migration mechanism can move pages across tiers at low overhead. Emerging multi-tiered large memory systems calls for rethinking of memory profiling and migration to address unique problems unseen in traditional single- or two-tier systems with smaller capacity.

Problems. The large memory capacity brings challenges to memory profiling. Linux and existing memory profiling mechanisms [28] manipulate specific bits in page table entries (PTEs) to track memory accesses at a per-page granularity. This profiling method has the benefit of application-transparency, but is not scalable on a large memory system. Our evaluation shows that tracking millions of pages could take several seconds – too slow to respond to time-changing access patterns, and causes 20% slowdown in TPC-C against VoltDB [53]. The most recent solution DAMON [45, 46, 51] dynamically forms memory regions out of the virtual memory space mostly based on spatial locality, and profiles a single page per region. The total number of regions is constrained, such that the profiling overhead is controlled. DAMON has been adopted by Linux [51], and solves the profiling overhead problem faced by the large memory system, but its profiling quality can be out of control (shown in Sec. 3): DAMON can miss more than 50% of frequently-accessed pages and is slow to respond to access pattern variance.

The limitation in profiling quality comes from (1) the rigid control over profiling overhead and (2) the ad-hoc formation

of memory regions. Memory profiling relies on PTE scans. Given a large memory system with a certain constraint on profiling overhead, the PTE scan for memory profiling can only happen certain times. Deciding the distribution of those PTE scans in memory regions is critical for effective profiling. Strictly enforcing one PTE scan per region (to profile one page), as in DAMON, breaks the functionality of the profiling mechanism and compromises profiling quality. Furthermore, the ad-hoc formation of memory regions (such as randomly selecting memory regions for split) takes a long time to find pages with similar memory access patterns to form memory regions for profiling, delaying page migration.

In addition, rich memory heterogeneity brings challenges to page migration. Existing solutions, e.g., tiered-AutoNUMA [28] for multi-tiered memory are built upon an abstraction extended from the traditional NUMA systems, where page migration occurs between two neighboring tiers with the awareness of no more than two NUMA distances. However, such an abstraction limits multi-tiered memory systems, because migrating pages from the lowest to the top tier, at tier-by-tier steps, has to make multiple migration decisions to reach the destination tier, which takes multiple seconds and fails to timely migrate pages for high performance.

Furthermore, Linux and existing solutions do not consider the implications of huge pages on memory profiling and migrations. Using huge pages is common in the large memory systems to improve performance and avoid long traverse of page tables. The transparent huge page mechanism (THP) in Linux mixes huge pages and 4KB pages, which brings complexity to form memory regions for profiling.

Solutions. We argue that the following principles must be upheld to address the above problems.

- Connecting the control of profiling overhead with the profiling mechanism to enable high-quality profiling;
- Building a page migration policy on the multi-tiered memory (four tiers in our study) for high performance;
- Introducing huge page awareness.

In this paper, we contribute a page management system called MTM (standing for *M*ulti-Tiered *M*emory *M*anagement) that realizes the above principles on large four-tier memory.

MTM decouples the control of profiling overhead from the number of memory regions, but connects it directly with the number of PTE scans (the profiling mechanism). Hence, profiling quality and overhead can be distributed proportionally according to the variation of both spatial and temporal locality. More PTE scans or page profiling can be enforced for a memory region where there is large variation hence demanding more fine-grained profiling. Also, the splitting of memory regions based on the variation is able to be guided rather than randomly happened as in DAMON.

MTM breaks the barrier that blocks the construction of a universal page-migration policy across tiers. This barrier comes from the limited memory profiling functionality (either at the slowest NUMA node [43] or random selection of hundreds of MB on NUMA nodes [12, 13, 27, 29, 33, 43]). MTM uses the overhead-controlled, high-quality profiling to establish a *global* view of all memory regions in all tiers and consider all NUMA distances to decide page migration. In particular, MTM enables a "fast promotion and slow demotion" policy for high performance. Hot (frequently-accessed) pages identified in all lower tiers are ranked and directly promoted to the top tier, minimizing data movement through tiers. When a page is migrated out of the top tier to accommodate hot pages, the page is moved to the next lower tier with available space. This policy needs no apriori knowledge of the number of tiers in a system and makes the best use of fast tiers.

MTM also features a fast migration mechanism. This mechanism dynamically chooses between an asynchronous page copy-based scheme and a synchronous page migration scheme, based on the read/write pattern of the migrated pages, to minimize migration time. Finally, page migration and profiling in MTM fully supports huge pages and THP, embodied as page alignment during splitting/merging of memory regions.

Evaluation. We rigorously evaluated MTM against seven solutions on a four-tier memory system, including two state-of-the-art solutions (AutoTiering [33] and HeMem [48]), an existing solution in Linux (tiered-AutoNUMA [12]), a hardware-based solution (Optane Memory Mode), and first-touch NUMA. MTM is also compared against two kernel-based page migration solutions (the ones in Linux and Nimble [56]). MTM outperforms Memory Mode, first-touch NUMA, tiered-AutoNUMA, AutoTiering, AutoNUMA and HeMem by 20%, 22%, 24%, 25% and 24%. MTM outperforms the Linux and Nimble migration approaches by 40% and 36% for read-intensive workloads, and performs similarly for write-intensive ones.

2 Background and Related Work

In a multi-tiered large memory system, each processor has its local memory as a fast memory tier, and has memory expansion or other processor's local memory as a tier of slow memory. The memory expansion may use CXL interconnect and appear as CPU-less memory nodes. The Intel Optane-based system is an example with two sockets and four memory components (each socket has one DRAM and one PM). Two PM components appear as CPU-less memory nodes in Linux. Each processor has its local DRAM as a fast tier, and other components as slow tiers.

2.1 Large Memory Systems

Software support for page management. Recent work manages large memory systems based on an existing NUMA balancing solution [38] in Linux. Tiered-AutoNUMA [28], for example, balances memory access between CPU-attached memory nodes, and then balance memory accesses between CPU-attached memory node and CPU-less memory node

based on page hotness. As a result, a hot page takes a long time to migrate to the fastest memory for high performance. AutoTiering [33] is a state-of-the-art solution based on NUMA balancing. It introduces flexible page migration across memory tiers. However, it does not have a systematic migration strategy guided by page hotness. HeMem [48] is a recent solution for two-tiered HM. HeMem only uses perf-counters for mem-profiling and fails to explore more than two tiers.

Hardware-managed memory caching. Some large memory systems use fast memory as a hardware-managed cache to slow memory. For example, DRAM can work as a hardware-managed cache to Optane. However, this solution results in data duplication, wasting fast memory capacity. It also causes write amplification when there are memory cache misses [8].

2.2 Two-Tiered Heterogeneous Memory

Existing application-transparent solutions measure data reuse and migrate data for performance [5, 15–18, 21, 25, 31, 32, 35, 40, 48, 56]. However, they can cause uncontrolled profiling overhead or low profiling quality, and are not designed for more than two memory-tiers. Pond [37] does not migrate data but only works for two tiers. The recent industry solutions TPP [43] and TMTS [19] only work for two tiers. The above solutions cannot work for more than two tiers, because they cannot decide which tier is the page-migration target when more than one tier are the potential target.

3 Motivation

An accurate, low-overhead profiling mechanism is critical for managing multi-tiered large memory. We study the profiling methods in state-of-the-art works (Thermostat [5], AutoTiering [33] and Linux's DAMON [46, 51]) and summarize their overhead and accuracy tradeoff as follows. AutoTiering randomly chooses 256MB pages for profiling to detect hot pages. Both Thermostats and DAMON maintain a list of memory regions and randomly choose one page per region for profiling. Thermostats keeps all memory regions in a fixed size while DAMON dynamically splits and merges memory regions to improve profiling quality – they control profiling overhead by changing the number of memory regions.

We compare the profiling methods by running GUPS [22] with a 512GB working set on the four-tier Opatane memory system. We know a priori page hotness in each profiling interval during execution the page hotness of GUPS follows a Gaussian distribution). Figure 1 reports profiling *recall* (i.e., the ratio of the number of correctly detected hot pages to the number of hot pages identified by priori knowledge) and profiling *accuracy* (i.e., the ratio of the number of correctly detected hot pages to the number of total detected hot pages including incorrect ones).

Under the same profiling overhead (5%), Thermostat and AutoTiering take long time to reach high recall (i.e., slower to identify hot pages in Figure 1.a), because their randomness in

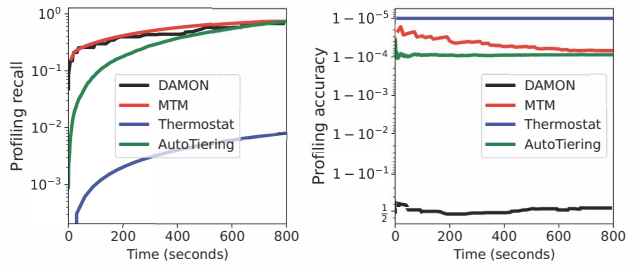

Figure 1. Comparison of different memory profiling methods in terms of their effectiveness of identifying hot pages. GUPS selects 20% of its memory footprint as hot pages according to the number of memory accesses.

page sampling and the formation of memory regions cause uncontrolled profiling quality. DAMON takes shorter time, but about 50% of hot pages detected by DAMON are not hot (see Figure 1.b), because of its ad-hoc design of forming regions and slow response to the variance of memory access patterns. Note that the profiling accuracy 50% is for the detection of *hot pages*, indicating that only half of the pages DAMON labels as hot are actually hot pages. In contrast, for the identification of cold pages, DAMON demonstrates high-precision: all pages designated as cold by DAMON are cold pages. Due to low profiling quality, DAMON, Thermostat, and AutoTiering perform 15% worse than MTM.

Memory profiling support for multi-tiered large memory in MTM significantly extends the latest Linux support (i.e., DAMON). DAMON splits a process's virtual memory space into memory regions. In each profiling interval, it randomly profiles one 4KB page per region to capture spatial locality. The control of profiling overhead is achieved through a user-defined maximum number of regions for profiling. DAMON merges two neighbor memory regions if they have similar profiling results, and splits *each* region into two randomly-sized regions to improve profiling quality if fewer than half of the maximum number of regions exist.

We identify multiple limitations in Linux (DAMON).

1. The control of profiling overhead is not directly connected with the profiling mechanism (i.e., scanning PTEs), but connected with the number of memory regions and only one random page per region is profiled. This constraint compromises profiling quality.
2. Splitting regions is ad-hoc. Splitting *each* region into two can lead to useless profiling when the new regions after splitting have the similar memory access patterns.
3. The process of forming memory regions is slow to capture access patterns, because of the time constraint in the profiling intervals. This is problematic in large memory systems with many memory regions to split and merge.
4. Temporal locality is not considered well.
5. Lack of support for profiling huge pages because of unawareness of huge pages.

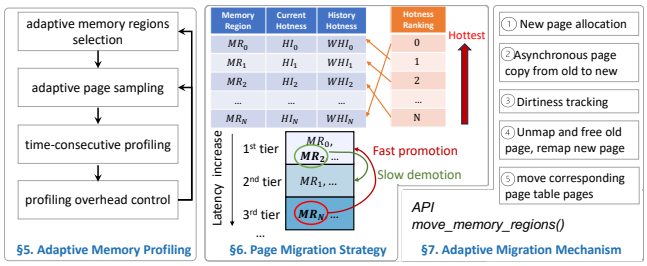

Figure 2. The overview of MTM

.

4 Overview

Figure 2 overviews MTM.name is designed to control the profiling overhead by considering total number of PTE scans in all memory regions (Sec. 5.3). Like DAMON in Linux, MTM partitions the virtual address space of a process into memory regions for profiling and dynamically merges and splits them. However, MTM has the freedom to perform PTE scan multiple times for a page or multiple pages in a memory region in a profiling interval (Sec. 5.2). Having such flexibility provides opportunities to re-distribute page-sampling quotas between regions under a fixed profiling overhead to improve profiling quality, addressing Limitations 1 and 2. Also, MTM uses performance counter-assisted PTE scan to quickly detect changes in access patterns and address Limitation 3.

MTM decides page migration between tiers based on a "global view" of all memory regions (Sec. 6). By calculating the exponential moving average of page hotness collected from all profiling intervals, MTM learns the distribution of hot regions in all memory tiers, addressing Limitation 4.

Guided by the new profiling techniques, MTM introduces the "fast promotion and slow demotion" policy (Sec. 6.2). Also, when migrating pages, MTM uses an asynchronous page-copy mechanism that overlaps page copying with application execution. This mechanism reduces the overhead of page copy, but come with the time cost of *extra* page copy, because when a page is updated during copying, the page has to be copied again. The traditional, synchronous page-copy mechanism does not need extra page copy, but completely exposes the overhead of page copy into the critical path. Hence, MTM uses a hybrid approach that takes advantage of both mechanisms, and selects one based on whether page modification happens during migration (Sec. 7).

To support huge pages, MTM enables page profiling at huge page level instead of 4KB-page level using page table information; memory merging and splitting are carefully managed to be aligned with huge page size, such that huge page semantics is not broken (Sec. 5.4).

In summary, MTM has (1) an adaptive profiling mechanism with high profiling quality and constrained overhead; (2) a page-migration strategy using a global view to make informed decisions; and (3) a page-migration mechanism adapting data copy schemes based on page access patterns.

5 Adaptive Memory Profiling

MTM tracks page accesses using a PTE reserved bit (the 11st bit) and PTE scan without flushing TLB to reduce overhead as in [5, 46, 51]). Each PTE maintains an access bit, indicating its access status. The access bit is initially set to 0, but changed to 1 by the memory management unit (MMU) when the corresponding page is accessed. By repeatedly scanning PTE to check the value of the access bit and resetting the access bit, page accesses can be monitored. This mechanism is commonly used in Linux [25, 46].

Scanning all PTEs to track memory accesses to each page is prohibitively expensive for large memory. For example, scanning a five-level page table for 1.5 TB memory in 2MB pages on an Optane-based platform (hardware details in Table 1) with helper threads takes more than one second – infeasible to capture workload behaviors online. Thus, page sampling is often used to avoid such high overhead. However, large memory systems have large numbers of pages and the profiling quality with unguided, random sampling [5, 27, 29, 33, 46] could lead to poor performance. MTM systematically forms memory regions to address this problem.

5.1 Formation of Memory Regions

A memory region is a contiguous address space mapped by a last-level page directory entry (PDE) by default. This indicates that in a typical five-level page table, the memory region has a default size of 2MB. During program execution, whenever a last-level PDE is set as valid by the OS, the corresponding memory region is subject to profiling.

Multiple scans of PTEs. In a profiling interval, the access bit of the PTE corresponding to a sampled page is scanned multiple times. The total number of scans per PTE per profiling interval is subject to a constant, *num_scans*.

We use the multi-scan, instead of single-scan to reduce skewness of profiling results. In a profiling interval, a single-scan can only detect whether a page is accessed, but cannot accurately capture the number of accesses. Although aggregating memory accesses across intervals alleviates this problem, the skewness of profiling results can be accumulated over time (see Sec. 6.1), leading to sub-optimal migration decisions. Using the multi-scan avoids this problem.

At the end of a profiling interval, the average number of accesses to all sampled pages in a memory region is used to indicate the *hotness* of that region. Based on the results, MTM may merge or split regions. Note that the formation of regions through merging and splitting is based on "logical" regions and there is no change to PTE during the formation.

Merge memory regions. MTM actively looks for opportunities to merge adjacent memory regions at the end of a profiling interval. If the disparity in hotness between two

contiguous regions in the most recent profiling interval is below a threshold τ_m, these regions are merged together.

Split a memory region. MTM performs a check to determine whether a memory region should be split in order to ensure that pages within the region exhibit similar hotness. If the maximum difference in the number of accesses between all sampled pages within a region surpasses a threshold τ_s, the region is then split into two equally-sized regions.

Selection of τ_m and τ_s. τ_m and τ_s define the minimum and maximum discrepancies in the number of memory accesses between sampled pages within a region. They, falling within the range of [0, *num_scans*], play a crucial role in avoiding frequent merging/splitting and balancing between them. To strike a balance, $\tau_m = 1/3 * num\_scans$ and $\tau_s = 2/3 * num\_scans$ by default. Also, τ_m can be dynamically fine-tuned to enforce profiling overhead constraint (see Sec. 5.3).

5.2 Adaptive Page Sampling

MTM can adapt the number of sampled pages in a region to improve profiling quality, because the profiling overhead is decoupled from the number of regions and relies on counting PTE scans. Since each sampled page has the same number of PTE scans per profiling interval, the control of PTE scans is implemented by the control of page samples (see Sec. 5.3).

Initial page sampling. MTM differentiates the slowest tier from other tiers in each profiling interval. The slowest memory tier relies on performance counters to identify memory regions with memory accesses, which are then subjected to PTE scan-based profiling. In this tier, each memory region has only one page profiled, specifically the page captured by the performance counters. In all other memory tiers, every memory region is profiled, with each region initially assigned a random page for profiling.

After merging two memory regions, the combined total of page samples from both regions is halved, under the constraint that the new region has at least one sample. This reduction reduces the profiling overhead for the two merged regions, enabling other memory regions to have additional samples without exceeding the overhead constraint.

The page samples saved through the merging process are redistributed to other memory regions. MTM distributes the sample quota to the memory regions exhibiting the largest variance in hotness indication across the last two profiling intervals among all memory regions. A large variance in hotness indication within these intervals suggests a change in memory access pattern. In such cases, allocating additional profiling samples enhances profiling quality.

To efficiently identify memory regions with the highest variance in hotness indication among all regions, MTM maintains a record of the top-five largest variances along with the corresponding regions during the analysis of profiling results. We choose "five" empirically to make it lightweight. Whenever a new profiling result for a region is available, MTM checks the top-five records and updates them accordingly. Following the merging process, the saved page-sample quota is redistributed to these top-five regions.

After splitting a memory region into two new regions, the page sample quota in the original region is evenly distributed to the two regions. Hence, splitting does not change the number of total PTE scans. Splitting memory regions brings two benefits. First, the hotness indication, which is the *average* number of accesses to all sampled pages in a memory region, provides better indication of memory accesses to the new, smaller memory regions, hence providing better guidance on page migration. Second, migration is more effective, because using the smaller memory region avoids unnecessary data movement coming with the larger region.

5.3 Profiling Overhead Control

MTM allows the user to define a profiling overhead constraint. MTM respects this constraint while maximizing profiling quality, by dynamically changing the number of memory regions and distributing page-sample quotas between the regions. The overhead constraint is a percentage of program execution time without profiling and migration. For example, in our evaluation, this constraint is 5%. Given the profiling interval (t_{mi}), profiling overhead constraint, overhead of scanning one PTE (*one_scan_overhead*), and the number of scans per PTE (*num_scans*), the total number of page samples in all memory regions to be profiled in a profiling interval, denoted as *num_ps*, is calculated in Equation 1.

$$num\_ps = \frac{t_{mi} \times profiling\_overhead\_constraint}{one\_scan\_overhead \times num\_scans} \quad (1)$$

t_{mi} can be set by the user, as in works [5, 25, 46]. "*one_scan_overhead*" is measured offline. As MTM merges or splits memory regions, the total number of page samples in all regions remains equal to *num_ps* to respect the overhead constraint.

The total number of regions needs to be smaller than *num_ps* so that each region has at least one page sample. When the total number of regions is too large, MTM temporarily increases τ_m (and keeps $\tau_s > \tau_m$) to merge regions more aggressively. τ_m is gradually increased across profiling intervals, until the number of regions is no larger than *num_ps*, and then τ_m is reset to the original value.

Another approach to enforce the profiling overhead constraint is to change the number of scans per page sample (i.e., *num_scans*). However, we do not use this approach, because of its significant impact on profiling quality. Changing *num_scans* leads to a change of profiling results in *all* memory regions. For example, in our evaluation, when changing *num_scans* from 2 to 3, MTM changes the migration decision for at least 20% of memory regions. We set *num_scans* as a constant "3". Our empirical study shows that using a value

larger than 3 leads to no obvious change (less than 5%) in the migration decision, compared to using 3.

Memory consumption overhead. For each region, MTM stores its current hotness (HI_i) and WHI_{i-1} as two floating-point numbers to build the histogram. We also store memory-region info (i.e., the beginning address and offset within a VMA) per region. Given a TB-scale memory, this yields an overhead of hundreds of MBs, which is small. In general, the memory overhead is less than 0.01%.

5.4 Support of Huge Pages

When MTM selects a page for profiling, the selection process is huge page-aware. Specifically, at the beginning of the profiling interval, MTM examines the PTE of the selected page to determine if it is a huge page. If it is, any access to that page is captured by scanning its PTE. This approach differs from the existing huge page-aware solution (Thermostat). Thermostat randomly selects a 4KB page from the huge page to estimate the number of memory accesses, resulting in a loss of profiling quality. Furthermore, if the huge page is `mprotected`, Thermostat's 4KB-based profiling cannot happen, because that violates `mprotect` semantics [5].

Forming memory regions in MTM is also huge page-aware. When a region is split, MTM checks if the split occurs in the middle of a huge page. If so, the split is slightly adjusted to align with the huge-page boundary. Without such handling, a huge page is profiled in two regions after splitting, which may lead to a conflicting migration decision. Furthermore, the two regions after the adjustment may not be equally-sized, increasing the risk of memory fragmentation after migration. However, in practice, considering the large size of the region (at the scale of MBs), the difference in size between the adjusted regions is often less than 10KB, which does not significantly contribute to memory fragmentation.

5.5 Performance Counter-Assisted PTE Scan

After the initial page sampling (Sec. 5.2), the slowest tier uses performance counters to assist PTE scan. This addresses a problem in memory systems with multi-terabyte capacity: forming memory regions can be slow to capture access patterns when there are many regions to split and merge.Specifically, at the beginning of each profiling interval, the performance counters are briefly activated (for 10% of the profiling interval) to collect memory accesses and identify regions where memory accesses happen. Such regions are subject to high-quality profiling (PTE scan). Compared with DAMON, using performance counters can save multiple profiling intervals to identify hot pages. DAMON relies on a time-interval-based approach to opportunistically capture hot pages, while MTM is event-driven: once a region is accessed, it is immediately subject to high-quality profiling to confirm its hotness. Using perf-counters alone is not enough to provide high-quality profiling (see Sec. 9.6 with HeMem) because the counters' randomness misses hot pages.

6 Page Migration Strategy

6.1 Which Memory Region to Migrate?

At the end of a profiling interval, MTM promotes some regions to the fastest tier, and the total size of migrated regions is a constant N (N=200MB in our evaluation). This is similar to the existing works [29, 33, 42] and industry practice [27] that periodically migrates a fixed number of pages. If there is no enough free space in the fastest tier, some pages in the fastest tier are demoted first to the slower tiers (see Sec. 6.2).

Select regions for promotion. The promotion places recent frequently accessed pages into faster tiers. MTM's migration decision is holistic – considering *all* regions together regardless of which tiers those regions are currently in. MTM uses time-consecutive profiling results based on the exponential moving average (EMA) [34]of hotness indication collected from all profiling intervals. Given a sequence of data points, EMA places a greater weight and significance on the most recent data points. MTM using EMA considers temporal locality in migration decision and avoids page migration due to the bursty access pattern in a profiling interval.

We define EMA of hotness indication as follows. Assume that HI_i is the hotness indication collected at the profiling interval i for a region, and the EMA of hotness indication for that region at i, denoted as WHI_i, is defined in Equation 2. This equation is a recursive formulation including WHI_i and WHI_{i-1} from the prior interval $i-1$.

$$WHI_i = \alpha \times HI + (1-\alpha) \times WHI_{i-1} \quad (2)$$

α indicates the importance of historical information in decision making. In practice, we set α as 0.5. There are two benefits of using EMA. First, the memory consumption is small. There is no need to store all prior profiling results. Second, the computation of EMA is lightweight.

Using the EMA of hotness indication in all memory regions, MTM builds a histogram to get the distribution of EMA of all regions. The histogram segments the range of EMA values into buckets, and tracks how many and what regions fall into each bucket. When determining which memory regions to migrate to the fastest memory, MTM selects regions that fall into the highest buckets of the histogram. Building and maintaining the histogram has low overhead: whenever the EMA of hotness indication of a region is available, the histogram only needs to be slightly updated accordingly.

6.2 Where to Migrate Memory Regions?

Promotion. As regions are promoted to the fastest tier using the histogram, it is likely that after a profiling interval, there is no region to promote because those regions falling into the highest buckets of the histogram are already in the fastest tier. In that case, the regions in the 2nd highest bucket (or even lower when no promotion) are selected to promote to the 2nd-fastest mem tier. The accumulated size of regions to

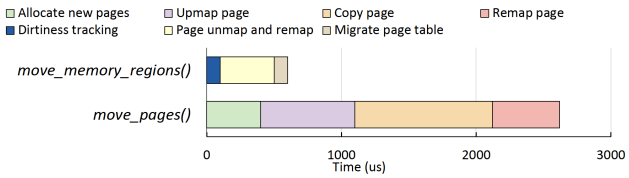

Figure 3. Breakdown for migration mechanisms.

migrate is always N. In general, MTM makes the best efforts to promote frequently accessed regions to faster tiers.

Demotion. When a memory tier is a destination of memory promotion but does not have enough space to accommodate memory promotion, some regions in that memory tier may need to be demoted to the next lower memory tier with enough memory capacity. Memory regions for demotion are selected based on the histogram – regions that are in the lowest buckets of the histogram are demoted to the next lower tier. We use the above slow-demotion strategy to avoid performance loss caused by migrating pages that are still likely to be accessed in near future.

Handling multi-view of tiered memory. The application managed by MTM can run multiple threads. Depending on which memory node a thread is close to, different threads can have different views on whether a memory node is fast or slow. For example, a thread on processor 0 thinks local DRAM is a fast memory, while a thread on processor 1 thinks that particular DRAM is a slow memory. We call this, the *multi-view of tiered memory*. The multi-view does not impact profiling results because the page hotness is only related to the number of page accesses, no matter where accesses come from. However, it impacts migration destination.

In MTM, the migration destination of a page is decided using the view of the thread that has the most accesses to that page, because enabling high performance for the most memory accesses gives the best overall performance. To support this design, during the memory profiling, MTM checks where memory accesses come from. This is implemented by leveraging the hint-fault mechanism in Linux [12]. A hint-fault takes 12× longer time than a PTE scan. To amortize this cost, every 12 PTE scans, MTM turns on the hint-fault mechanism to capture one memory access, and includes the amortized cost into *one_scan_overhead* in Equation 1.

7 Adaptive Migration Mechanism

7.1 Performance Analysis

Linux provides an API, *move_pages()*, for a privileged process to move a group of 4KB pages from a source memory node (or tier) to a target memory node (or tier). *move_pages()* has four steps, and they are performed sequentially: (1) allocate new pages in the target node; (2) unmap pages to migrate (including invalidating PTE); (3) copy pages from source to target node; (4) map new pages (including updating PTE).

Figure 3 shows the performance of migrating a 2MB memory region from the fastest tier to the slowest tier on the Optane-based platform. Copying pages is the most time-consuming step, taking 40% of total time. One reason for the high overhead is that *move_pages()* moves 4KB-sized pages sequentially. Even with multi-threaded page copy enabled by recent work [56] to maximize memory bandwidth, it remains a bottleneck, particularly for large memory region transfers.

7.2 Adaptive Page Migration Schemes

Asynchronous page copy. We introduce an asynchronous page copy mechanism to reduce page copy overhead. In the asynchronous page copy, the thread that triggers migration (named *main thread*) launches one or more helper threads to run the steps (1) and (3); the main thread runs the steps (2) and (4), and then waits for the helper thread(s) to join. With the asynchronous page copy, it is possible that copying a page happens before invalidating its PTE but the page is modified in the source memory tier after copying the page. In such cases, the page must be copied again to update its copy in the target tier, which is costly. We introduce an adaptive page-migration mechanism to address this limitation.

Adaptive page migration. For read-intensive pages, the asynchronous page copy brings performance benefit. However, for write-intensive pages, due to repeated data copy, it is likely that the asynchronous page copy performs worse than the synchronous. Hence, MTM chooses suitable migration mechanism based on write intensity of pages. In particular, MTM uses the asynchronous page copy by default. But whenever any page in the region for migration is written after the asynchronous page copy starts, MTM switches to the synchronous page copy immediately. To track page write, MTM utilizes PTE access bits and page faults, and flushes TLB for once. After detecting the first write, tracking turns off. Such tracking overhead (including setting the access bit in PTE) is small, compared with page migration itself.

We implement the above mechanism and introduce a new API called *move_memory_regions()*. In this implementation, tracking page write, performing page map/unmap, and migrating PTEs are still on the critical path, but the time-consuming page copying could be performed off the critical path. Figure 3 presents *move_memory_regions()* migrating 2MB memory region using the same setting as for *move_pages()*, and excludes the overhead of page copying (and page allocation in the step (1), using asynchronous page allocation). *move_memory_regions()* is 4.37× faster than *move_pages()* in this case. Sec. 9.5 shows more results.

The asynchronous migration is conducted by multiple kernel threads, similar to the recent Linux patch Nimble [56]. These threads are assigned with the lowest possible nice-value (a high priority) to ensure timeliness of migration under the management of thread scheduling.

8 Implementation

We implement the adaptive profiling as a kernel module that periodically scans PTEs based on adaptive page sampling and uses performance counters to guide profiling. The kernel module takes a process ID as input. Profiling results are saved in a shared memory space, and stored as a table.

We implement the page management as a daemon service at the user space (not the kernel space), which minimizes kernel changes and provides flexibility to change the page migration strategy. The daemon service executes with the application and calls the kernel module for profiling at the beginning of each profiling interval. At the end of each profiling interval, the service reads collected profiling results from the shared memory space. With overhead control, the kernel module ensures that profiling always finishes within a profiling interval. The daemon service then makes the migration decision and performs migration using *move_memory_regions(). move_memory_regions()* takes the same input as Linux *move_pages()*, but implements the adaptive migration mechanism. It detects page dirtiness during the migration by setting a reserved bit in PTE to trigger a write protection fault when there is write to the memory region. Leveraging a user-space page fault handler, *move_memory_regions()* tracks writes, and decides whether to stop the async page copy and switch to the sync.

To use performance counters, MTM uses Intel processor event-based sampling mode (PEBS) to collect memory accesses into a preallocated buffer, and uses a register interrupt handler to indicate when the buffer is full. Specifically, MTM uses two hardware events to capture memory accesses, `MEM_LOAD_RETIRED.LOCAL_PMM` and `MEM_LOAD_RETIRED.REMOTE_PMM`. The sampling frequency is 200, as in production environment [43] ("200" means taking one sample out of 200 memory accesses), which is sufficient to distinguish hot and cold pages [48]. Except architecture-dependent events in hardware performance counter, MTM is a general design working for other architecture, and as long as there are memory access-related events for slow and fast memories, MTM is able to work. For example, AMD processors provide accesses to performance counter similar to PEBS, such as Instruction Based Sampling (IBS)[1] and Lightweight Profiling (LWP)[6], which can be used to profile memory accesses at different memory tiers.

9 Evaluation

Testbed. We evaluate MTM on a two-socket machine. Each socket has Intel Xeon Gold 6252 CPU (24 cores), 756GB Intel Optane DC PM and 96GB DRAM. Table 1 shows the details for the platforms. By default, we use THP by using `madvise` and use 2MB as huge page size, which is typical in large memory systems. We set the profiling overhead constraint to 5% and the profiling interval to 10 seconds. This setting is similar to existing works and production environments [5, 25, 43, 46]. We used Linux v6.6-rc1.

Table 1. Hardware overview of experimental system.

Optane-based Multi-tiered Memory System		
Fast Mem Local Access (1st tier)	latency: 90ns	bw: 95 GB/s
Fast Mem Remote Access (2nd tier)	latency: 145ns	bw: 35 GB/s
Slow Mem Local Access (3rd tier)	latency: 275ns	bw: 35 GB/s
Slow Mem Remote Access (4th tier)	latency: 340ns	bw: 1 GB/s

Table 2. Workloads for evaluation.

Workloads	Descriptions	Mem	R/W
GUPS [22]	A measurement of how frequently a computer can issue updates to randomly generated memory locations.	512GB	1:1
VoltDB [53]	A commercial in-memory database with TPC-C [36] using 5K warehouse.	300GB /1.2TB	1:1
Cassandra [9]	A highly-scalable partitioned row store with YCSB [11] (using update-heavy benchmark A).	400GB	1:1
BFS [47]	A parallel implementation of graph traversing and searching on a graph with 0.9B nodes and 14B edges.	525GB	read-only
SSSP [47]	A parallel implementation of finding the shorted path between two vertices on a graph with 0.9B nodes and 14B edges.	525GB	read-only
Spark [58]	A spark program running the TeraSort benchmark [23].	350GB	1:1

Workloads. We use large-memory workloads in Table 2. Those workloads range from in-memory database, graph analysis, to data sorting, representing the most common memory-consuming applications. They are commonly used to evaluate big memory systems [33, 41, 48]. Their memory footprints are larger than the fastest two tiers, enabling effectively evaluation on all tiers. Unless indicated otherwise, we use eight application threads per workload.

Baselines. We use eight solutions as baselines.

- *Hardware-managed memory caching (HMC)* uses the fast memories as hardware-managed cache for slow memories. We use Memory Mode in Optane.
- *First-touch NUMA* is a common allocation policy. It allocates pages in a memory tier close to the running task that first touches the pages. It does not migrate pages.
- *Tiered-AutoNUMA* [28] in Linux (named as *vanilla tiered-AutoNUMA*). There are two recent patches for memory tiering to improve tiered-AutoNUMA: advanced hot page selection [4] based on PTE scan and automatic hot threshold adjustment [3]. The maximum promoting/demoting throughput is set to the same value as in MTM, which is 200MB per migration interval. We add the two patches and name it as *patched tiered-AutoNUMA. In the rest of the paper, we use the patched tiered-AutoNUMA by default.*
- *AutoTiering* [33].
- *HeMem* [48] is for two-tiered systems. HeMem leverages performance counters alone to find hot pages.
- *Thermostat* [5] is for two-tiered systems. It allocates all pages in the fast tier and selectively moves them to the slow tier. It cannot support applications with footprint larger than the fast tier.

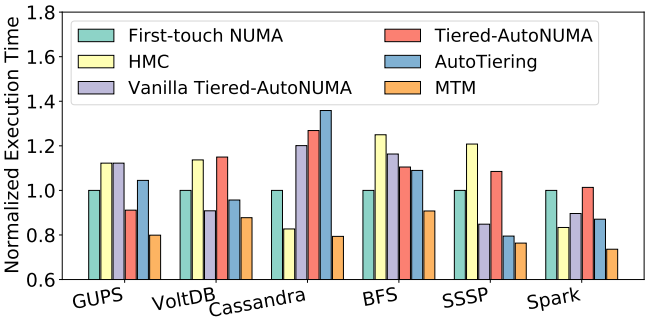

Figure 4. Overall performance (normalized to first-touch NUMA's) using existing solutions and MTM on Optane.

- *DAMON* [46, 51] is a Linux profiling tool at the memory region granularity. We use it to evaluate profiling quality.
- *Nimble* [56] is a page migration mechanism using bi-direction page copy and parallel page copy. MTM includes the techniques in Nimble but adds adaptive migration. We use Nimble to evaluate our page migration mechanism.

9.1 Overall Performance

Figure 4 shows that MTM outperforms all baselines. We have four observations.

(1) MTM outperforms HMC by up to 40% (avg. 19%). HMC incurs write amplification when cache misses occur [24], causing unnecessary data movement and low performance.

(2) MTM outperforms first-touch NUMA by up to 24% (avg. 17%). Without page migration, first-touch NUMA outperforms HMC on VoltDB and BFS, and outperforms tiered-AutoNUMA on Cassandra and BFS, indicating that page migration is not always helpful. tiered-AutoNUMA performs worse because of ineffectiveness in identifying hot pages.

(3) MTM outperforms the vanilla tiered-AutoNUMA and (patched) tiered-AutoNUMA by up to 37% and 35% (on average 23% and 20%) respectively. Specifically, tiered-AutoNUMA implements the Most Frequently Used (MFU) algorithm to precisely identify hot pages [4]. This algorithm is coupled with an automatic adjustment of the hot threshold to promote hot pages timely [3]. This strategy enables effective memory management by dynamically adapting to varying usage of memory.

To better understand the performance difference between vanilla tiered-AutoNUMA, (patched) tiered-AutoNUMA, and MTM, we examine the volume of hot pages identified and the number of accesses to the fast memory tier (i.e., the local DRAM) every 60s for the three solutions. Specifically, we configure the promotion rate of tiered-AutoNUMA and MTM to be the same, at 200MB per migration interval. Table 3 shows the results. We observe that tiered-AutoNUMA and MTM identify more hot pages than vanilla tiered-AutoNUMA by 8.2× and 7.2×, respectively, indicating that both are more effective in identifying hot pages. Meanwhile, MTM achieves 15% and 12% more fast-tier accesses on average than vanilla tiered-AutoNUMA and tiered-AutoNUMA, respectively. The same trend is observed in overall performance in Figure 4. This is because tiered-AutoNUMA uses the traditional page migration strategy, which prioritizes page swapping within a single NUMA socket. Consequently, tiered-AutoNUMA may lag in promoting pages across sockets, and promoting hot pages does not necessarily increase fast-tier accesses – in some cases, it can even be detrimental (shown in VoltDB, Cassandra, and Spark).

Table 3. Average volume of hot pages identified and number of accesses to fast memory tiers across vanilla tiered-AutoNUMA, tiered-AutoNUMA and MTM.

Workload		volume of hot pages identified	# of fast tier accesses
GUPS	vanilla-AutoNUMA	5GB	140M
	tiered-AutoNUMA	60GB	165M
	MTM	60GB	175M
VoltDB	vanilla-AutoNUMA	5GB	260M
	tiered-AutoNUMA	40GB	220M
	MTM	20GB	295M
Cassandra	vanilla-AutoNUMA	5GB	140M
	tiered-AutoNUMA	20GB	130M
	MTM	40GB	185M
BFS	vanilla-AutoNUMA	5GB	160M
	tiered-AutoNUMA	20GB	180M
	MTM	20GB	205M
SSSP	vanilla-AutoNUMA	5GB	180M
	tiered-AutoNUMA	20GB	150M
	MTM	20GB	190M
Spark	vanilla-AutoNUMA	5GB	330M
	tiered-AutoNUMA	45GB	300M
	MTM	20GB	370M

Table 4. GUPS time with different initial page placements.

Giga-Updates	1000	2000	3000	4000	5000
Slow tier first	170s	215s	276s	364s	450s
First-touch NUMA	162s	220s	276s	364s	450s

(4) MTM outperforms AutoTiering by up to 42% (avg. 17%). AutoTiering uses random sampling and opportunistic demotion, failing to effectively identify pages for migration.

Performance breakdown. We show the breakdown into application execution time, migration time, and profiling time in Figure 5. The migration time is the overhead exposed on critical path. Specifically, we first measure the computation time by disabling asynchronous migration. Then, we calculate the migration time by subtracting the profiling overhead and computation time from the total end-to-end execution time. We only compare tiered-AutoNUMA, AutoTiering and MTM because they are the only solutions that leverage four memory tiers for migration. We add first-touch NUMA as a baseline because the evaluated solutions use it for memory allocation. In all cases, the profiling overhead falls within the overhead constraint.

With tiered-AutoNUMA, the time reduction is lower or equal to the overhead of profiling and migration (see VoltDB and Cassandra). Hence, they perform worse than first-touch NUMA. Compared to tiered-AutoNUMA, MTM spends similar time in profiling but 3.5× faster in migration, reducing the execution time by 21% on average. Compared to AutoTiering, MTM spends similar time in profiling but 25% faster in migration, and reduces the execution time by 19% on average.

Impact of initial page placement. MTM initially allocates pages in a local slow memory tier, while existing solutions leveraging first-touch NUMA allocates pages in a local fast memory tier. With MTM, we study the impact of these two solutions for initial page placement. Table 4 shows the results for GUPS whose memory footprint is larger than memory capacity of the first two tiers. Near the beginning of the execution (1000 Giga-updates), there is 4.9% performance difference between the two placements. Such a difference becomes ignorable as GUPS makes more progress because MTM effectively uses pages in all tiers.

Performance contribution of adaptive profiling. We compare MTM with the patched tiered-AutoNUMA and Thermostat (using periodical PTE scan without adaptive memory regions for profiling), and all the three use async page migration. Figure 7 shows the results. With the adaptive profiling method, MTM outperforms the patched tiered-AutoNUMA by 14% and 17%, respectively. Since DAMON is a profiling tool and not integrated with any HM management system, we compare our profiling method with DAMON in Section 9.3.

Performance contribution of async page migration. We explore the performance of MTM with the traditional sync page migration and our async page migration using the same adaptive profiling method. Figure 7 (see "w/o asyn migration") shows the result. The async page migration reduces migration overhead by 60%, and leads to 12% performance improvement.

Memory overhead with using MTM. Table 5 shows the memory overhead caused by MTM and the total memory consumption for each workload. Specifically, MTM records memory region ID and the address range for all memory regions. The number of memory regions is determined by the profiling overhead control (see Equation 1). MTM tracks both the current and historical hotness of each memory region. Additionally, it utilizes a hash map for efficient indexing of address ranges of memory regions. Overall, the memory overhead incurred by MTM is less than 0.01%

9.2 Profiling Statistics

Number of memory accesses. We count the number of memory accesses in each memory tier when running VoltDB.

Table 5. Extra memory used in MTM for memory management.

	GUPS	VoltDB	Cassandra	BFS	SSSP	Spark
memory overhead	240MB	120MB	100MB	250MB	250MB	180MB
workload memory	512GB	300GB	400GB	525GB	525GB	350GB

Table 6. Quantifying mem accesses using VoltDB.

# of memory accesses	Tiered-AutoNUMA	AutoTiering	MTM
tier 1	270M	258M	293M
tier 2	5M	34M	220K
tier 3	25M	30M	10M
tier 4	90K	2.5M	205K

Table 7. Statistics of forming regions using MTM. "MR" and "PI" stand for memory regions and profiling interval.

	# of PI	avg # of MR merged in a PI	avg # of MR splited in a PI	avg # of MR in a PI
GUPS	1000	26.5	20.4	2410
VoltDB	800	53.2	50.6	1274
Cassandra	1600	42.5	63.2	1073
BFS	120	16.7	17.3	2574
SSSP	360	21.3	18.2	2492
Spark	800	35.9	32.5	1852

We only report the results for tiered-AutoNUMA, AutoTiering, and MTM because only they can leverage all four memory tiers for migration. We use Intel Processor Counter Monitor [30] to count the number of memory accesses and *exclude memory accesses caused by page migration.* This counting method allows us to evaluate the number of memory accesses from the application (not from page migration). Table 6 shows the results where there are eight VoltDB clients residing in one processor, and the view of tiered memory is defined from their view. Table 6 shows that with MTM, the number of memory accesses in the fastest memory tier (tier 1) is 12% and 14% more than with tiered-AutoNUMA and AutoTiering. This indicates that MTM effectively migrates frequently accessed pages to the fast tier for high performance because of the new profiling method.

Statistics of memory regions. On average, the number of memory regions merged and split in a profiling interval is 3.4% of all memory regions, as reported in Table 7. Compared with DAMON, MTM has less merging/splitting because of performance counter guidance and effective formation of memory regions. For example, GUPS with MTM has 12% less merging and 32% less splitting than with DAMON.

9.3 Effectiveness of Adaptive Profiling

Comparison with tiered-AutoNUMA and Thermostat. We study profiling quality and overhead, and compare MTM with two sampling-based profiling methods: one used in tiered-AutoNUMA and AutoTiering, and the other used in

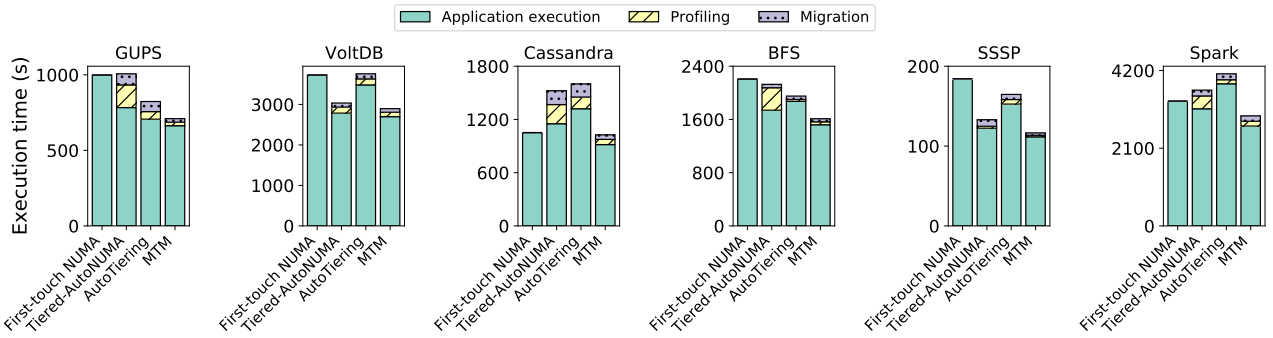

Figure 5. Breakdown of application execution time.

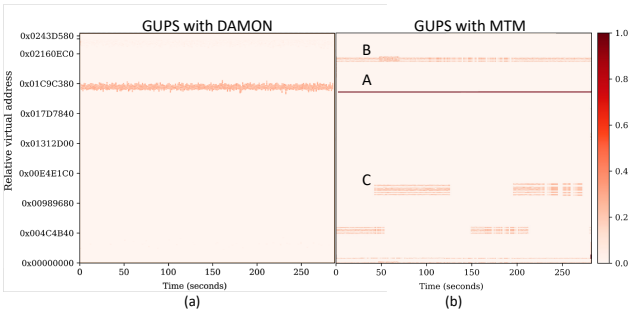

Figure 6. Heatmap of memory accesses in GUPS using (a) DAMON and (b) MTM under the same profiling overhead (5%).

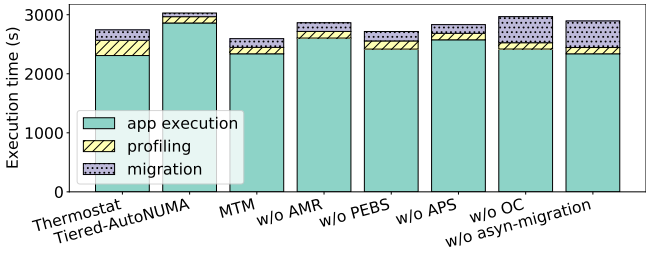

Figure 7. Evaluation of the effectiveness of adaptive memory regions ("AMR"), adaptive page sampling ("APS"), profiling overhead control ("OC"), and async migration using VoltDB.

Thermostat. We use tiered-AutoNUMA and Thermostat, and replace their migration with MTM's, which excludes the impact of their migration on performance, and is fair.

Tiered-AutoNUMA randomly profiles a 256MB virtual address space in each profiling interval, and then manipulates the present bit in each PTE in the chosen address space. This method tracks page accesses by counting page faults. The profiling method in Thermostat randomly chooses a 4KB page out of each 2MB memory region for profiling. This method manipulates page protection bits in PTE and leverages protection faults to count accesses.

Figure 7 shows that MTM outperforms tiered-AutoNUMA and Thermostat by 17% and 7%, respectively. The profiling overhead in Thermostat is 6× higher than in tiered-AutoNUMA, since the number of sampled pages in Thermostat profiling is much larger than that in tiered-AutoNUMA. The profiling overhead in Thermostat is 2.5× higher than in MTM, because manipulating reserved bits in PTE and counting protection faults in Thermostat is more expensive than scanning PTEs in tiered-AutoNUMA and MTM. With tiered-AutoNUMA, the application run time is longer than with MTM by 22%. This indicates that random sampling-based profiling is not as effective as our adaptive profiling.

Comparison with DAMON. We use GUPS that randomly accesses 512GB memory with 24 threads. Specifically, 20% of GUPS' memory footprint is randomly selected as the hotset. Each thread randomly updates the memory 1M times, and 80% of them happens in the hotset. 1M-updates repetitively happens, so that there is variance on hot pages. Using knowledge on GUPS, we know there are three hot data objects: the indexes to access the hotset (labeled as "A"), the hotset information (labeled as "B"), and the hotset (labeled as "C").

Figure 6 shows results. (1) MTM finds C, while DAMON cannot because of its slow response. (2) MTM finds B, but DAMON cannot because its memory regions are initially set based on the virtual memory area tree, which is too coarse-grained to capture B even after splitting regions. (3) DAMON and MTM find A, but the scope of A found by MTM is correctly narrowed down, which reduces unnecessary migration.

Effectiveness of adaptive memory regions. We disable adaptive memory regions but respect the profiling overhead constraint. Figure 7 shows application execution time is 22% longer, although the constraint is met.

Effectiveness of adaptive page sampling. This technique distributes PTE scans between regions by using time-consecutive profiling, which includes information on temporal locality. We disable it and randomly distribute PTE scans between regions, and observe 21% performance loss.

Evaluation of profiling overhead control. We set $\tau_m = \tau_s = 0$ (i.e., no merging/splitting memory regions) and observe that the profiling time is increased by 3× in Figure 7.

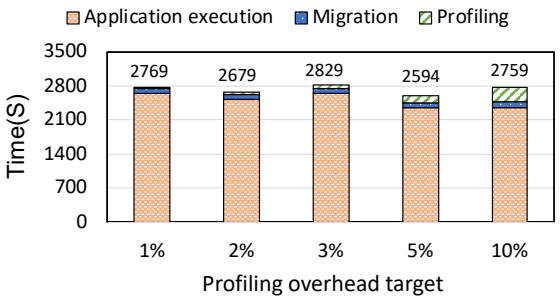

Figure 8. Execution time with various profiling overhead targets. We evaluate voltDB with MTM.

Evaluation of performance-counter assistance. MTM-w/o-PEBS in Figure 7 does not use performance counters to guide profiling. It performs worse than MTM because of lack of PEBS guidance. For VoltDB, it performs 4% worse. The overhead of PEBS is less than 1%. For all benchmarks, the performance improvement is 10.6% on average.

9.4 Sensitivity Study

Impacts of profiling overhead. We study the relationship between the profiling overhead and quality. Figure 8 shows the results. We set profiling interval as 5s, and test a set of profiling overhead targets. As the overhead increases from 1% to 10%, application execution time is reduced by 12%. Such performance improvement comes from the improvement of profiling quality when trading profiling overhead (by taking more samples) for quality. However, taking more samples (or using a larger overhead target) does not necessarily lead to better performance. As shown in Figure 8, application execution time increases by 7% as the profiling overhead increases from 5% to 10%. We use 5% as the overhead target, which universally works well for all applications in our study.

Impacts of profiling thresholds τ_m and τ_s. We study the relationship between memory region merging/split thresholds (τ_m and τ_s) and profiling quality. Increasing τ_m leads to aggressive merging of memory regions, and decreasing τ_s leads to aggressive split of regions. We change τ_m and τ_s and measure performance. Figure 9 shows the results.

With *num_scans* set as 3, $\tau_m = 1$ and $\tau_s = 2$ lead to the best performance, outperforming other configurations of τ_m and τ_s by at least 7%. The execution time of voltDB increases as τ_m increases. We observe that more aggressive merging of memory region leads to inaccurate profiling results: In the extreme case, when $\tau_m = num\_scans$, there is only one region. The inaccurate profiling leads to bad application performance. Both profiling overhead and execution time increase when τ_s decreases. With aggressive split of regions, MTM uses a long time to capture memory access patterns, which increases profiling overhead and loses profiling quality. We observe the same trend when *num_scans* is set as 6.

Impact of memory promotion threshold α. α is used in Equation 2 to balance the contributions of the historical profiling results and current profiling results. When $\alpha = 0$, MTM makes migration decisions only based on the historic result. When $\alpha = 1$, MTM ignores historic information. We set α with various values. Figure 10 demonstrates that different applications have different sensitivity to α. Using both profiling results are generally helpful for most of applications (e.g., GUPS, voltDB, Cassandra, BFS and SSSP).

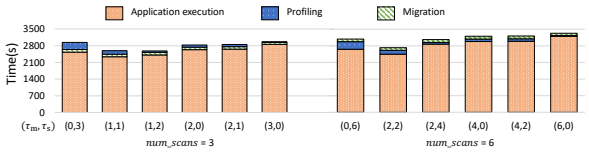

Figure 9. Sensitivity study for τ_m and τ_s. We run voltDB.

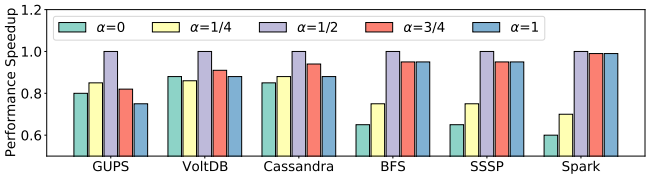

Figure 10. Performance when changing α. The performance is normalized by that of the default setting $\alpha = 1/2$.

9.5 Migration Mechanism in MTM

Effectiveness of migration mechanism We use three microbenchmarks to evaluate the migration mechanisms in MTM, Nimble [56], and *move_pages()* in Linux: sequential read-only (shown as R), 50% read (i.e., a sequential read followed by an update on an array element, shown as R/W), and 100% sequential write (shown as W) on a 1GB array. The array is allocated and touched in a tier, and then migrated to another.

Figure 11 shows the performance breakdown of three migration scenarios. For read-intensive pages, MTM brings large benefit because of async page copy. On average, MTM outperforms *move_pages()* and Nimble by 40% and 23% on average for different cases. For write-intensive pages, MTM tracks page dirtiness. When write is detected on migrated page, MTM switches async page copy to the sync. In this case, MTM performs similar to *move_pages()*. We measure its overhead by repeatedly triggering faults. On average, it takes about 40μs to handle a page fault. This overhead is small, because it is paid only once when moving a region and moving a region (at least 2 MB) takes at least multiple milliseconds, much longer than 40μs. Migrating pages between the tiers 1 and 2, MTM's mechanism performs 40%, 23%, and -0.5% better than *move_pages()*, and performs 26% 4% and -6% better than Nimble, for the three benchmarks respectively. We see the same trend in other tiers.

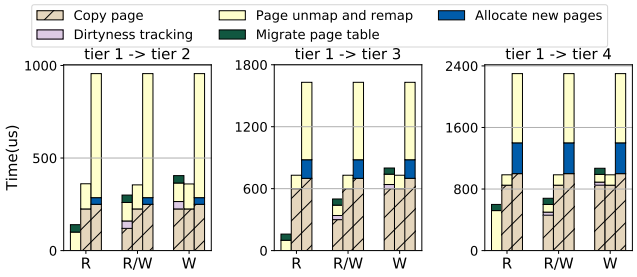

Figure 11. Performance comparison between Nimble, *move_pages*() and MTM.

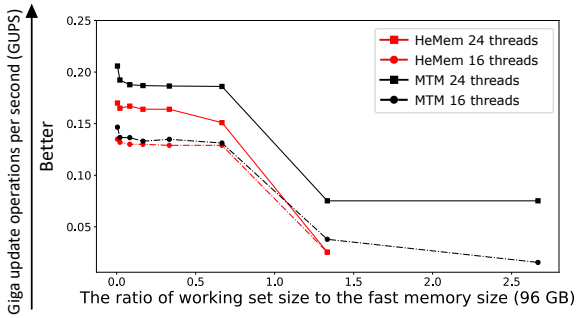

Figure 12. MTM and HeMem on two-tiered HM.

9.6 Evaluation with Two-Tiered HM

We use a single socket with two tiers (DRAM and Optane) on the Optane platform as a two-tiered HM. We use GUPS [22] as in HeMem [48] (a solution for two-tiered HM). Figure 12 reports the performance of using 16 and 24 application threads. The results show that when the working set size fits into the fast memory tier (i.e., the ratio in the x axis is smaller than 1.0), MTM performs similarly to HeMem at 16 threads but better at 24 threads. Once the working set size exceeds fast memory, HeMem fails to sustain performance at 24 threads while MTM still sustains higher performance at 24 threads than at 16 threads. MTM performs better because its profiling method can quickly adapt to changes in memory accesses and identify more hot pages.

10 Conclusions

Emerging multi-tiered large memory systems calls for rethinking memory profiling and migration for high performance. We present MTM, a page management system customized for large memory systems.

Acknowledgments

This paper is a result of a research project sponsored by SK hynix Inc. This project was partly supported by the NSF grant 2348350. We would like to thank the anonymous reviewers, as well as our shepherd, Antonio Barbalace, for their feedback on the paper.

References

[1] 2017. AMD IBS Toolkit. https://github.com/jlgreathouse/AMD_IBS_Toolkit.

[2] 2021. Compute Express Link Industry Members. https://www.computeexpresslink.org/members.

[3] 2022. Memory Tiering: Adjust Hot Threshold Automatically . https://github.com/torvalds/linux/commit/c959924b0dc53bf6252793f41480bc01b9792570.

[4] 2022. Memory Tiering: Hot Page Selection with Hint Page Fault Latency . https://github.com/torvalds/linux/commit/33024536bafd9129f1d16ade0974671c648700ac.

[5] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[6] AMD. 2010. AMD64 technology: Lightweight profiling specification.

[7] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. 2017. SAP HANA Adoption of Non-volatile Memory. *Proc. VLDB Endow.* 10, 12 (Aug. 2017).

[8] Julian T. Angeles, Mark Hildebrand, Venkatesh Akella, and Jason Lowe-Power. 2021. Investigating Hardware Caches for Terabyte-scale NVDIMMs. In *Annual Non-Volatile Memories Workshop*.

[9] Apache. 2021. Open Source NoSQL Database. https://cassandra.apache.org/.

[10] Cheng Chen, Jun Yang, Mian Lu, Taize Wang, Zhao Zheng, Yuqiang Chen, Wenyuan Dai, Bingsheng He, Weng-Fai Wong, Guoan Wu, Yuping Zhao, and Andy Rudoff. 2021. Optimizing In-Memory Database Engine for AI-Powered Online Decision Augmentation Using Persistent Memory. In *Proceedings of the VLDB Endowment*.

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

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

[13] Vladimir Davydov. 2015. Idle memory tracking. https://lwn.net/Articles/643578/.

[14] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. 2020. Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs. *Proc. VLDB Endow.* 13, 9 (May 2020), 1598–1613. https://doi.org/10.14778/3397230.3397251

[15] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. 2019. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *International Symposium on High-Performance Parallel and Distributed Computing*.

[16] Thaleia Dimitra Doudali, Daniel Zahka, and Ada Gavrilovska. 2021. Cori: Dancing to the Right Beat of Periodic Data Movements over Hybrid Memory Systems. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 350–359. https://doi.org/10.1109/IPDPS49936.2021.00043

[17] Zhuohui Duan, Haikun Liu, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Yu Zhang. 2019. HiNUMA: NUMA-Aware Data Placement and Migration in Hybrid Memory Systems. In *2019 IEEE 37th International Conference on Computer Design (ICCD)*. 367–375. https://doi.org/10.1109/ICCD46524.2019.00058

[18] Subramanya R Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the Eleventh European Conference on Computer Systems*. ACM, 15.

[19] 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. 2023. Towards an Adaptable Systems Architecture for Memory Tiering at Warehouse-Scale. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[20] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. 2020. Single Machine Graph Analytics on Massive Datasets Using Intel Optane DC Persistent Memory. *Proc. VLDB Endow.* 13, 8 (April 2020), 1304–1318. https://doi.org/10.14778/3389133.3389145

[21] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. 2022. Direct Access, High-Performance Memory Disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 287–294. https://www.usenix.org/conference/atc22/presentation/gouk

[22] GUPS. 2021. Giga Updates Per Second. http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/.

[23] Ewan Higgs. 2018. Spark-terasort. https://github.com/ehiggs/spark-terasort.

[24] Mark Hildebrand, Julian T. Angeles, Jason Lowe-Power, and Venkatesh Akella. 2021. A Case Against Hardware Managed DRAM Caches for NVRAM Based Systems. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[25] Takahiro Hirofuchi and Ryousei Takano. 2016. RAMinate: Hypervisor-based Virtualization for Hybrid Main Memory Systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*.

[26] Amazon Inc. 2018. Amazon EC2 High Memory Instances with 6, 9, and 12 TB of Memory, Perfect for SAP HANA. https://aws.amazon.com/blogs/aws/now-available-amazon-ec2-high-memory-instances-with-6-9-and-12-tb-of-memory-perfectfor-sap-hana/.

[27] Intel. 2019. Intel Memory Optimizer. https://github.com/intel/memory-optimizer.

[28] Intel. 2020. Autonuma: Optimize Memory Placement in Memory Tiering System. https://lwn.net/Articles/803663/.

[29] Intel. 2021. Intel Memory Tiering. https://lwn.net/Articles/802544/.

[30] Intel. 2021. Intel Processor Counter Monitor. https://github.com/opcm/pcm.

[31] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *International Symposium on Computer Architecture*.

[32] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. 2021. KLOCs: Kernel-Level Object Contexts for Heterogeneous Memory Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

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

[34] Frank Klinker. 2011. Exponential moving average versus moving exponential average. *Mathematische Semesterberichte* 58 (2011), 97–107.

[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. 2019. Software-Defined Far Memory in Warehouse-Scale Computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*.

[36] Scott T. Leutenegger and Daniel Dias. 1993. A Modeling Study of the TPC-C Benchmark. In *SIGMOD Record*.

[37] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. Pond: CXL-Based Memory Pooling Systems for Cloud Platforms. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[38] Linux. 2014. Automatic NUMA Balancing. https://www.linux-kvm.org/images/7/75/01x07b-NumaAutobalancing.pdf.

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

[40] Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li. 2019. Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems. *IEEE Transactions on Parallel and Distributed Systems* 30, 10 (2019), 2223–2236. https://doi.org/10.1109/TPDS.2019.2908175

[41] Mark Mansi and Michael Swift. 2020. 0sim: Preparing System Software for a World with Terabyte-scale Memories. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[42] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively Prefetching Remote Memory with Leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 843–857. https://www.usenix.org/conference/atc20/presentation/al-maruf

[43] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2022. TPP: Transparent Page Placement for CXL-Enabled Tiered Memory. https://doi.org/10.48550/ARXIV.2206.02878

[44] Bao Nguyen, Hua Tan, and Xuechen Zhang. 2017. Large-Scale Adaptive Mesh Simulations through Non-Volatile Byte-Addressable Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '17)*.

[45] SeongJae Park, Madhuparna Bhowmik, and Alexandru Uta. 2022. DAOS: Data Access-Aware Operating System. In *Proceedings of International Symposium on High-Performance Parallel and Distributed Computing (HPDC)*.

[46] SeongJae Park, Yunjae Lee, and Heon Y. Yeom. 2019. Profiling Dynamic Data Access Patterns with Controlled Overhead and Quality. In *Proceedings of the 20th International Middleware Conference Industrial Track* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 1–7. https://doi.org/10.1145/3366626.3368125

[47] Zhen Peng, Alexander Powell, Bo Wu, Tekin Bicer, and Bin Ren. 2018. Graphphi: efficient parallel graph processing on emerging throughput-oriented architectures. In *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques*. 1–14.

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

[49] Jie Ren, Jiaolin Luo, Ivy Peng, Kai Wu, and Dong Li. 2021. Optimizing Large-Scale Plasma Simulations on Persistent Memory-based Heterogeneous Memory with Effective Data Placement Across Memory Hierarchy. In *International Conference on Supercomputing (ICS)*.

[50] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. 2020. AIFM: High-Performance, Application-Integrated Far Memory. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 315–332. https://www.usenix.org/conference/osdi20/presentation/ruan

[51] SeongJae Park. [n. d.]. DAMON: Data Access Monitor. https://sjp38.github.io/post/damon/.

[52] Billy Tallis. [n. d.]. The Intel Optane Memory (SSD) Preview: 32GB of Kaby Lake Caching. http://www.anandtech.com/show/11210/the-intel-optane-memory-ssd-review-32gb-of-kaby-lake-caching.

[53] voltDB. 2021. voltDB. https://www.voltdb.com/.

[54] Kai Wu, Jie Ren, and Dong Li. 2018. Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Memory for Task Parallel

Programs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*.

[55] Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. 2021. MD-HM: Memoization-based Molecular Dynamics Simulations on Big Memory System. In *International Conference on Supercomputing (ICS)*.

[56] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems*.

[57] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*.

[58] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. 15–28.