Memtis: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination

Taehyung Lee
Sungkyunkwan University

Sumit Kumar Monga
Virginia Tech

Changwoo Min
Igalia

Young Ik Eom*
Sungkyunkwan University

Abstract

The evergrowing memory demand fueled by datacenter workloads is the driving force behind new memory technology innovations (*e.g.*, NVM, CXL). Tiered memory is a promising solution which harnesses such multiple memory types with varying capacity, latency, and cost characteristics in an effort to reduce server hardware costs while fulfilling memory demand. Prior works on memory tiering make suboptimal (often pathological) page placement decisions because they rely on various heuristics and static thresholds without considering overall memory access distribution. Also, deciding the appropriate page size for an application is difficult as huge pages are not always beneficial as a result of skewed accesses within them. We present Memtis, a tiered memory system that adopts informed decision-making for page placement and page size determination. Memtis leverages access distribution of allocated pages to optimally approximate the hot data set to the fast tier capacity. Moreover, Memtis dynamically determines the page size that allows applications to use huge pages while avoiding their drawbacks by detecting inefficient use of fast tier memory and splintering them if necessary. Our evaluation shows that Memtis outperforms state-of-the-art tiering systems by up to 169.0% and their best by up to 33.6%.

CCS Concepts: • **Software and its engineering → Memory management**; • **Computer systems organization → Heterogeneous (hybrid) systems**.

Keywords: Operating system, Tiered memory management, Virtual memory

ACM Reference Format:
Taehyung Lee, Sumit Kumar Monga, Changwoo Min, and Young Ik Eom. 2023. Memtis: Efficient Memory Tiering with Dynamic Page Classification and Page Size Determination. In *ACM SIGOPS 29th Symposium on Operating Systems Principles (SOSP '23), October 23–26, 2023, Koblenz, Germany*. ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3600006.3613167

1 Introduction

Motivation. Main memory significantly contributes to application performance and server costs due to the scaling limitations of DRAM technologies [40, 46, 61]. For instance, memory accounts for about 37.1% of Meta's server costs [49] and about 50% of Microsoft Azure's server costs [64]. Driven by memory-intensive applications, such as graph processing and Machine Learning (ML), the demand for main memory is continuing to expand [55, 82]. For example, ML models are rapidly growing, and are expected to grow 50× in the next five years [62]. With this rapid pace of growth, the existing memory hierarchy will not be able to keep up.

Advances in non-DRAM memory technologies (*e.g.*, NVM: Non-Volatile Memory [6]) and cache-coherent memory interconnects (*e.g.*, CXL: Compute Express Link [17, 51]) provide new opportunities to alleviate this problem. Tiering multiple types of memory with different properties, such as capacity, latency, and cost traits, provides an opportunity to build a cost-effective system with vast amounts of memory [10, 19, 24, 70]. However, the higher access latency of these technologies and the higher address translation cost of big memory applications [11, 67] can significantly degrade performance. *Therefore, a desirable tiered memory system should 1) wisely place data at the appropriate memory tier and 2) mitigate address translation cost to minimize performance degradation of high capacity tiered memory.*

Limitations of the state-of-the-art systems. Unfortunately, existing works in tiered memory systems [14, 21, 27, 29, 30, 32, 44, 48, 49, 54, 68, 69, 76, 78, 79, 84] and huge page management approaches [23, 26, 38, 43, 47, 50, 53, 57, 58, 67, 87] fail to meet the above criteria.

A desirable tiered memory system should place frequently accessed hot pages in fast tier memory (e.g., local DRAM) while

*Dept. of Electrical and Computer Engineering / College of Computing and Informatics

SOSP '23, October 23–26, 2023, Koblenz, Germany

ACM ISBN 979-8-4007-0229-7/23/10…$15.00
https://doi.org/10.1145/3600006.3613167

putting cold pages in capacity tier memory (e.g., NVM, CXL-attached memory). The biggest limitation of prior systems is their inability to effectively classify page hotness across diverse memory configurations and workloads. They rely on various heuristics and/or pre-configured thresholds to identify hot pages. As a result, identified hot pages are often either smaller or larger than the fast tier capacity, so they fail to place the hottest pages on fast tier memory. Moreover, some prior works migrate pages between tiers in the critical path (*e.g.*, page fault handler), adding non-negligible latency. We provide a detailed analysis of these problems in §2.2.

Employing huge pages is standard to reduce the address translation overhead and increase TLB reach. However, in tiered memory systems, using a huge page can waste precious fast tier memory. We found that not all subpages in a huge page are equally hot. For some workloads, some subpages are rarely accessed or not accessed at all. If an entire huge page gets promoted to the fast tier due to its few very hot subpages, the fast tier memory space for rarely or not at all accessed subpages gets wasted. In this case, it would be more beneficial to split such highly skewed huge pages and promote only hot subpages to the fast tier. *There is no one-size-fits-all page size in tiered memory systems.* In §2.3, we discuss the access skewness of subpages in a huge page.

Our work. In this paper, we introduce Memtis, *the first tiered memory system to achieve both access distribution-based page placement and skewness-aware page size determination within a bounded CPU overhead.* Our evaluation shows that Memtis outperforms all state-of-the-art tiered memory systems in almost all cases.

To make the best use of fast tier memory, Memtis dynamically determines if a page is hot, warm, or cold by considering the overall access frequency distribution of pages. Memtis collects the distribution using a page access histogram with negligible CPU (< 3%) and memory (< 0.195%) overhead. It then dynamically decides a threshold for hot, warm, and cold pages and places pages in the appropriate memory tier. Since Memtis determines page hotness by considering the overall access distribution, it can properly fill the fast tier with the hottest pages.

Memtis automatically balances the memory access cost and address translation cost. To decide page size, Memtis considers the subpage [1] access frequency in a huge page. By default, Memtis uses a huge page to reduce address translation costs. However, if only a tiny fraction of subpages in a huge page are frequently accessed (*i.e.*, a highly skewed huge page), thereby overshadowing the access benefits of the fast tier and wasting precious fast tier memory, Memtis breaks up such a huge page into multiple base pages and migrates only the hot subpages into the fast tier. Since such huge page split is an expensive operation involving data copy and TLB shootdown, Memtis carefully estimates the maximum benefit and splits only the most skewed, hottest huge pages as per the estimated benefit.

In addition, Memtis can track fine-grained, subpage-granularity memory accesses using processor event-based sampling (Intel PEBS). To avoid the excessive CPU overhead of the sampling approaches, we propose a technique that dynamically adjusts the memory access sampling intervals. Finally, all Memtis operations – memory access tracking, page migration, and huge page split/merge – are performed asynchronously in the background, so Memtis never slows down the critical path.

Contributions. We make the following contributions:

- **Analyses.** We thoroughly analyze the behavior of existing tiered memory systems with real-world memory-intensive applications and reveal two new findings: 1) hotness detection is suboptimal, resulting in a large portion of fast tier memory containing non-hottest pages; 2) access frequency of subpages within a huge page is often highly skewed, so rarely accessed subpages waste precious fast tier memory.
- **Memtis design.** We propose Memtis, the first tiered memory system that solves the above problems within a bounded CPU (< 3%) and memory (< 0.195%) overhead. Memtis harnesses page access distribution for the best tiering decision and dynamically chooses page size according to subpage access skewness of huge pages.
- **Evaluation.** We evaluate Memtis with eight representative memory-intensive applications and compare Memtis against six state-of-the-art systems [32, 49, 68, 76, 78, 84] while varying the ratios of fast tier (DRAM) and capacity tier (NVM or CXL memory). Memtis outperforms other systems by 33.6% on average (geomean). In addition, by dynamically splitting huge pages based on their skewness, Memtis achieves up to a 19.9% performance improvement and reduces memory bloat by up to 45.4%.

The Memtis prototype is available at https://github.com/cosmoss-jigu/memtis.

2 Analysis of Tiered Memory Systems

This section discusses existing tiered memory systems. In particular, we analyze the three essential aspects of tiered memory system design: 1) tracking memory access (§2.1), 2) placing memory pages into either fast tier (*e.g.*, DRAM) or capacity tier (*e.g.*, NVM, CXL-attached memory) (§2.2), and 3) deciding the best page size (§2.3). We present a comparison summary of prior work in Table 1.

2.1 Tracking Memory Accesses

Tracking memory access is an essential step in characterizing the memory access of applications. The information from the memory access characterization aids in deciding page placement such that frequently accessed pages (*i.e.*, hot pages) are kept in the fast tier while rarely accessed pages (*i.e.*, cold pages) live in the capacity tier.

[1]The term *subpage* means each 4KB-sized region in a huge page.

	Access tracking		Memory placement				Considering page size
	Mechanism	Subpage tracking	Promotion metric	Demotion metric	Criteria for thresholding	Critical path migration	
AutoNUMA [76]	Page fault	No	Recency	-	Static access count	Promotion	None
AutoTiering [32]	Page fault	No	Recency	Frequency	Static access count (promotion) LFU (demotion)	Promotion	None
Tiering-0.8 [78]	Page fault	No	Recency	Recency	Promotion rate	Promotion	None
TPP [49]	Page fault	No	Recency + Frequency	Recency	Static access count	Promotion	None
HotBox [14]	Page fault	No	Recency + Frequency	Recency	Static access count	Promotion	Base page only
Nimble [84]	PT scanning	No	Recency	Recency	Static access count	None	None
MULTI-CLOCK [48]	PT scanning	No	Recency + Frequency	Recency	Static access count	None	None
TMTS [22]	PT scanning & HW-based sampling	No	Recency + Frequency	Recency	Static access count (promotion) Period never accessed (demotion)	None	Split upon demotion
HeMem [68]	HW-based sampling	No	Recency + Frequency	Recency + Frequency	Static access count	None	None
Memtis	HW-based sampling	Yes	Exponential moving average of access frequency		Memory access distribution	None	Split based on access skew

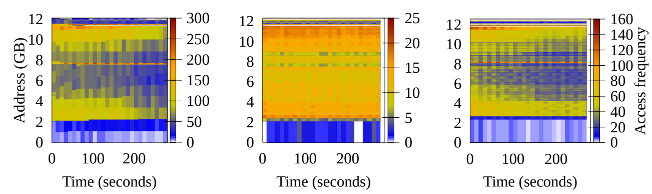
Table 1. Comparison of tiered memory systems in terms of memory access tracking, memory placement, and determining page sizes. Memory access tracking using page faults [14, 32, 49, 76, 78] increases access latency while page table scanning [48, 84] is not scalable as memory size grows. Also, page table-based approaches cannot perform fine-grained (*i.e.*, subpage granularity) tracking when a huge page is used. For memory placement, all prior works rely on either recency or frequency of page access [14, 32, 48, 49, 76, 78, 84] and/or use a statically pre-determined hotness threshold [14, 32, 48, 49, 68, 76, 84]. The low-cost, but inaccurate, hotness metric together with the static threshold makes the placement decision suboptimal. No prior work has considered access skewness in huge pages in tiering decision-making.

Page table-based access tracking. Several systems [14, 32, 49, 76, 78] use page faults to track memory access. Others [22, 27, 30, 48, 60, 84] check whether a page is accessed by scanning the associated reference bit; for `mmap`-ed pages, the processor turns on the reference bit of a page, and for file-backed pages, the OS updates this bit.

The page table-based approaches have several critical limitations. They incur high runtime overhead by triggering additional page faults or costly TLB shootdowns. Moreover, the overhead increases as the memory size and the number of processes increase because more pages and page tables need scanning. Lastly, the tracking accuracy is limited; they can only identify whether a page gets accessed between successive scanning intervals; in addition, a page being the smallest unit of tracking, fine-grained access tracking for each 4KB page is not possible when using huge pages.

Recently introduced DAMON [59] somehow mitigates the monitoring overhead of page table-based access tracking. However, it monitors memory at a coarser granularity (*i.e.*, *region*) than a page, assuming page access frequency within a region will be the same, with a short scanning interval (5 msec by default). Figure 1 clearly shows the trade-off between scanning granularity, scan interval, and accuracy in DAMON. Coarse granularity results in grouping pages with distinct access frequencies (Figure 1a) whereas fine-grained access tracking with a longer interval fails to differentiate access frequencies among regions (Figure 1b). Unfortunately, achieving high accuracy comes at significant CPU overhead (72.85% in Figure 1c).

Hardware-based memory access sampling. Leveraging the processor's hardware event-based sampling to track memory access on tiered memory systems is another approach used by recent works [15, 22, 54, 68]. Modern processors provide hardware event sampling features – Processor Event-Based Sampling (PEBS) in Intel and Instruction-Based Sampling (IBS) in AMD processors. For example, depending on the types of hardware events (*e.g.*, LLC miss) and their sampling interval (*e.g.*, once every 1000 events), PEBS stores



(a) 5ms-10-1000 **(b)** 500ms-10K-20K **(c)** 5ms-10K-20K

Figure 1. Memory access heat map of 654.roms by DAMON [59]. *s-m-X* in caption denotes DAMON configuration: *s* msec scanning interval with minimum *m* and maximum *X* regions. The CPU overhead for (a), (b), and (c) are 2.15%, 3.18%, and 72.85%, respectively.

sampled events with process ID and virtual address accessed in a PEBS buffer. Event-based access sampling is capable of reporting exact memory addresses without scanning page table entries. Notably, it can track subpage accesses in a huge page to figure out the utilization of huge pages, a property none of the prior works [15, 22, 54, 68] exploited. However, the overhead increases linearly with shorter sampling intervals since more samples are collected and processed.

Insight #1. Tracking memory access using page faults incurs high latency on the critical path. Also, page table-based approaches are coarse-grained and provide inaccurate access tracking in both space (*i.e.*, huge page) and time (*i.e.*, scanning interval) dimensions. On the other hand, event-based memory access sampling, like Intel PEBS, reports the exact address accessed but its overhead increases proportionally with the sampling frequency. Hence, efficient and accurate memory access tracking is essential for a tiered memory system that allows fine-grained access monitoring (*e.g.*, subpage accesses) and scales well for large memory sizes.

2.2 Deciding Where to Place Pages

A tiered memory system identifies frequently accessed *hot pages* and rarely accessed *cold pages*, and correspondingly migrates memory pages to the appropriate tiers.

Hotness metrics. The *recency* and *frequency* of page access are widely used metrics to predict future accesses.

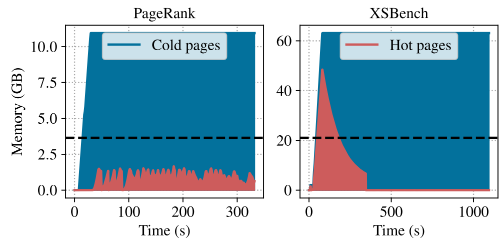


Figure 2. Identified hot and cold pages over time in HeMem [68]. The black dashed line denotes the size of fast tier (DRAM).

The recency of page access is measured in various ways, including the most recently accessed page (AutoNUMA [76]), a group of recently accessed pages (active list in Nimble [84]), and the approximate re-fault interval (Tiering-0.8 [78]). Although recency can be collected efficiently, it cannot capture a page's access history, so a placement decision solely based on recency can be suboptimal.

The access frequency of a page compensates for the limited information of recency. TPP [49] and MULTI-CLOCK [48] promote pages accessed twice or more to the fast tier by extending LRU policies. AutoTiering [32] maintains the access history of a page using an N-bit history vector, where each bit represents if a page is accessed in a scan interval. While access frequency retains more information than recency, most systems currently capture frequency in a very limited form (*e.g.*, one bit in a scan interval [32]).

Hotness threshold. In most prior approaches, a hotness threshold (used to determine if a page is hot) is deeply entangled in their design. For example, the hotness threshold in AutoNUMA [76], AutoTiering [32], and Nimble [84] is one (*i.e.*, only the most recently accessed page is hot); the threshold in TPP [49] and MULTI-CLOCK [48] is two. Such static thresholds do not reflect workload characteristics, so placement decisions based on them are likely to be suboptimal. Although a few systems dynamically determine their thresholds, they do so in limited ways – *e.g.*, selecting victim pages for demotion in AutoTiering [32] or selecting candidate pages for promotion to throttle migration traffic in Tiering-0.8 [78]. TMTS [22] also employs a static criteria for promotion, while it uses an adaptable policy for demotion.

Criticality of hotness detection. The quality of hot and cold page detection critically affects the effectiveness of memory management systems.

One example is the multi-generational LRU (MGLRU) framework [85]. Linux kernel community recently replaced the conventional 2Q LRU with MGLRU. MGLRU makes a better page eviction decision using fine-grained, multi-generational page classification, boosting performance [63].

Another example is HeMem [68]. Although HeMem precisely tracks page access frequency using PEBS, it makes sub-optimal (often pathological) hotness decisions due to its pre-defined, static thresholds. Pages with access count beyond the static hot threshold are promoted to the fast tier. Whenever the access count of any page reaches the static cooling threshold, the access count of all pages is halved.

Figure 2 shows the number of hot pages classified in two memory-intensive applications: PageRank and XSBench. We describe our evaluation setup in §6. As Figure 2 clearly shows, hot page detection in HeMem is not optimal. When the size of identified hot pages is smaller than the fast tier's capacity (entire duration in PageRank and after 200s in XSBench), HeMem can place the hot pages in the fast tier, with the remaining space in the fast tier occupied by *arbitrary cold pages*. On the other hand, when the hot set size is greater than the fast tier's capacity (50s–180s in XSBench), an *arbitrary subset of hot pages* will be placed in the fast tier. One can try different threshold values to get the best result for applications, however, it is unlikely that a single threshold will work well across different workloads.

When to migrate. After classifying hot and cold pages, tiered memory systems migrate pages to their designated memory tiers. Prior works [14, 32, 49, 76, 78] identify hot pages upon page faults and migrate them to fast tier memory on the critical path. Doing so leads to extended blocking of the application during page fault, imposing overhead.

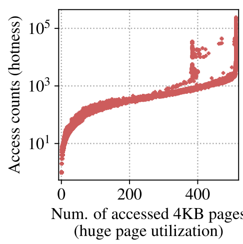
Insight #2. For optimal page placement, a tiered memory system should use accurate hotness metrics and adjust hotness criteria according to workload characteristics and memory configurations (*e.g.*, fast tier capacity). Also, it should migrate pages off the critical path (not in the page fault handler) to minimize additional latency.

2.3 Mitigating Address Translation Cost

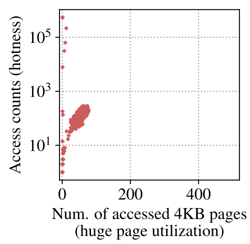
Address translation overhead is a well-known bottleneck in memory-intensive applications [25]. As the memory footprint grows, there is a higher chance of TLB misses, increasing address translation costs. Using huge pages is a standard practice to mitigate this by increasing the TLB reach and lowering the TLB miss penalty (*i.e.*, three levels in the page table instead of four).

However, huge pages make page migration between memory tiers more expensive [83, 84]. Moreover, memory access tracking techniques consider the entire huge page hot even when only a small fraction of subpages in a huge page are frequently accessed [4]. As a result, a small fraction of hot subpages in a huge page triggers the promotion of the entire huge page, wasting precious fast tier memory [14, 22, 86].

Analysis of huge page utilization. To investigate the access skew in subpage accesses across huge pages, we ran two memory-intensive benchmarks, Liblinear [45] and Silo [75]. We enabled Transparent Huge Page (THP) for huge page allocation and sampled memory accesses using PEBS. From the collected memory traces, we calculated *huge page utilization*, defined as the number of accessed subpages in a huge page. Since a 2MB huge page consists of 512 4KB subpages, utilization ranges from 0 to 512.



(a) Liblinear



(b) Silo

Figure 3. Hotness distribution to huge page utilization for Liblinear and Silo benchmarks. Each dot represents a huge page.

Figure 3 presents the access count distribution against the huge page utilization. When a huge page with high access count exhibits a high utilization (*e.g.*, Liblinear in Figure 3a), placing such a hot huge page in the fast tier can fully exploit fast memory access and address translation benefits. On the other hand, if there is no positive correlation between the access count and utilization of a huge page (*e.g.*, Silo in Figure 3b), only a few subpages in such a hot huge page are accessed. So migrating hot huge pages with low utilization would waste memory in the fast tier.

Insight #3. Our analysis demonstrates that *one page size does not fit all.* A tiered memory system should dynamically decide the appropriate page size according to page hotness and huge page utilization. To realize this, fine-grained access tracking is essential.

3 Memtis Design Overview

This section overviews Memtis as illustrated in Figure 4: (1) how to track memory accesses in a fine-grained and lightweight manner using hardware-based memory access sampling, (2) how to dynamically and precisely determine hot and cold pages considering the overall memory access frequency distribution, and (3) how to dynamically determine page size (huge page vs. base page) to reduce translation cost without wasting fast tier memory.

(1) Fine-grained, lightweight access tracking. Memtis samples memory accesses using PEBS. Since PEBS samples contain exact memory addresses (❶), Memtis can support fine-grained access tracking regardless of OS page size. `ksampled` – a Memtis-managed kernel background thread – processes the sampled addresses and updates memory access statistics in two different histograms, *page access histogram* and *emulated base page histogram* (❷, ❸). Memtis dynamically adjusts the memory access sampling frequency to ensure its CPU overhead is under a threshold (< 3%).

(2) Histogram-based hot set classification. Memtis maintains a *hotness distribution* of all allocated pages using a *page access histogram* of page access counts (❸). This histogram represents the number of distinct pages (y-axis) with access counts falling within a particular access count range (x-axis). Memtis utilizes the access histogram to know

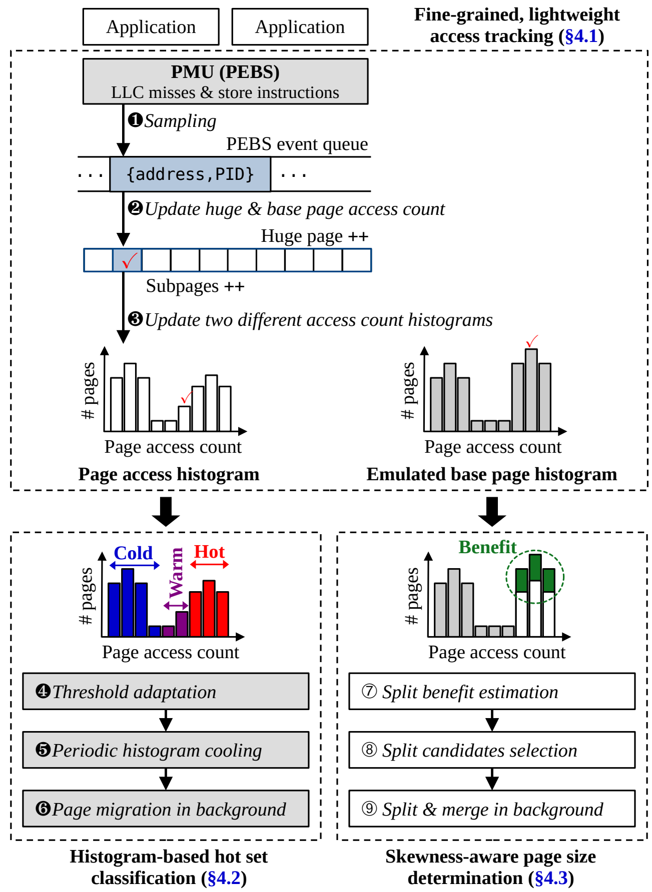


Figure 4. Overall architecture of Memtis.

the hotness distribution, so it can make the best tiering decisions, placing the hottest pages in the fast tier to minimize access latency. *As far as we know, Memtis is the first system leveraging page access frequency distribution to make optimal placement decisions in tiered memory systems.*

Memtis determines if a page is either *hot*, *cold*, or *warm* essentially based on its access count by adapting the threshold (❹). Memtis maintains the hot set size (highlighted red in Figure 4) close to the fast tier capacity so that the fast tier can accommodate all *hot pages. Cold pages* live in the capacity tier, and Memtis avoids migrating *warm pages* when the migration overhead would overshadow the benefit of lower access latency in the fast tier. Memtis maintains the freshness of the histograms and page statistics via *cooling* (❺), a process that halves the access count of all pages. This decreases old access counts exponentially to maintain the trend of page access frequency. Memtis performs page promotion/demotion in the background using a dedicated per-memory node migration thread (`kmigrated`) to avoid extending the critical path and performance slowdown (❻). *It is worth noting that the entire process of Memtis – including page access tracking, hotness classification, promotion, and demotion – is done in the background, never extends critical path.*

(3) Skewness-aware page size determination. Memtis uses Transparent Huge Pages (THP) in Linux by default to reduce address translation costs. However, as discussed in §2.3, a single page size does not work well for all workloads. For instance, when only a small fraction of subpages in a huge page are frequently accessed (*i.e.*, low huge page utilization as Figure 3b), it is better to break up such a huge page and migrate only the hot subpages to the fast tier.

Memtis detects such scenarios via *split benefit estimation*. Memtis estimates the maximum hit ratio when only base pages are employed using an *emulated base page histogram* and compares the estimated maximum hit ratio against the actual hit ratio obtained from the sampled PEBS records. This gap (⑦, the green part in Figure 4) approximates the potential benefit of a huge page split. If the potential benefit is large, Memtis chooses huge pages with *high access skew* in their subpages as split candidates (⑧). Then, it splinters the huge pages in the background and places each split subpage into the appropriate memory tier by referring to subpage access information maintained in the huge page (⑨). Since splitting a huge page is an expensive operation involving subpage migration and TLB shootdown, Memtis makes the split decision after observing long-term page access trends. *As far as we know, Memtis is the first system that dynamically chooses the page size according to subpage access skewness.*

4 Detailed Design of Memtis

We now elaborate on the Memtis design: fine-grained, lightweight access tracking (§4.1), histogram-based hot set classification with off-the-critical-path page migration (§4.2), and skewness-aware page size determination (§4.3).

4.1 Fine-grained, Lightweight Access Tracking

4.1.1 Sampling Memory Accesses Using PEBS. Memtis samples retired LLC load misses and retired store instructions using PEBS for fine-grained access tracking. It dynamically adjusts the sampling intervals to maximize the number of sampled events with bounded overheads. Memtis initially sets the sampling intervals to 200 and 100,000 for LLC load misses and store instructions, respectively. `ksampled` periodically calculates the exponential moving average of its CPU usage and adjusts the sampling intervals (using `__perf_event_period`) to meet the upper limit of its CPU usage (by default, 3% of a single core). `ksampled` uses hysteresis to prevent continual updates on the sampling period: it increments or decrements the period if the CPU usage and its upper limit are separated by 0.5%. We observed that, across all our evaluated benchmarks, `ksampled` only consumes 2.016% of a single CPU with 0.922% of performance overheads on average.

4.1.2 Page Access Metadata. Memtis maintains *hotness*, *utilization*, and *skewness* for memory pages. We calculate the hotness for all page types (*i.e.*, base page, huge page, subpage in a huge page). In contrast, utilization and skewness are maintained only for huge pages (§4.3.2).

Hotness represents the access trend using the exponential moving average (EMA) of page access count. The hotness factor (H_i) for page i is defined by the page's access count (C_i) and page type as follows:

$$H_i = \begin{cases} C_i & if \text{ page } i \text{ is a huge page} \\ C_i \times nr\_subpages & if \text{ page } i \text{ is a base page} \end{cases} \quad (1)$$

where $nr\_subpages$ is the number of subpages constituting a huge page (*i.e.*, 512 in x86). Since a huge page is $nr\_subpages$ times more likely to be accessed than a base page, we compensate for a base page's hotness using $C_i \times nr\_subpages$. `ksampled` increments C_i of page i by one for each PEBS sample. Note that C_i (and H_i) will be periodically halved during the cooling process (§4.2.2) for calculating EMA.

4.1.3 Page Access Histogram. A *page access histogram* consists of 16 bins by default. Each bin represents a specific range of hotness factor (H_i) following an exponential scale; n-th bin has the range of hotness factor $[2^n, 2^{n+1})$, and the last bin has no upper bound on it. The value (y-axis) of each bin denotes the number of distinct pages (counting at 4KB granularity) in the hotness range.

Our exponential scale bins are compact (*i.e.*, 16 bins × 8-byte counter = 128 bytes). Also, the exponential scale simplifies our histogram management in the cooling process since cooling halves the access counts (refer to details in §4.2.2). Most importantly, it matches well with the non-linear (often exponential, *e.g.*, Zipf [3] or Pareto [7]) nature of page accesses – hot pages have several orders of magnitude more accesses than warm or cold pages. Such non-linear page access frequency is difficult to be captured with an equally-divided bin design.

Updating the page access histogram is very efficient. Whenever `ksampled` updates a page's hotness factor (H_i), it checks if the new hotness factor falls into a different bin. Suppose that originally H_i falls into bin 6; after incrementing C_i, if the new H_i falls into bin 7, `ksampled` decrements the page count in bin 6 and increments the page count in bin 7.

Note that Memtis manages two histograms – page access histogram and emulated base page histogram (or *base page histogram* in short) illustrated in Figure 4. Memtis uses the page access histogram to determine hot pages (§4.2) and the base page histogram to determine page sizes (§4.3).

4.2 Histogram-based Hot Set Classification

Memtis periodically adapts the threshold of hot, warm, and cold pages in the page access histogram by considering the hotness distribution (§4.2.1). Also, it cools down the histogram to calculate EMA and capture trends of page access frequency (§4.2.2). Lastly, it quickly moves pages to the appropriate memory tier (§4.2.3) in the background.

4.2.1 Dynamic Threshold Adaptation. Memtis maintains hot, warm, and cold thresholds denoted by T_{hot}, T_{warm}, and T_{cold}, respectively. These thresholds are the bin indices of the page access histogram. If page i's bin index (B_i) is greater than or equal to T_{hot} (*i.e.*, hot page, $T_{hot} \leq B_i$), it will be placed in the fast tier. Similarly, if $B_i \leq T_{cold}$ (*i.e.*, cold page), it will be moved to the capacity tier. Otherwise (*i.e.*, warm page, $T_{cold} < B_i \leq T_{warm}$), it is hard to decisively determine the page's hotness, so MEMTIS does not migrate such a page and leaves it where it is.

Determining thresholds. `ksampled` periodically adjusts the thresholds based on the page access distribution encoded in the histogram. As shown in Algorithm 1, `ksampled` expands T_{hot} as much as possible to hold the hottest pages in higher bins before overflowing the fast tier (Lines 2-6).

The identified hot set size (s) could be smaller than the fast tier capacity (MS_{fast}) since MEMTIS organizes histogram bins on an exponential scale. If the identified hot set size is close enough to the fast tier capacity ($s > MS_{fast} \times \alpha$), MEMTIS can fully utilize fast tier and the small fraction of unused memory can be reserved for future page allocations and promotions (Lines 7-8). We set α to 0.9 empirically. Note that MEMTIS allocates pages on the fast tier whenever available.

When the size of the identified hot pages is not close enough to the fast tier capacity ($\leq$ 90% when $\alpha = 0.9$), MEMTIS might retain arbitrary cold pages in fast tier memory, similar to prior work (see PageRank in Figure 2). In such a case, some soon-to-be hot pages could be demoted, but would subsequently get promoted back shortly to the fast tier after becoming hot. This generates unnecessary migration traffic and overshadows the benefit of our dynamic page placement. To remedy this problem, we introduce a *warm threshold*, T_{warm}. When $s \leq MS_{fast} \times \alpha$, T_{warm} is set to $T_{hot} - 1$ (Line 10). MEMTIS employs T_{warm} to exclude pages whose hotnesses are close to the hot threshold (*i.e.*, warm pages) in the fast tier from demotion candidates. Nevertheless, in a case where there are no cold pages in the fast tier and MEMTIS needs to secure free space for newly allocated pages or hot pages to be promoted, MEMTIS proceeds to demote warm pages. T_{cold} is set to $T_{warm} - 1$ once T_{warm} is calculated (Line 12). Initially, T_{hot}, T_{warm}, and T_{cold} are set to 1, 1, and 0, respectively. Initial hotness for newly allocated pages is set to the current hotness threshold (T_{hot}) to prevent them from being immediately chosen as demotion candidates.

Threshold adaptation interval. Given the purpose of the threshold adaptation, it is adequate to initiate the adaptation process when the total capacity of sampled pages is similar to the fast tier capacity. Moreover, while it is acceptable for the interval to be short, selecting an excessively lengthy interval could negatively affect performance. Based on this rationale, MEMTIS performs the adaptation for every 100,000 sampled events. We show a sensitivity study on the adaptation interval in §6.3.4.

4.2.2 Periodic Histogram Cooling.

The memory access trend of a page could change over time. Hence, MEMTIS should decay the impact of old accesses and give more weight

Algorithm 1: Dynamic adaptation of thresholds.

MS_{fast}: size of fast tier memory
HS_b: total size of pages belonging to histogram bin b
max: maximum bin index of the historgram (*i.e.*, 15)

// Calculate hot threshold, T_{hot}
1 $s = 0, b = max$
2 **while** $b \geq 0$ *or* $(s + HS_b \leq MS_{fast})$ **do**
3     $s = s + HS_b$
4     $b = b - 1$
5 **end**
6 $T_{hot} = b + 1$
// Calculate warm threshold, T_{warm}
7 **if** $s > MS_{fast} \times \alpha$ **then**
8     $T_{warm} = T_{hot}$
9 **else**
10     $T_{warm} = T_{hot} - 1$
11 **endif**
// Calculate cold threshold, T_{cold}
12 $T_{cold} = T_{warm} - 1$

to recent accesses. To this end, MEMTIS periodically halves every page's access count (C_i). This cooling process is indeed calculating the exponential moving average (EMA) of H_i with a decay factor of 0.5. Since we configure the histogram bins exponentially, cooling requires merely shifting the value of each bin in the histogram one bin to its left. The hot and warm thresholds are updated based on the shifted histogram. Then, `kmigrated` scans the page lists and halves each page's access count. For huge pages, `kmigrated` also performs cooling for every subpage's metadata. If a page has the highest bin index (*i.e.*, $B_i = max$), B_i could be unchanged after cooling, so MEMTIS checks the bin index of cooled pages and corrects the histogram if necessary.

MEMTIS performs cooling based on the number of sampled memory accesses. The cooling period has to be sufficiently large as it determines the total number of sampled memory accesses reflected in the histogram. MEMTIS performs cooling for every two million records, which is large enough considering the gigabyte-scale fast tier memory size.

4.2.3 Page Migration in the Background.

MEMTIS creates a `kmigrated` kernel thread for each memory tier and performs all promotion and demotion operations in the background. MEMTIS maintains a *promotion list* for the capacity tier and a *demotion list* for the fast tier. The promotion list contains only the hot pages while the demotion list has both warm and cold pages. Whenever `ksampled` processes a memory access sample, it compares the page's hotness factor (H_i) to T_{hot} and moves the page to the promotion list if it is hot. When `kmigrated` performs cooling by halving a page's access count, some pages could become warm or cold, in which case `kmigrated` moves them to the demotion list.

`kmigrated` is woken up periodically (500ms). The capacity tier `kmigrated` checks if there are hot pages in the capacity tier and free space is available in the fast tier. If so, it promotes hot pages in the capacity tier to the fast tier.

When available memory in the fast tier falls below a free-space threshold, the fast tier `kmigrated` starts demotion. We set the free-space threshold to 2% of the fast tier size for future page allocations and promotions. `kmigrated` chooses victim pages in the demotion list of the fast tier. It first demotes cold pages in the demotion list ($H_i \leq T_{cold}$) to the capacity tier. If enough free space is secured after demoting cold pages, `kmigrated` stops. Otherwise, it demotes warm pages to the capacity tier until enough free space is acquired. Hence, MEMTIS is able to keep as many warm pages as possible in the fast tier. Note that MEMTIS treats all pages within a given hotness group as equivalent, so there is no strict demotion order among pages in the same group (*i.e.*, warm or cold).

4.3 Skewness-aware Page Size Determination

Huge pages are not always beneficial, due to their memory bloat and access skew in the fast tier. Also, splitting a huge page is very expensive, involving page table updates and TLB shootdown, so aggressively splitting huge pages can do more harm than good. Hence, MEMTIS first estimates the maximum benefit of huge page split based on the long-term page access history (§4.3.1). Then it determines how many and what huge pages need to be split (§4.3.2), and finally performs page type conversion in the background (§4.3.3).

4.3.1 Estimating the Benefit of Huge Page Split. If only base pages were used and the hottest base pages were placed in the fast tier, we avoid wasting fast tier memory due to the low utilization of huge pages. *eHR* is the estimated hit ratio when we exclusively use base pages. MEMTIS compares the actual measured hit ratio (*rHR*) and the estimated hit ratio (*eHR*) of the fast tier memory. Since *rHR* characterizes the current utilization of the fast tier, if *rHR* is much lower than *eHR*, there is room to increase the hit ratio by splitting skewed huge pages and filling the fast tier memory with hot base pages.

Calculating *rHR* and *eHR*. When `ksampled` processes a sampled memory access, it checks if the address falls into the fast tier. If so, `ksampled` increments *rHR*. MEMTIS maintains an *emulated base page histogram* to estimate *eHR*. The base page histogram manages a memory access distribution at a 4KB page granularity (including subpages in huge pages), regardless of actual OS-managed page size. The base page histogram is updated and cooled in the same way as the regular page access histogram. Using the base page histogram, MEMTIS calculates its thresholds – say T^{BP}_{hot} – using Algorithm 1. Whenever `ksampled` updates page metadata, it checks if a corresponding 4KB (sub- or base-) page is hotter than T^{BP}_{hot}. If so, MEMTIS increments the hit count of *eHR*.

Triggering huge page split. MEMTIS performs benefit estimation when a large number of memory accesses is sampled to make a decision based on the long-term, stable memory access trends. It calculates *eHR* whenever the number of sampled records exceeds a quarter of the total number of allocated pages (*e.g.*, at least every 1 million records for 4GB ($2^{20}\times$ 4KB) memory size). Moreover, MEMTIS triggers the huge page split procedure only when the potential benefit ($eHR - rHR$) is sufficiently large (5% or higher).

4.3.2 Split Candidates Selection. When the expected benefit is sufficiently large, MEMTIS decides the number of huge pages to be split as follows:

$$N_s = min((eHR - rHR) \times \frac{\Delta L}{L_{fast}} \times \frac{nr\_samples \times \beta}{avg\_samples\_hp}, \frac{nr\_samples}{avg\_samples\_hp}) \quad (2)$$

MEMTIS aggressively splits more huge pages when the expected benefit is higher, the latency gap between the two memory tiers is larger, and when more huge pages are accessed. Specifically, when the expected access benefit ($eHR - rHR$) is higher, MEMTIS splits more huge pages. Also, when the latency gap (ΔL) between the capacity tier (L_{cap}) and the fast tier (L_{fast}) is larger, huge pages are split more aggressively ($\Delta L / L_{fast}$). The number of huge pages split (N_s) should be proportional to the number of distinct huge pages accessed in a benefit estimation interval. To approximate the number of distinct huge pages accessed in a benefit estimation interval, MEMTIS uses the total number of samples (*nr_samples*) and the average of sampled accesses within a huge page (*avg_samples_hp*) with a scale factor ($\beta = 0.4$). Our approximation is trivial to calculate without incurring overheads of precisely managing the set of accessed huge pages. Also, N_s cannot exceed the number of distinct huge pages ($min(...)$).

Calculating the skewness of a huge page. Upon determining the number of huge pages to split, MEMTIS decides which huge pages to split based on the subpage access skew. We define the *skewness factor* of a huge page i as follows:

$$S_i = \frac{\sum_{j=0}^{nr\_subpages} H_{ij}^2}{U_i^2} \quad (3)$$

where U_i is the utilization factor and H_{ij} is the hotness factor of the j-th subpage of huge page i. The utilization factor U_i indicates the number of hot subpages in a huge page ($T^{BP}_{hot} \leq H_{ij}$). *The skewness factor S_i gets higher when the huge page's utilization (U_i^2) decreases and its subpage hotness factors ($\sum H_{ij}^2$) rise.* We squared U_i and H_{ij} because such non-linear, monotonic-increasing transformation helps to distinguish a skewed access pattern from a uniformly hot access pattern.

Choosing top-N_s highly skewed huge pages. `kmigrated` updates the skewness factor of each huge page at every cooling interval since cooling scans entire pages, and its period is long enough to capture long-term access behavior. To efficiently choose the top-N_s highly skewed pages, MEMTIS builds an array of skewness factors during cooling, where each entry contains a list of huge pages in a skewness range. Then, it chooses the top-N_s huge pages from the array.

4.3.3 Page Type Conversion in the Background.

The chosen split candidates are then moved to a split queue and `kmigrated` splinters the huge pages in the split queue. Specifically, it classifies subpages within each split candidate into hot and cold base pages using their subpage hotness factors (H_{ij}). When Memtis breaks up huge pages, it unmaps and frees all-zero (never updated) subpages to reduce memory usage. Finally, it migrates each subpage into the appropriate memory tier. Coalescing base pages into a huge page is also expensive and it requires to consider the potential hotness and access skewness of the newly coalesced huge page. Thus, Memtis coalesces base pages only when all constituent base pages are hot. Coalescing rarely happens because Memtis enables transparent huge page (THP) and splinters allocated huge pages conservatively.

5 Implementation

We implemented Memtis in Linux Kernel v5.15.19. The total changed lines of code (LoC) is 5,166.

We utilize the Linux kernel's `compound_page` structure to manage access metadata for huge pages. The `compound_page` structure consists of 512 `struct pages`, each of which is the metadata for a 4KB physical page frame. Linux kernel uses the first three `struct pages` (0–2) for huge page information itself while the rest are not used. We leverage the unused `struct pages` (3–131) to store the huge page access metadata (in 3) and subpage access metadata (in 4–131). *In this way, Memtis manages the metadata for huge pages and their subpages without any additional memory overhead.*

Storing access metadata of a base page is a bit more tricky because there is no unused (padding) space in the `struct page` used for page cache and anonymous pages. Instead of adding an extra field in the `struct page`, which makes the size bigger than a cache line (64B), we leverage a PTE page frame of a page table. Since `struct page` for a PTE page frame has an unused, 8-byte padding space, we re-purposed this unused field as a pointer to a 4KB metadata page, which contains 512 metadata entries for 512 base pages. *In the worst case, where all pages are base pages, the memory overhead of Memtis is at most 0.195% of the memory footprint.*

6 Evaluation

We evaluate Memtis by answering the following questions:

- How does Memtis perform with real-world memory-intensive applications compared to state-of-the-art memory tiering systems? (§6.2)
- How effective is Memtis's optimization? (§6.3)
- Would Memtis still be effective on CXL-based tiered memory systems? (§6.4)

6.1 Evaluation Methodology

Hardware setup. We evaluated Memtis on a dual-socket server equipped with Intel Xeon Gold 5218R @2.1 GHz processors (20 cores), where each socket has 6×16GB DDR4 DRAM and 6×128GB Intel Optane DCPMM. To demonstrate Memtis's generality and robustness, we use two different tiered memory settings: 1) DRAM + NVM (Optane DCPMM, load: 300ns) and 2) DRAM + emulated CXL memory (cross-NUMA DRAM with increased latency, load: 177ns). Similar to prior works [49, 68, 84], we use a single socket for our evaluations to avoid NUMA effects, which are out of scope of this paper.

Benchmark	RSS	RHP	Description
Graph500	66.3 GB	99.9%	Generation and search of large graphs [52]
PageRank	12.3 GB	99.9%	Compute the PageRank score of a graph [12] (Twitter dataset [37])
XSBench	63.4 GB	100%	Computational kernel of the Monte Carlo neutron transport algorithm [74]
Liblinear	67.9 GB	99.9%	Linear classification of a large data set (KDD12 dataset) [45]
Silo	58.1 GB	97.4%	In-memory database engine [75]
Btree	38.3 GB	75.2%	In-memory index lookup benchmark [1]
603.bwaves	11.1 GB	99.5%	Explosion modeling in SPEC CPU 2017 [20]
654.roms	10.3 GB	96.6%	Regional ocean modeling in SPEC CPU 2017 [20]

RSS: Resident Set Size   RHP: Ratio of Huge Pages Allocated with THP

Table 2. Benchmark characteristics.

Benchmark	Over-allocation size	Benchmark	Over-allocation size
Graph500	60 MB	Silo	1400 MB
PageRank	500 MB	Btree	9800 MB
XSBench	420 MB	603.bwaves	1900 MB
Liblinear	90 MB	654.roms	900 MB

Table 3. Over-allocation sizes of HeMem.

Benchmarks. We choose eight representative memory-intensive applications, including graph processing (Graph500, PageRank), an HPC workload (XSBench), machine learning (Liblinear), an in-memory database engine (Silo), an in-memory index lookup (Btree), and SPEC CPU 2017 (603.bwaves, 654.roms). These benchmarks are widely used to evaluate tiered memory systems [36, 68], huge page management [26, 38, 57, 67], and large memory servers [2, 5, 56]. Note that we choose only two benchmarks from SPEC CPU 2017 since they are the only ones that consume more than 10GB memory. We ran all these benchmarks with 20 threads, stressing all CPU cores, to account for any CPU overheads from sampling and migration of Memtis. Table 2 shows the detailed benchmark description, including memory size and the ratio of huge pages allocated.

Comparison targets. We compare Memtis to six state-of-the-art systems: AutoNUMA [76], AutoTiering [32], Tiering-0.8 [78], TPP [49], Nimble [84], and HeMem [68]. We report the relative performance normalized to the performance of the all-NVM case with THP enabled, where each benchmark runs entirely on the capacity tier for easy comparison.

Tiering configurations. We configured the ratio of fast to capacity tier memory size as 1:2, 1:8, and 1:16. In the 1:2 configuration, the fast tier size is set to 33% (1/3) of the resident set size (RSS) for each benchmark (shown in Table 2), while in the 1:16 configuration, it is reduced to 5.9% (1/17). To control the size of the fast tier, we used a memory cgroup interface for MEMTIS and Nimble. For AutoNUMA, AutoTiering, Tiering-0.8, and TPP, we changed the kernel boot argument (memmap GRUB option [65]) to limit the fast tier size. We configured HeMem's fast tier size at compile time. However, HeMem always places small allocations in the fast tier, so the actual fast tier usage could be larger than the configured size. For a fair comparison, we measured such small allocations on the fast tier (denoted as *over-allocation size*), as shown in Table 3, and accounted for them by reducing the configured fast tier size by the amounts in Table 3 when compiling the HeMem code. This specific setting for HeMem was consistently maintained throughout our evaluation, unless explicitly stated otherwise.

6.2 Performance Comparison

Figure 5 shows the performance comparison of tiered memory systems when using NVM as the capacity tier. MEMTIS performs the best in almost all cases (23/24), and its geomean performance is 33.6% higher than the second-best system. Although TPP has the second-best performance in 14 out of 24 cases (8 benchmarks each of which has 3 memory configurations), it also shows the worst performance in PageRank at the 1:2 configuration. Our experimental results show that MEMTIS performs well under various memory settings and access patterns. We now present the detailed analysis for each benchmark.

6.2.1 Graph Processing: Graph500 and PageRank. Graph500 [52] generates a graph and conducts a BFS search for 64 keys. Similarly, the GAP benchmark [12] uses the Twitter dataset [37] to generate the graph and runs 20 iterations of the PageRank algorithm. Both benchmarks access a large memory region frequently during the graph generation. During the search phase, they frequently access a small memory region. Also, their huge page utilization is high. MEMTIS can differentiate the page access frequencies in the generation phase and detect hot pages in the search phase in a timely fashion. As a result, MEMTIS outperforms the second-best system by 16.3%–17.7% for Graph500 and 10.2%–47.9% for PageRank, as shown in Figure 5(a) and Figure 5(b).

The gain of other systems heavily depends on memory access patterns and configurations. For example, TPP is the second-best in Graph500 but not in PageRank. HeMem also shows lower performance due to its static thresholds and high CPU usage (≈ 100%) of the sampling thread.

6.2.2 HPC Workload: XSBench. Our analysis reveals that XSBench has a very skewed hot memory region allocated at an early stage. MEMTIS quickly identifies the entire hot memory region. Then it places all of them in the fast tier using huge pages. As a result of precise hot set detection and usage of huge pages, MEMTIS, even under the 1:16 setting, outperforms others except AutoNUMA in the 1:2 setting.

Existing systems excluding AutoNUMA actively demote pages in the fast tier to make room for future allocations and promotions, leading to the demotion of huge pages in

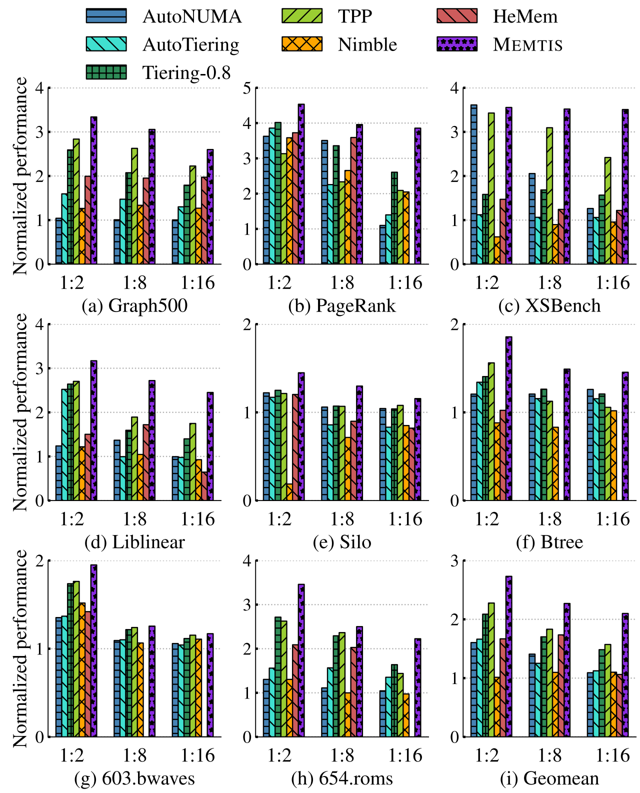


Figure 5. The performance comparison of MEMTIS against other systems under various tiering settings (fast tier vs. capacity tier = 1:2, 1:8, 1:16). We used NVM as capacity tier memory and results solely using NVM (with THP) as baseline performance. With HeMem, we failed to run several benchmarks in the 1:8 and 1:16 settings as we could not enforce the fast tier size due to excessive small-sized allocations by the benchmark. *MEMTIS performs best in most cases (23/24) and outperforms the second-best systems by 33.6% on average (geomean).*

an eager manner. Thus, their performance depends on how quickly they promote hot pages again. HeMem classifies only 2–30MB as hot data (shown in Figure 2), thereby underutilizing the rest of the fast tier. Ironically, AutoNUMA lacks the demotion feature, so it cannot demote the early allocated hot pages, thus showing better performance in the 1:2 configuration.

6.2.3 Machine Learning: Liblinear. We ran the Liblinear benchmark [45] with KDD12 dataset. As Figure 3a shows, hot huge pages of Liblinear have high utilization. MEMTIS preferentially places hot pages with high utilization in the fast tier, resulting in high hit ratios ranging from 96.39% to 99.99%. As a result, MEMTIS outperforms the second-best by 17.3%–43.7%, as shown in Figure 5(d).

TPP shows the second-best performance in Liblinear. TPP identifies more hot pages than the fast tier size for 1:16 and 1:8 configurations due to its coarse-grained, 2Q LRU-based hot page classification. So, it could not place the hottest pages in the fast tier while continuously migrating pages between memory tiers. Even though the fast tier can hold additional hot pages in the 1:2 setting, TPP could not immediately detect them due to its unscalable page table scanning.

6.2.4 In-Memory Database Engine: Silo.

We ran Silo [75] using the YCSB-C workload [18] following a Zipfian distribution. We populated 400 million key-value pairs and performed 15 billion lookup operations. The key and value sizes are 64B and 100B, respectively. Silo frequently accesses only 5–15% of subpages in a huge page, as analyzed in Figure 3b. With such a low huge page utilization and high skewness, it is hard to fully harness the fast tier due to underutilized cold subpages in a huge page.

Figure 5(e) shows the superior performance of MEMTIS, where it outperforms the second-best by 15.9%, 21.2%, and 7.1% for 1:2, 1:8, and 1:16, respectively. The performance gain comes from our skewness-aware huge page split. MEMTIS effectively finds skewed hot huge pages, splits them, and migrates their hot subpages to the fast tier. The RSS remains unchanged after the split since there is no memory bloat due to huge pages (*i.e.*, all cold subpages are accessed).

Nimble generates massive page migration (56.43× more than MEMTIS), resulting in poor performance. Nimble classifies pages as hot if they are accessed just once during the scan interval. Silo accesses a lot of huge pages, so the identified hot set is much larger than the fast tier size.

6.2.5 In-Memory Index Lookup: Btree.

We measured the lookup performance of an in-memory Btree [1]. We populated the Btree with 157 million key-value pairs and performed 8 billion random lookup operations. The key and value sizes are 8B and 16B, respectively. Our analysis indicates that this benchmark has skewed access patterns and low huge page utilization (mostly 8.3–12.5%). The root cause of low utilization is memory bloat, a notorious problem of huge pages wasting memory [38, 57]. In practice, using huge pages improves the Btree performance by 12.5% when we run it entirely on the fast tier, but it severely increases the RSS from 15.2GB to 38.3GB.

Figure 5(f) shows that MEMTIS outperforms the second-best system by 15.5%–18.9%. In particular, the performance benefit of MEMTIS mostly comes from skewness-aware huge page split, as will be shown in Figure 11. Our huge page split reduces the RSS under the 1:2, 1:8, and 1:16 configurations from 38.3GB to 36.95GB, 27.2GB, and 20.9GB, respectively.

6.2.6 SPEC CPU 2017: 603.bwaves and 654.roms.

We ran 603.bwaves and 654.roms with reference (`ref`) input size. As shown in Figure 5(g) and Figure 5(h), MEMTIS outperforms the second-best system by 1.3%–10.6% in 603.bwaves and 5.7%–35.9% in 654.roms.

603.bwaves allocates short-lived and long-lived data. Tiering-0.8, TPP, and MEMTIS allocate short-lived data to the free space in the fast tier reserved for new allocations. AutoTiering uses the background thread for demotion to reserve free pages in the fast tier but it utilizes them only for promotion. Thus, it always allocates short-lived data to the capacity tier, thereby showing lower performance.

6.2.7 Scalability.

We evaluate MEMTIS by increasing the RSS of Graph500 from 128GB to 690GB. The fast tier size is 64GB in all experiments. Figure 6 shows that MEMTIS outperforms the second-best by 8.1%–60.5%, as the RSS increases. HeMem shows the second-best performance when the RSS is 336GB and 690GB. These results clearly demonstrate the effectiveness of PEBS and the importance of precise hotness classification.

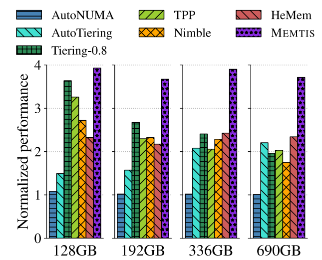


Figure 6. Performance comparison under varying memory sizes.

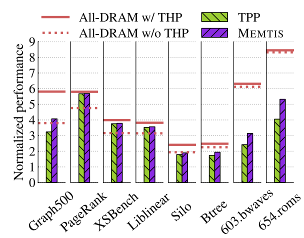


Figure 7. The performance of MEMTIS and TPP under the 2:1 configuration.

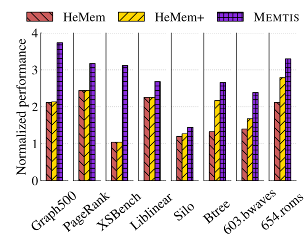


Figure 8. The performance of MEMTIS and HeMem with 16 threads.

6.2.8 2:1 Configuration.

We also evaluate MEMTIS on a 2:1 configuration, which is Meta's default production target environment [49]. TPP was originally designed for this environment. Figure 7 presents the performance of TPP and MEMTIS, along with the all-DRAM performance with or without using THP for reference. Although sampling-based memory access tracking has inherent limitations in detecting rarely accessed pages, MEMTIS is still effective even under the 2:1 configuration and exhibits comparable performance to the all-DRAM cases, except for the SPEC benchmarks. MEMTIS outperforms TPP by 6.1%–33.3% when the capacity of sampled pages is larger than the fast tier capacity. MEMTIS shows similar performance to TPP when there is a small set of explicit hot pages relative to the fast tier capacity (PageRank, XSBench, and Liblinear). In this case, MEMTIS promotes pages to the fast tier as soon as they are sampled once, resulting in a high access ratio to the fast tier memory (> 99.5%).

6.2.9 Detailed Comparison to HeMem.

We compare the performance of MEMTIS against HeMem on HeMem's most favorable settings. First, the performance of HeMem could be affected by CPU contention caused by its sampling threads. Thus, we conducted all experiments with 16 threads, leaving the other cores available for HeMem's service threads. Second, the configured fast tier size of HeMem

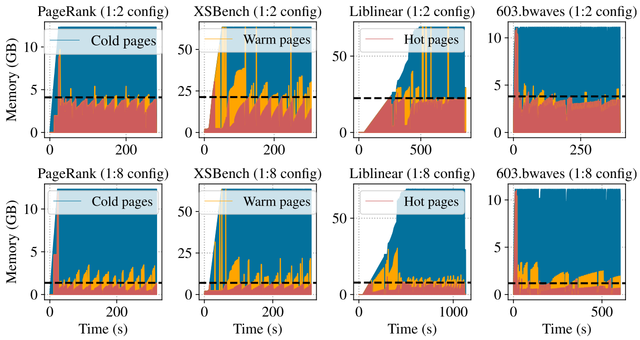


Figure 9. Amount of hot, warm, and cold data identified by MEMTIS in two tiering settings. The dashed line indicates the fast tier size.

is smaller than that of other systems, as mentioned in §6.1, and this could affect the performance of HeMem. So, we additionally measure the performance of HeMem under scenarios where HeMem's configured fast tier size is same as that of MEMTIS. In this case, HeMem consumes more fast tier capacity than MEMTIS by the amounts in Table 3. This performance is labeled as HeMem+ in Figure 8. Note that the experiments are performed under the 1:2 configuration.

The results clearly indicate that MEMTIS consistently outperforms HeMem when no CPU contention exists. The primary factor contributing to HeMem's performance degradation lies in its reliance on page classification based on static thresholds. The degradation is similarly observed in HeMem+. For instance, in PageRank, HeMem+ harnesses additional fast tier capacity (over-allocated one) of 500MB compared to HeMem but its performance does not show improvement. This is because HeMem+ wastes a part of fast tier memory with arbitrary cold pages, as shown in Figure 2. Interestingly, MEMTIS also achieves higher performance than HeMem+ in Btree, which entails an over-allocation size of 9800MB. We measured a version of MEMTIS employing only the histogram-based hot set classification, and found that it performed only 1.4% below HeMem+. Further enabling the skewness-aware huge page split leads to an impressive performance enhancement of 22.6% over HeMem+.

6.3 Understanding MEMTIS Performance

This section analyzes the impact of each optimization technique in MEMTIS. In particular, we discuss 1) the accuracy of access distribution-based hot set classification (§6.3.1), 2) how much page migration traffic is reduced using the warm set and splitting the huge pages (§6.3.2), 3) how huge page splits affect performance and memory footprint (§6.3.3), 4) the sensitivity of MEMTIS to threshold adaptation and cooling intervals (§6.3.4), and 5) the overheads of PEBS-based access tracking (§6.3.5).

6.3.1 Effectiveness of Hot Set Classification. Figure 9 shows the amount of hot, warm, and cold memory identified by MEMTIS at runtime. Overall, the identified hot set size is very close to the fast tier size (denoted by the black dashed line). MEMTIS identifies a hot set as large as the fast tier size using the page access histogram. The identified hot set could be below the fast tier size according to the histogram status with the warm pages filling the remaining fast tier. Although it is possible that the hot set temporarily exceeds the fast tier size since many warm/cold pages could become hot before adjusting T_{hot}, MEMTIS can quickly recover the hot set. Such hot set management is impossible without considering the overall memory access distribution.

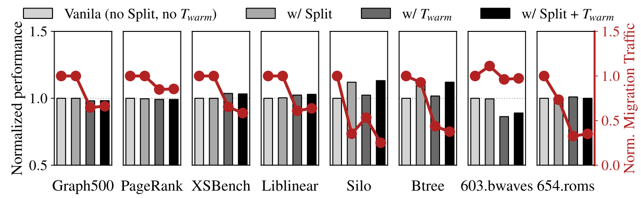


Figure 10. Impact of the use of warm set and huge page split on performance and memory migration traffic.

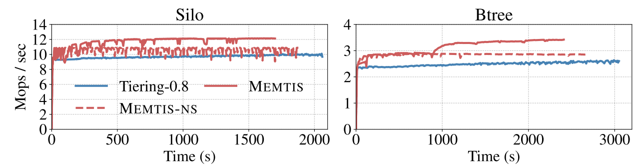


Figure 11. Performance of Silo and Btree over time. We ran MEMTIS with and without huge page split (MEMTIS vs. MEMTIS-NS). We also provide the performance of the second-best system (Tiering-0.8).

6.3.2 Reducing Memory Migration with Warm Set. MEMTIS effectively reduces memory migration traffic. Since warm pages could be getting cooler or hotter, MEMTIS can reduce significant migration traffic by not migrating such warm pages (2.7%–64.8% as in shown Figure 10). In addition, splitting huge pages somewhat reduces migration traffic, owing to the migration of smaller-sized pages. However, using the warm set degrades the performance in 603.bwaves; a large warm set makes it difficult to quickly reserve free space in the fast tier, and hence, a part of the short-lived allocations are handled in the capacity tier.

6.3.3 Impact of Huge Page Split. Figure 11 shows the performance of Silo and Btree benchmarks over time in the 1:8 configuration. Our skewness-based huge page split improves the overall performance by 10.6% for Silo and 10.4% for Btree (MEMTIS vs. MEMTIS-NS, where NS stands for no split). For Silo, MEMTIS detects the highly skewed huge pages in the fast tier at about 80s and starts splintering them. After a small performance dip right after the split, it quickly surpasses other works by detecting hot subpages and migrating them to fast tier memory. For Btree, where huge pages cause

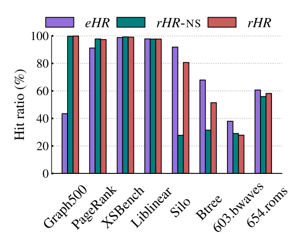
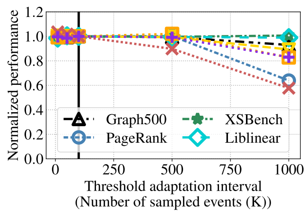
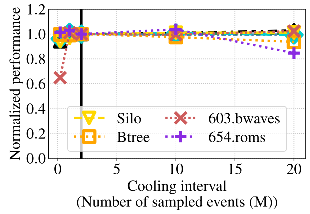


Figure 12. Fast tier hit ratios.



(a) Threshold adaptation intervals



(b) Cooling intervals

Figure 13. Sensitivity results for both threshold adaptation intervals and cooling intervals in the 2:1 configuration, normalized by the performance of the default setting. Note that the black vertical line indicates the default parameter value.

severe memory bloat, Memtis detects the skewed huge pages and begins splitting them near 800s. This improves throughput by up to 19.9% (at 2410s) and reduces RSS by 28.96% (38.3GB→27.2GB as discussed in §6.2.5).

Figure 12 compares three types of hit ratios in the 1:8 configuration: 1) *eHR* – the estimated hit ratio when only base pages are used, 2) *rHR* – the actual hit ratio with our huge page split, and 3) *rHR*-ns – the actual hit ratio of Memtis-ns without using our huge page split. Silo and Btree exhibit a big gap between *eHR* and *rHR*-ns, at 64.1% and 36.42%, respectively. Memtis splinters huge pages, improving the hit ratio (*rHR*) by 52.91% for Silo and 19.92% for Btree. Huge page split does not result in performance improvements in 654.roms, but it improves the hit ratio by 2.25% and reduces page migration by 26.6%. Memtis has very low *rHR* in 603.bwaves, which repeatedly allocates and frees short-lived data. Since Memtis always tries to secure some free space in the fast tier for new allocations, the repetitive allocations of short-lived data lead to frequent demotions of hot pages. Thus, huge page split does not increase *rHR* in this case. Note that *eHR* could be lower than *rHR* as in Graph500 and PageRank, when there is no access skew and/or strong spatial locality in the huge pages. In such cases, there is no need to split huge pages.

6.3.4 Parameter Sensitivity.

To assess the sensitivity of Memtis to threshold adaptation and cooling intervals, we conduct a sensitivity study by varying them from one-tenth of the default interval to ten times that. Note that each interval is represented by the number of sampled events collected under dynamically adjusted sampling rates, ranging from one sample every 200 to 1400 underlying events. As shown in Figure 13, Memtis shows a robust insensitivity to changes in both intervals except for the case of a 1M adaptation interval (*i.e.*, ten times the default adaptation interval). An excessively extended interval undermines the efficacy of our histogram-based hot set classification method, particularly when dealing with applications possessing small resident set sizes. In such scenarios, the fast tier capacities assigned to these applications are also quite limited, causing the hot set size identified over the extended interval to potentially exceed the fast tier size. In this case, the fast tier is filled with an arbitrary set of hot pages, instead of hottest ones.

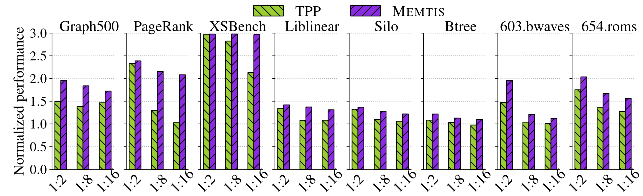


Figure 14. Performance comparison between Memtis and TPP [49] using (emulated) CXL memory as capacity tier.

6.3.5 The Overheads of Access Tracking.

`ksampled` dynamically adjusts the PEBS sampling interval based on its CPU usage. For example, `ksampled` increases its period from once for every 200 events to that for every 1400 events in 654.roms to limit its CPU usage. On the other hand, `ksampled` keeps its initial period in 603.bwaves because its CPU usage is always lower than the upper limit (3%). As a result, the average CPU usage of `ksampled` is only 2.016% of a single CPU (3.0% maximum). Accordingly, its performance implications on evaluated benchmarks are also modest: 0.922% on average and up to 2.384%.

6.4 Performance with CXL Memory

To explore how Memtis will work with upcoming CXL memory, we ran with an emulated CXL memory as the capacity tier.[2] We emulated CXL memory using a remote NUMA node and decreased the cross-NUMA interconnect frequency. We set the access latency of emulated CXL memory to 177ns since previous studies report that CXL adds 70-90ns to local memory access [42]. Note that we only consider directly-attached CXL memory supported by CXL 1.1 since we expect accessing memory via CXL switches has a much higher latency, similar to or even more than NVM access latency.

Figure 14 compares Memtis against TPP [49] since TPP is specifically designed for CXL-based tiered memory systems. As the latency gap between memory tiers decreases compared to the NVM case in Figure 5, the performance gap between TPP and Memtis becomes smaller as well. However, Memtis outperforms TPP in all evaluated benchmarks by up to 32.8%, 102.9%, 39.2%, 27.1%, 16.4%, 12.4%, 32.3%, and 23.2% for Graph500, PageRank, XSBench, Liblinear, Silo, Btree, 603.bwaves, and 654.roms, respectively. Thus, due to its generality, we believe Memtis will be beneficial when CXL memory is finally available.

7 Related Work

Software-based tiered memory system. Prior software-controlled tiered memory systems have explored the design space in various aspects including page migration [34, 71, 83, 84], hotness detection [4, 15, 41], page replacement [28, 48], and kernel object tiering [31, 36] at different layers such as the application [29, 44, 69, 79], library [21, 54, 68], and OS [14, 27, 30, 32, 76, 78]. HotBox [14] suggests not using huge pages in tiered memory systems due to the hotness

[2] As of this writing, there is no publicly available CXL memory hardware.

fragmentation in huge pages. Thermostat [4] precisely detects the access frequency of huge pages using page faults, which incur significant tracking overhead. MaPHeA [54] is a profile-guided optimization technique for heap allocations. It relies on offline profiling, so it is not suited to identify dynamically changing memory access patterns at runtime.

Anti-thrashing mechanisms. HeMem halts both page promotion and demotion when the hot set size exceeds the fast tier size to prevent unnecessary page thrashing among hot pages. TMTS handles all page allocations in the fast tier, and those pages are subsequently protected from demotion for an extended period due to TMTS's demotion policy [22]. This behavior can prevent page thrashing for short-lived allocations. MEMTIS's hotness identification ensures that really hot pages are always placed in the fast tier while minimizing unnecessary page thrashing with warm pages.

Hardware support for tiered memory system. Several studies have focused on hardware mechanisms for heterogeneous memory management [8, 16, 33, 35, 66, 72, 73, 80]. They usually target GPU memory or small-sized HBM. PRISM [9] provides architectural support for variable-sized metadata, such as access bits and dirty bits, by decoupling memory metadata from page size. Their approach enables subpage access tracking for huge pages.

Huge page management. There have been many prior studies on huge page management [13, 23, 26, 38, 43, 47, 50, 53, 57, 58, 67, 87]. They have focused on methods to allocate huge pages, usually under fragmented physical memory, but not for tiered memory systems. None of them, including HawkEye [57], considers skewed accesses when splitting huge pages. Similarly, a kernel patch proposal, called THP Shrinker [86], also splinters huge pages that have many zeroed subpages to reduce huge page-induced memory bloat.

8 Discussion

Comparison to TMTS. TMTS [22] and MEMTIS serve distinct design objectives that could potentially complement each other. TMTS primarily focuses on replacing a portion of its DRAM usage with the slower memory tier to reduce memory cost while minimizing performance impact (<5%), rather than expanding the system memory capacity. TMTS sets its target secondary tier residency ratio (STRR) to 25%, which aligns well with the cold memory ratio observed in WSCs [22, 39, 81]. The demotion and promotion policies of TMTS are designed to maintain secondary tier access ratio (STAR) of applications to remain within the target range (<0.5%), especially in scenarios where the hot working set of applications can fit within the fast tier. TMTS classifies cold pages by identifying idle ages of pages through the kernel daemon, *kstaled* [39], constructing a cold age histogram [39], and adapting the demotion age threshold. Its criteria to select promotion candidates (*i.e.*, hot pages) is rather simple; one access by PEBS or at least two accesses by page table scanning.

Challenges may arise when TMTS operates on tiered memory systems with large capacity tier memory (*e.g.*, 1:2, 1:8, 1:16 configs.). The hot working set of applications could easily exceed the fast tier capacity in such environments, and this is not a scenario that TMTS mainly targets. In this case, its node agent, Borglet, and the cluster-level Borg scheduler [77] may evict applications to maintain the total hot working set size below the fast tier capacity on a machine (and thereby protect the performance SLOs of applications). Extending TMTS to suit such scenarios might require a reconsideration of STAR and STRR, as well as page classification policies, which MEMTIS could potentially complement.

TMTS and MEMTIS also adopt different approaches to huge page management in determining which huge pages to split, when to execute the split, and how many huge pages to split. In TMTS, all demoted huge pages, which are entirely cold and hence do not have any access skewness, undergo splitting upon demotion. Additionally, TMTS utilizes application-level hints to allocate memory objects with similar hotness together within a huge page in TCMalloc, thereby alleviating the access skewness problems within huge pages. In contrast, MEMTIS performs the split only when it is expected to improve the *rHR*, specifically targeting huge pages with high access skewness.

Limitations. Hardware event-based sampling like PEBS has inherent limitations in distinguishing hotness among rarely (or never) accessed pages which are not likely to be sampled, so hotness detection for such pages would be inaccurate. Similar to TMTS, incorporating page table scanning with memory access sampling is a potential solution to alleviate these limitations, although it could introduce runtime overhead without yielding performance benefits.

9 Conclusion

We present MEMTIS, a novel tiered memory system. MEMTIS incorporates dynamic hot set classification based on memory access distribution, enabling it to utilize almost the entirety of fast tier memory for the hottest data. MEMTIS determines page size at runtime to harness the advantages of huge pages while suppressing their downsides by considering their skewness and expected benefits. Our extensive evaluation shows that MEMTIS outperforms state-of-the-art tiering systems in a wide range of workload types and memory configurations, all accomplished with bounded CPU and memory overheads.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Kimberly Keeton, for their invaluable comments. This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (No. 2022R1F1A1070065).

References

[1] Reto Achermann and Ashish Panwar. 2019. Mitosis workload BTree. https://github.com/mitosis-project/mitosis-workload-btree.

[2] Reto Achermann, Ashish Panwar, Abhishek Bhattacharjee, Timothy Roscoe, and Jayneel Gandhi. 2020. Mitosis: Transparently Self-Replicating Page-Tables for Large-Memory Machines. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 283–300. https://doi.org/10.1145/3373376.3378468

[3] Lada A Adamic and Bernardo A Huberman. 2002. Zipf's law and the Internet. *Glottometrics* 3, 1 (2002), 143–150.

[4] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-Transparent Page Management for Two-Tiered Main Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*. 631–644. https://doi.org/10.1145/3037697.3037706

[5] Chloe Alverti, Stratos Psomadakis, Vasileios Karakostas, Jayneel Gandhi, Konstantinos Nikas, Georgios Goumas, and Nectarios Koziris. 2020. Enhancing and Exploiting Contiguity for Fast Memory Virtualization. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*. 515–528. https://doi.org/10.1109/ISCA45697.2020.00050

[6] Anandtech. 2018. Intel Launches Optane DIMMs Up To 512GB: Apache Pass Is Here! https://www.anandtech.com/show/12828/intel-launches-optane-dimms-up-to-512gb-apache-pass-is-here.

[7] Barry C Arnold. 2014. Pareto distribution. *Wiley StatsRef: Statistics Reference Online* (2014), 1–10.

[8] Rachata Ausavarungnirun, Joshua Landgraf, Vance Miller, Saugata Ghose, Jayneel Gandhi, Christopher J. Rossbach, and Onur Mutlu. 2017. Mosaic: A GPU Memory Manager with Application-Transparent Support for Multiple Page Sizes. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*. 136–150.

[9] Rachata Ausavarungnirun, Timothy Merrifield, Jayneel Gandhi, and Christopher J. Rossbach. 2020. PRISM: Architectural Support for Variable-Granularity Memory Metadata. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 441–454. https://doi.org/10.1145/3410463.3414630

[10] Piotr Balcer. 2022. Disaggregated Memory - In Pursuit of Scale and Efficiency. https://pmem.io/blog/2022/01/disaggregated-memory-in-pursuit-of-scale-and-efficiency/.

[11] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. 2013. Efficient Virtual Memory for Big Memory Servers. In *Proceedings of the 40th Annual International Symposium on Computer Architecture*. Association for Computing Machinery, New York, NY, USA, 237–248. https://doi.org/10.1145/2485922.2485943

[12] Scott Beamer, Krste Asanovic, and David A. Patterson. 2015. The GAP Benchmark Suite. *CoRR* abs/1508.03619 (2015). arXiv:1508.03619 http://arxiv.org/abs/1508.03619

[13] Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martín Farach-Colton, Rob Johnson, Sudarsun Kannan, William Kuszmaul, Nirjhar Mukherjee, Don Porter, Guido Tagliavini, Janet Vorobyeva, and Evan West. 2021. Paging and the Address-Translation Problem. In *Proceedings of the 33rd ACM Symposium on Parallelism in Algorithms and Architectures*. 105–117. https://doi.org/10.1145/3409964.3461814

[14] Shai Bergman, Priyank Faldu, Boris Grot, Lluís Vilanova, and Mark Silberstein. 2022. Reconsidering OS Memory Optimizations in the Presence of Disaggregated Memory. In *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management*. 1–14. https://doi.org/10.1145/3520263.3534650

[15] Jinyoung Choi, Sergey Blagodurov, and Hung-Wei Tseng. 2021. Dancing in the Dark: Profiling for Tiered Memory. In *Proceedings of the 2021 IEEE International Parallel and Distributed Processing Symposium*. 13–22. https://doi.org/10.1109/IPDPS49936.2021.00011

[16] Chia Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. 2014. CAMEO: A Two-Level Memory Organization with Capacity of Main Memory and Flexibility of Hardware-Managed Cache. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 1–12. https://doi.org/10.1109/MICRO.2014.63

[17] CXL Consortium. 2022. Compute Express Link Specification 3.0. https://www.computeexpresslink.org/download-the-specification.

[18] 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*. 143–154.

[19] Jonathan Corbet. 2022. CXL 1: Management and tiering. https://lwn.net/Articles/894598/.

[20] Standard Performance Evaluation Corporation. 2022. SPEC CPU 2017. https://www.spec.org/cpu2017/.

[21] 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*. 1–16. https://doi.org/10.1145/2901318.2901344

[22] 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 *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 727–741. https://doi.org/10.1145/3582016.3582031

[23] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quema. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Annual Technical Conference*. 231–242. https://www.usenix.org/conference/atc14/technical-sessions/presentation/gaud

[24] Alessandro Goncalves. 2022. Make Sense of Memory Tiering. https://www.snia.org/educational-library/make-sense-memory-tiering-2022.

[25] Krishnan Gosakan, Jaehyun Han, William Kuszmaul, Ibrahim N. Mubarek, Nirjhar Mukherjee, Karthik Sriram, Guido Tagliavini, Evan West, Michael A. Bender, Abhishek Bhattacharjee, Alex Conway, Martin Farach-Colton, Jayneel Gandhi, Rob Johnson, Sudarsun Kannan, and Donald E. Porter. 2023. Mosaic Pages: Big TLB Reach with Small Pages. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 433–448. https://doi.org/10.1145/3582016.3582021

[26] Fan Guo, Yongkun Li, Yinlong Xu, Song Jiang, and John C. S. Lui. 2017. SmartMD: A High Performance Deduplication Engine with Mixed Pages. In *Proceedings of the 2017 USENIX Annual Technical Conference*. 733–744. https://www.usenix.org/conference/atc17/technical-sessions/presentation/guo-fan

[27] Vishal Gupta, Min Lee, and Karsten Schwan. 2015. HeteroVisor: Exploiting Resource Heterogeneity to Enhance the Elasticity of Cloud Platforms. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 79–92. https://doi.org/10.1145/2731186.2731191

[28] Taekyung Heo, Yang Wang, Wei Cui, Jaehyuk Huh, and Lintao Zhang. 2022. Adaptive Page Migration Policy With Huge Pages in Tiered Memory Systems. *IEEE Trans. Comput.* 71, 1 (2022), 53–68. https://doi.org/10.1109/TC.2020.3036686

[29] Mark Hildebrand, Jawad Khan, Sanjeev Trika, Jason Lowe-Power, and Venkatesh Akella. 2020. AutoTM: Automatic Tensor Movement in Heterogeneous Memory Systems Using Integer Linear Programming. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 875–890. https://doi.org/10.1145/3373376.3378465

[30] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. 2017. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 521–534. https://doi.org/10.1145/3079856.3080245

[31] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. 2021. KLOCs: Kernel-Level Object Contexts for Heterogeneous Memory Systems. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 65–78. https://doi.org/10.1145/3445814.3446745

[32] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *Proceedings of the 2021 USENIX Annual Technical Conference*. 715–728. https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon

[33] Apostolos Kokolis, Dimitrios Skarlatos, and Josep Torrellas. 2019. PageSeer: Using Page Walks to Trigger Page Swaps in Hybrid Memory Systems. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. 596–608. https://doi.org/10.1109/HPCA.2019.00012

[34] Vamsee Reddy Kommareddy, Simon David Hammond, Clayton Hughes, Ahmad Samih, and Amro Awad. 2019. Page Migration Support for Disaggregated Non-Volatile Memories. In *Proceedings of the International Symposium on Memory Systems*. 417–427. https://doi.org/10.1145/3357526.3357543

[35] Jagadish B. Kotra, Haibo Zhang, Alaa R. Alameldeen, Chris Wilkerson, and Mahmut T. Kandemir. 2018. CHAMELEON: A Dynamically Reconfigurable Heterogeneous Memory System. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*. 533–545. https://doi.org/10.1109/MICRO.2018.00050

[36] Sandeep Kumar, Aravinda Prasad, Smruti R. Sarangi, and Sreenivas Subramoney. 2021. Radiant: Efficient Page Table Management for Tiered Memory Systems. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management*. 66–79. https://doi.org/10.1145/3459898.3463907

[37] Haewoon Kwak, Changhyun Lee, Hosung Park, and Sue Moon. 2010. What is Twitter, a Social Network or a News Media?. In *Proceedings of the 19th International Conference on World Wide Web*. 591–600. https://doi.org/10.1145/1772690.1772751

[38] Youngjin Kwon, Hangchen Yu, Simon Peter, Christopher J. Rossbach, and Emmett Witchel. 2016. Coordinated and Efficient Huge Page Management with Ingens. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*. 705–721. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon

[39] 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*. New York, NY, USA, 317–330. https://doi.org/10.1145/3297858.3304053

[40] Seok-Hee Lee. 2016. Technology scaling challenges and opportunities of memory devices. In *Proceedings of the 2016 IEEE International Electron Devices Meeting*. 1.1.1–1.1.8. https://doi.org/10.1109/IEDM.2016.7838026

[41] Taehyung Lee and Young Ik Eom. 2022. Optimizing the Page Hotness Measurement with Re-Fault Latency for Tiered Memory Systems. In *2022 IEEE International Conference on Big Data and Smart Computing (BigComp)*. 275–279. https://doi.org/10.1109/BigComp54360.2022.00059

[42] 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 *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 574–587. https://doi.org/10.1145/3575693.3578835

[43] Xinyu Li, Lei Liu, Shengjie Yang, Lu Peng, and Jiefan Qiu. 2019. Thinking about A New Mechanism for Huge Page Management. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*. 40–46. https://doi.org/10.1145/3343737.3343745

[44] Zhe Li and Mingyu Wu. 2022. Transparent and Lightweight Object Placement for Managed Workloads atop Hybrid Memories. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 72–80. https://doi.org/10.1145/3516807.3516822

[45] Chih-Jen Lin. 2021. LIBLINEAR – A Library for Large Linear Classification v2.43. https://www.csie.ntu.edu.tw/~cjlin/liblinear/.

[46] Chris A. Mack. 2011. Fifty Years of Moore's Law. *IEEE Transactions on Semiconductor Manufacturing* 24, 2 (2011), 202–207. https://doi.org/10.1109/TSM.2010.2096437

[47] Mark Mansi, Bijan Tabatabai, and Michael M. Swift. 2022. CBMM: Financial Advice for Kernel Memory Managers. In *Proceedings of the 2022 USENIX Annual Technical Conference*. 593–608. https://www.usenix.org/conference/atc22/presentation/mansi

[48] Adnan Maruf, Ashikee Ghosh, Janki Bhimani, Daniel Campello, Andy Rudoff, and Raju Rangaswami. 2022. MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems. In *Proceedings of the 2022 IEEE International Symposium on High-Performance Computer Architecture*. 925–937. https://doi.org/10.1109/HPCA53966.2022.00072

[49] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit Kanaujia, and Prakash Chauhan. 2023. TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 742–755. https://doi.org/10.1145/3582016.3582063

[50] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. 2019. MEGA: Overcoming Traditional Problems with OS Huge Page Management. In *Proceedings of the 12th ACM International Conference on Systems and Storage*. 121–131. https://doi.org/10.1145/3319647.3325839

[51] Timothy Prickett Morgan. 2022. The future of system memory is mostly CXL. https://www.nextplatform.com/2022/07/05/the-future-of-system-memory-is-mostly-cxl/.

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

[53] Juan Navarro, Sitaram Iyer, and Alan Cox. 2002. Practical, Transparent Operating System Support for Superpages. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*. https://www.usenix.org/conference/osdi-02/practical-transparent-operating-system-support-superpages

[54] Deok-Jae Oh, Yaebin Moon, Eojin Lee, Tae Jun Ham, Yongjun Park, Jae W. Lee, and Jung Ho Ahn. 2021. MaPHeA: A Lightweight Memory Hierarchy-Aware Profile-Guided Heap Allocation Framework. In *Proceedings of the 22nd ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*. 24–36. https://doi.org/10.1145/3461648.3463844

[55] John Ousterhout, Parag Agrawal, David Erickson, Christos Kozyrakis, Jacob Leverich, David Mazières, Subhasish Mitra, Aravind Narayanan, Guru Parulkar, Mendel Rosenblum, Stephen M. Rumble, Eric Stratmann, and Ryan Stutsman. 2010. The Case for RAMClouds: Scalable High-Performance Storage Entirely in DRAM. *SIGOPS Oper. Syst. Rev.* 43, 4 (jan 2010), 92–105. https://doi.org/10.1145/1713254.1713276

[56] Ashish Panwar, Reto Achermann, Arkaprava Basu, Abhishek Bhattacharjee, K. Gopinath, and Jayneel Gandhi. 2021. Fast Local Page-Tables for Virtualized NUMA Servers with VMitosis. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 194–210. https://doi.org/10.1145/3445814.3446709

[57] Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. 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*. 347–360. https://doi.org/10.1145/3297858.3304064

[58] Ashish Panwar, Aravinda Prasad, and K. Gopinath. 2018. Making Huge Pages Actually Useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. 679–692. https://doi.org/10.1145/3173162.3173203

[59] SeongJae Park. 2021. Introduce Data Access MONitor (DAMON). https://lwn.net/Articles/849708/.

[60] SeongJae Park, Madhuparna Bhowmik, and Alexandru Uta. 2022. DAOS: Data Access-Aware Operating System. In *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*. Association for Computing Machinery, 4–15. https://doi.org/10.1145/3502181.3531466

[61] Industry Perspectives. 2015. Don't Forget About Memory: DRAM's Surprising Role in the High Cost of Data Centers. https://www.datacenterknowledge.com/archives/2015/11/12/dont-forget-memory-drams-surprising-role-high-cost-data-centers.

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

[63] Phoronix. 2022. MGLRU Looks Like One Of The Best Linux Kernel Innovations Of The Year. https://www.phoronix.com/news/MGLRU-LPC-2022.

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

[65] pmem.io. 2019. Using the memmap Kernel Option. https://docs.pmem.io/persistent-memory/getting-started-guide/creating-development-environments/linux-environments/linux-memmap.

[66] Andreas Prodromou, Mitesh Meswani, Nuwan Jayasena, Gabriel Loh, and Dean M. Tullsen. 2017. MemPod: A Clustered Architecture for Efficient and Scalable Migration in Flat Address Space Multi-level Memories. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. 433–444. https://doi.org/10.1109/HPCA.2017.39

[67] Venkat Sri Sai Ram, Ashish Panwar, and Arkaprava Basu. 2021. Trident: Harnessing Architectural Resources for All Page Sizes in X86 Processors. In *Proceedings of the 54th Annual IEEE/ACM International Symposium on Microarchitecture*. 1106–1120. https://doi.org/10.1145/3466752.3480062

[68] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 392–407. https://doi.org/10.1145/3477132.3483550

[69] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. 2021. Sentinel: Efficient Tensor Migration and Allocation on Heterogeneous Memory Systems for Deep Learning. In *Proceedings of the 2021 IEEE International Symposium on High-Performance Computer Architecture*. 598–611. https://doi.org/10.1109/HPCA51647.2021.00057

[70] Rafał Rudnicki. 2022. Memory Tiering (Part 1). https://pmem.io/blog/2022/06/memory-tiering-part-1.

[71] Jee Ho Ryoo, Lizy K. John, and Arkaprava Basu. 2018. A Case for Granularity Aware Page Migration. In *Proceedings of the 2018 International Conference on Supercomputing*. 352–362. https://doi.org/10.1145/3205289.3208064

[72] Jee Ho Ryoo, Mitesh R. Meswani, Andreas Prodromou, and Lizy K. John. 2017. SILC-FM: Subblocked InterLeaved Cache-Like Flat Memory Organization. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture*. 349–360. https://doi.org/10.1109/HPCA.2017.20

[73] Jaewoong Sim, Alaa R. Alameldeen, Zeshan Chishti, Chris Wilkerson, and Hyesoon Kim. 2014. Transparent Hardware Management of Stacked DRAM as Part of Memory. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 13–24. https://doi.org/10.1109/MICRO.2014.56

[74] John R Tramm, Andrew R Siegel, Tanzima Islam, and Martin Schulz. 2014. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*. https://www.mcs.anl.gov/papers/P5064-0114.pdf

[75] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-Memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. 18–32. https://doi.org/10.1145/2517349.2522713

[76] Rik van Riel and Vinod Chegu. 2014. Automatic NUMA balancing. *Red Hat Summit* (2014).

[77] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. 2015. Large-Scale Cluster Management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*. Association for Computing Machinery, New York, NY, USA, Article 18, 17 pages. https://doi.org/10.1145/2741948.2741964

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

[79] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. 2019. Panthera: Holistic Memory Management for Big Data Processing over Hybrid Memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 347–362. https://doi.org/10.1145/3314221.3314650

[80] Xiaoyuan Wang, Haikun Liu, Xiaofei Liao, Ji Chen, Hai Jin, Yu Zhang, Long Zheng, Bingsheng He, and Song Jiang. 2019. Supporting Superpages and Lightweight Page Migration in Hybrid Memory Systems. *ACM Trans. Archit. Code Optim.* 16, 2, Article 11 (2019), 26 pages. https://doi.org/10.1145/3310133

[81] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent Memory Offloading in Datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA, 609–621. https://doi.org/10.1145/3503222.3507731

[82] Network World. 2018. Facebook and Amazon are causing a memory shortage. https://www.networkworld.com/article/3247775/facebook-and-amazon-are-causing-a-memory-shortage.html.

[83] Zi Yan. 2017. mm: page migration enhancement for thp. https://lwn.net/Articles/723764/.

[84] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble Page Management for Tiered Memory Systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 331–345. https://doi.org/10.1145/3297858.3304024

[85] Yu Zhao. 2022. Multigenerational LRU Framework. https://lwn.net/Articles/880393/.

[86] Alexander Zhu. 2022. [RFC 0/3] THP Shrinker. https://lwn.net/ml/linux-kernel/cover.1661461643.git.alexlzhu@fb.com/.

[87] Weixi Zhu, Alan L. Cox, and Scott Rixner. 2020. A Comprehensive Analysis of Superpage Management Mechanisms and Policies. In *Proceedings of the 2020 USENIX Annual Technical Conference*. 829–842. https://www.usenix.org/conference/atc20/presentation/zhu-weixi