TieredMMS: A Modular Tiered Memory Management System

Song Wei, Minmin Ren, Chao Jia, Gang Wang, Longguang Yue, Xin Zhong, Haifeng Chen
China Unicom Cloud Data Co., Ltd, Beijing, China
China Unicom Digital Technology Co., Ltd., Beijing, China
China United Network Communication Group Co., Ltd, Beijing, China
{weis28, renmm6, jiac13, wangg163, yuelg, zhongx59, chenhaifeng}@chinaunicom.cn

***Abstract*—With the rise of data-intensive applications, the rapidly growing memory demand poses a challenge in data centers. Tiered memory systems with emerging memory technologies (e.g., NVM, CXL) is a promising solution. However, previous tiering systems have some limitations. Some incur scan overhead from their page tracking module. Others have kernel-intrusive page migration modules that are difficult to evolve with kernel versions. In this work, we present TieredMMS, a modular and hardware-based memory tiering system. TieredMMS comprises a userspace page access tracker and a page migration kernel module. The userspace component communicates with the kernel through shared ring buffers and *ioctl* system calls. Leveraging eBPF, TieredMMS can be aware of the page life cycle. The page access tracker measures page access count within each life cycle via hardware sampling and classifies pages as hot, warm, or cold. The page migration module (1) orders OS-maintained LRU lists by classification results; and (2) performs periodic or direct page migration based on available DRAM memory. Our evaluation shows that TieredMMS performance is up to 4.97× that of AutoNUMA. Moreover, TieredMMS outperforms state-of-the-art tiered memory systems by up to 26.3% in 3/6 test cases.**

***Index Terms*—Data-intensive, Tiered Memory Systems, Page Hotness, Address Sampling, Page Migration**

I. Introduction

With the rapid development of data-intensive applications, such as artificial intelligence, graph analytics, and big data, computing systems are facing challenges in memory capacity, memory bandwidth, and memory utilization.

Fortunately, the emergence of storage class memory (SCM) and high-speed interconnect standards provides opportunities to address these memory issues. One typical commodity storage medium is Intel's Optane DIMM, which is byte-addressable and non-volatile [1]. For performance, it sits between flash and DRAM in the storage layers. One obvious drawback of DDR interface media is that it occupies limited memory slots. Compute eXpress Link (CXL) [2], an open standard released by a number of vendors in 2019, avoids this limitation. It is based on the PCI Express (PCIe) 5.0 physical layer infrastructure and provides new features maintaining a unified, coherent memory space between the host CPU and memory on the attached CXL device. Thus, the host CPU accesses the device-attached memory with load/store semantics. According to existing theoretical analysis, its latency is comparable to a remote NUMA node [3]. This assumption applies to DRAM memory. However, CXL-attached memory is not limited to DRAM but is diverse. DRAM, flash, or even a mixture of both could be connected to a server through the CXL interface and treated as memory.

Combining multiple memory types delivers better performance, higher resource utilization, and cost trade-offs. These benefits depend on an efficient tiered memory management system. Linux memory management assumes all memory has uniform capabilities based on DRAM. Previous researchers have made numerous attempts and explorations in this area. However, previous works on tiered memory systems have some limitations. The key to an efficient data tiering policy is identifying frequently and recently used data. Some studies track page access by scanning page tables [4–6], but scalability is limited as the working set size grows. Some works use hardware sampling for page access tracking [7, 8], but their page placement module is intrusive to the Linux kernel. MaPHeA [7] is a profile-guided optimization technique for heap allocations. It relies on offline profiling and is thus unsuited to identifying dynamically changing memory access patterns at runtime. TMO [9] only knows how much data can be offloaded but cannot identify the best candidates. Moreover, some approaches are tightly integrated with the kernel [4, 10, 11]. Therefore, software deployment requires recompiling the kernel. X-Mem [12] defines a set of customized APIs and only applications adapted to these APIs can benefit. Additionally, some research is confined to specific application scenarios [13–18]. None of these works meet our design objective: an efficient and modular tiered memory management system.

In this paper, we propose TieredMMS, the first hardware-based, efficient, and modular tiered memory management system. TieredMMS tracks page allocation and release in userspace via eBPF, initializing it on page allocation and clearing it on page release. The access count is accumulated during the page's lifetime using hardware-based sampling. The hot pages are determined by accommodating the most frequently accessed pages into DRAM until it is nearly full. The rest of the pages are treated as warm and cold pages. Hot/cold pages are put into the hot/cold ring buffer shared by the userspace and kernel space. Warm pages serve as a buffer between hot and cold pages to avoid frequent page migration back and forth. TieredMMS augments the OS-maintained LRU

lists by moving hot pages to the active list and cold pages to the inactive list. To execute page migration operation, TieredMMS starts a demotion thread on each DRAM node and a promotion thread on each non-DRAM node. On the DRAM node, TieredMMS defines 2 watermarks, *high_wm* and *low_wm*. These two watermarks indicate the free memory size on the current node. When the free memory is below the *high_wm*, TieredMMS isolates pages from the inactive list and tries to move them to the non-DRAM node until the free memory reaches *high_wm* again. If the free memory continues to decrease until it is below *low_wm*, the migration thread is woken up immediately and tries to perform the page migration. On the non-DRAM node, as long as the anonymous active list is not empty, TieredMMS tries to migrate pages to the DRAM node.

We choose two representative memory-intensive workloads, Liblinear [19] and Graph500 [20]. TieredMMS moves frequently accessed pages to the DRAM node and improves the default Linux's AutoNUMA performance by up to 3.97×. We compare TieredMMS with Memtis, a state-of-the-art tiered memory management system. Our evaluation shows that TieredMMS outperforms Memtis by 8%-26.3% in 3/6 test cases.

We summarize the contributions of this paper as follows: (1) We present TieredMMS, which is implemented by a user program and a kernel module. Therefore, it is easy to maintain and deploy. We plan to open source TieredMMS. (2) We propose a page access counting method based on the page life cycle and a page migration mechanism based on watermarks. (3) We combine the advantages of hardware-based sampling and OS-maintained LRU. LRU is built based on the recency principle, and PEBS reflects the page access frequency.

II. Background and Motivation

A. Tiered Memory System

With the development of new storage media and memory semantic interconnection bus. There are usually multiple types of memory in the computing system. The typical storage media is persistent memory and CXL-attached memory.

DRAM+PMem memory systems. For persistent memory, Intel Optane DC PMem is the first and only commodity product. It increases the memory capacity at the cost of ∼5× lower bandwidth and ∼2-3× latency compared with DRAM [21, 22]. Besides the read and write performance is not asymmetric [23]. Although Intel has dropped most of its storage business, it does not affect our research. The future Intel CPU will integrate HBM which is a new memory tier.

CXL-based memory systems. CXL is a promising industry interconnection protocol [2]. With the cache-coherent and memory semantic, CXL enables memory expansion, memory pooling, and even memory sharing. Although there are many issues before wide deployment, CXL-based tiered memory has great potential in dealing with memory stranding [3], memory capacity, and memory bandwidth.

Due to the performance gap between different types of memory media, the tiered memory management aims at scheduling data among different memory layers to achieve better performance. A classic approach achieves this goal by tracking memory access patterns, classifying hot/cold pages, and automatically migrating pages between fast memory and slow memory. Therefore, an efficient data placement policy is critically important.

B. Processor Event-Based Sampling

PEBS is a hardware mechanism of PMU on Intel processors. It was supported on the Nehalem processor and extended in the subsequent processor. PEBS is capable of recording EFLAG register, instruction pointer (IP), general registers, and so on. From Haswell, PEBS can also record the data linear address for both memory load and store. With data linear address, memory access profiling is available.

The PEBS relies on specified hardware counters and the Debug Store (DS) mechanism [24]. DS maintains a software-defined area of memory that is used to save PEBS records. The base address of this memory area is stored in IA32_DS_AREA MSR. There are 3 parts in DS save area: buffer management area, branch trace store (BTS) buffer, and PEBS buffer. The buffer management area contains the metadata of PEBS buffer and BTS buffer. One important field in the buffer management area is the PEBS index, which remembers where the next PEBS record will be written.

The PEBS hardware counters are driven by hardware events that we are interested in, such as retired load/store micro-operations (μOPs). On overflow of a PEBS-enabled counter, the PEBS facility is armed. At the occurrence of the next PEBS event, the PEBS hardware will take an assist, and a PEBS record is written in the OS-defined PEBS buffer.

When the PEBS buffer is nearly full, a Performance Monitoring Interrupt (PMI) generates, and the control flow transfers to the kernel PMI handler. PMI handler reads the PEBS buffer and stores the required data into a ring buffer. The ring buffer is defined by users and shared between the user and kernel space. *data_head* points to the head of the ring buffer. *data_tail* indicates the data to be read by the user. The user process setup PEBS by the *perf_event_open* system call. When reading data from the ring buffer, the user needs to maintain a copy of *data_head* and *data_tail*. **data_head** is written by the kernel and read by the user. When the user copies it, an acquiring load is required to ensure the user sees *data_head* first instead of the ring buffer data. **data_tail** is written by the user and read by the kernel. When the user updates it, a releasing store is required to ensure the user has seen the data from the ring buffer.

C. eBPF

BPF (Berkeley Packet Filter) is an interface that allows users to offload a simple function to be executed by the kernel. Linux's framework for BPF is called eBPF (extended BPF) [25, 26]. Functions are verified by the kernel at install-time to ensure they are safe; for example, they are checked to make sure they do not contain too many instructions, unbounded loops, or accesses to out-of-bounds

memory addresses [27]. After verification, the eBPF functions can be called normally. The desired data can be transferred to userspace through the map, ring buffer, or perf buffer.

III. Design of TieredMMS

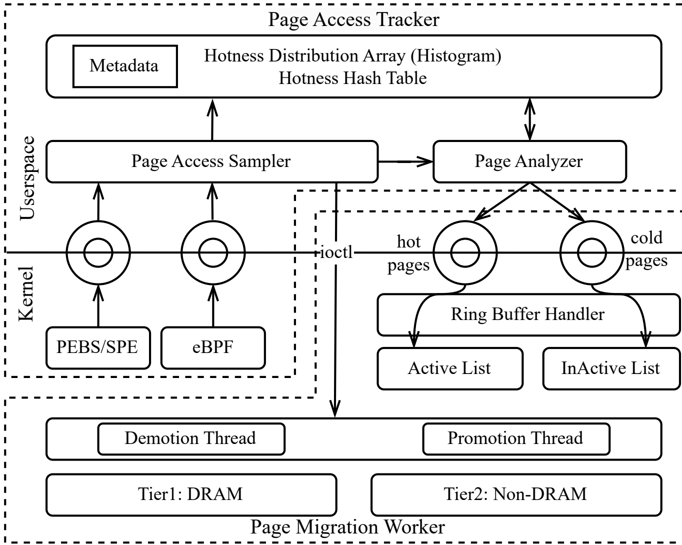


Fig. 1: Architecture of TieredMMS.

The overview architecture of TieredMMS is shown in Figure 1. TieredMMS consists of two parts: the page access tracker and the page migration module. The page access tracker runs in userspace. It is responsible for (1) handling PEBS samples and updating the page access metadata; (2) hooking the memory allocation path via eBPF and notifying the page migration module if the DRAM node is under extreme pressure; (3) performing page classification and filling the page into the hot/cold ring buffer. The page migration module runs in the kernel and its ring buffer handler moves pages between the LRU lists according to the hot/cold ring buffer and performs periodic page migration.

A. Page Access Tracker

As shown in Figure 1, the page access tracker includes the page access sampler and the page analyzer. The two components communicate through the page access metadata. The page access sampler parses the PEBS sample records and stores them in the page access metadata. The page analyzer conducts (1) threshold calculation; (2) metadata cooling; and (3) page classification.

1) Page Access Sampler: Production workloads often run with multiple threads and may spawn child threads during the execution. All threads read and write memory and are likely to migrate across different cores. Therefore, the page access sampler should (1) count memory access for the specified task and its child tasks on all CPU cores; and (2) record both reads and writes. Figure 2 shows the page access tracker design. It samples memory access using PEBS and the *perf_event_open* system call as described in Section II. To achieve the first goal, we open one file descriptor per CPU core and set the *inherit*

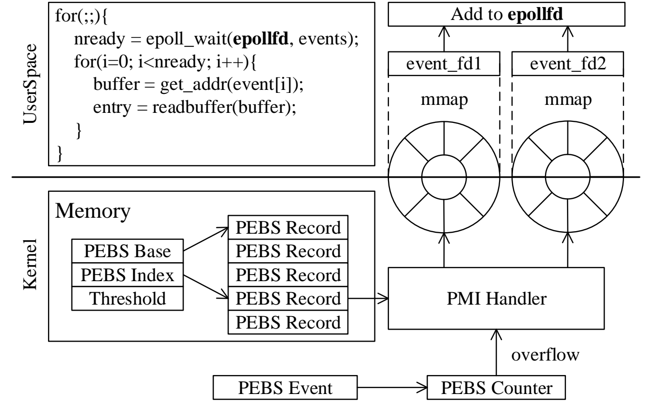


Fig. 2: Page access sampler design based on PEBS.

parameter. Thus, the counter could count both the specified task and its children regardless of which core they run on. Since each file descriptor binds to one event, we open multiple file descriptors on each CPU core to achieve the second goal. We also create an *epoll* instance and add all file descriptors to *epoll* interest list. We poll the *epoll* file descriptor in an infinite loop until the specified task terminates. Upon wakeup, we can read records from the ring buffer.

The three hardware events that we are interested in are *mem_load_l3_miss_retired.local_dram* (0x01D3), *mem_load_retired.local_pmm*(0x80D1), and *mem_inst_retired.all_stores* (0x82D0). When considering memory load, *mem_inst_retired.all_loads* (0x81D0) is not suitable because it does not provide information regarding real memory access to the main memory. Hardware sampling mechanisms have the capability to sample both virtual and physical addresses. We record both types of them. First, the physical address will change after page migration. This brings troubles for the maintenance of page access count. Therefore, we record virtual addresses, which remain unchanged during page migration. Second, to avoid translation from virtual addresses into physical addresses, we also record physical addresses. However, the hardware sampling mechanism is not aware of the life cycle of pages. To avoid new pages not being affected by previous counts, we hook two kernel functions (*do_anonymous_page* and *free_pages*) with eBPF to track page allocation and deallocation. We give each new page an initial value and reset the value when the page is free.

It is generally accepted that there is a trade-off between overhead and accuracy. To obtain an optimal balance point we made the following efforts. (1) Reduce the wake-up times. The *mmap* ring buffer space is limited. To capture samples timely, we set the threshold for overflow notification as 1. This means the memory access tracker will be woken up as long as there is a sample. When receiving the notification, the page access tracker reads all the data until the ring buffer is empty; (2) Since the kernel memory management and data

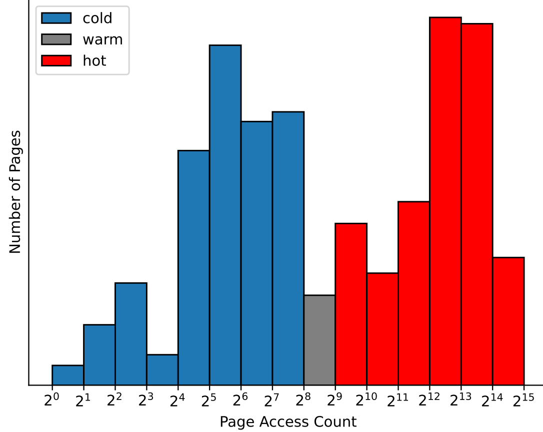


Fig. 3: Hotness histogram.

migration granularity is page, we do not care about every single load/store operation. Instead, we merge the adjacent accesses that fall on the same page. This can be achieved by masking the 12 least significant bits of each sampled address. (3) Sample rate. Sampling every memory operation is not practical. If we set the sampling frequency to a higher value, the kernel will automatically reduce the sampling frequency and perf will be throttled. Since there are more store instruction events than Last Level Cache (LLC) load miss events, we set different intervals for the events. In our experiments, the sample intervals for LLC load miss and all store events are 199 and 100,007 respectively. In our experiments, we find that the upper limit of CPU usage is 4.25% of a single core.

2) Page Analyzer: The first thing we need to clarify is the page access metadata, including the hotness distribution array and hotness hash table. The hotness hash table is used to store the hotness value of each page. In this work, we choose the page base address as the key. The page's hotness value, represented by the page access count in a period, is stored in the hash bucket. The hotness distribution array represents the global page access count distribution. The pseudo hotness distribution is shown in Figure 3. The *x-axis* is the page access count and is divided into 16 intervals that are represented with 16 bars. A page with hotness in range $[2^n, 2^{n+1})$ falls into the *n-th* interval. The bar height is the number of distinct pages. The hotness distribution figure is represented by an array with 16 elements.

Hotness distribution provides an important basis for threshold calculation. As shown in Figure 3, pages are divided into three groups: hot, warm, and cold. The warm group merits particular attention. Some cold pages exhibit hotness close to the hot threshold, while some hot pages have hotness near the cold threshold. Pages in these categories are prone to frequent migration between hot and cold sets. The warm interval serves as a buffer to mitigate this. The threshold calculation algorithm appears in Algotithm 1 and α is 75%. The key idea of determining the three thresholds is to maximize pages fitting

Algorithm 1: Threshold Calculation

Input: The number of free pages in DRAM node (F_s), The number of pages that fall in histogram bar b (S_b), maximum of bar index of the histogram max

```
1  Initialization: sum=0;
2  for b = max; b >= 0; b − − do
3      if sum + S_b > F_s then
4          break;
5      end
6      sum+ = S_b;
7  end
8  T_hot = b + 1;
9  if sum < F_S × α then
10     T_warm = T_hot − 1;
11 else
12     T_warm = T_hot;
13 end
14 T_cold = T_warm − 1;
```

within the DRAM node. We accumulate bars from right to left until exceeding the DRAM node capacity. The current bar index (T_{hot}) represents the hot threshold. One bar is reserved as a buffer between hot and cold pages, termed warm pages. Thus, the warm threshold (T_{warm}) and cold threshold (T_{cold}) are $T_{hot} - 1$ and $T_{warm} - 1$ respectively.

The complete workflow of the page access tracker is shown in Algorithm 2. First, we allocate memory for the hash table and the hotness distribution array. Second, the main thread sleeps until data is available in the monitored ring buffers. Third, we read one entry from the ring buffer and update the hotness hash table. We update the hotness distribution array when the sampled address moves from one bar to another. Fourth, perform page classification. Pages on the right of T_{warm} go to the hot ring buffer. Pages on the left of T_{warm} go to the cold ring buffer. Fifth, we apply the recency effect by halving the page access count in the hash table and refilling the hotness distribution array. If a page becomes cold after cooling, it goes to the cold ring buffer.

B. Page Migration

Page migration is responsible for promoting hot pages to the DRAM node and demoting cold pages to the non-DRAM node. It is implemented as a kernel module. For simplicity, we reuse the OS-maintained LRU lists. Since Linux kernel v5.0, non-DRAM memory can be exposed as a CPU-less NUMA node. Each NUMA node has an LRU list vector containing active and inactive lists. However, the main purpose of the OS-maintained LRU lists is to provide a relative classification of pages for reclamation. The LRU lists are built based on the recency principle but the page access frequency is omitted. We consider the frequency factor by actively moving pages between active and inactive lists based on their hotness, as measured by the page access tracker. The workflow is shown

Algorithm 2: Workflow of Page Access Tracker

```
1  Initialization: Hash table and hotness distribution array;
2  while !ring_buffer_empty() do
3      Get one sample entry;
4      if valid address then
5          nr_sampled++;
6          Update the hotness distribution array;
7          Update the hotness hash table;
8          Perform page classification;
9          if nr_sampled % threshold_period == 0 then
10             Calculate the threshold;
11         else if nr_sampled % cool_period == 0 then
12             Halving the hash table;
13             Refilling the histogram;
14         end
15     end
16 end
```

Algorithm 3: Ring Buffer Handler

```
1  while !shouldExit do
2      sleep(100us);
3      nr_entries = data_head - data_tail;
4      while nr_entries do
5          Get the page address from ring buffer;
6          if hot page then
7              Move it to the head of the active list;
8          else
9              Move it to the head of the inactive list;
10         end
11         nr_entries−−;
12     end
13 end
```

in Algorithm 3. If the current page is hot, the ring buffer handler moves it to the head of the active list. Otherwise, the page is moved to the inactive list. In this way, the LRU is ordered by page hotness.

To perform the actual page migration operation, we learn from *swap*. First, we start a thread on each memory node. On the DRAM node, there is a demotion thread. On the non-DRAM node, there is a promotion thread. Second, for the DRAM node, we define two memory watermarks, *high_wm* and *low_wm*, which are shown in Figure 4. We set *high_wm* and *low_wm* to 3% and 2% of the DRAM node capacity, respectively. The page migration thread periodically (500ms) checks the watermarks. When the free memory size is lower than the *high_wm* watermark, the demotion thread will migrate pages to the non-DRAM node, which is called periodic demotion. Third, we hook the page allocation path and check free memory size. If the free memory size is lower than the *low_wm* watermark, the page access tracker will wake up the demotion thread immediately by the system call, which is called direct demotion. The promotion condition is that

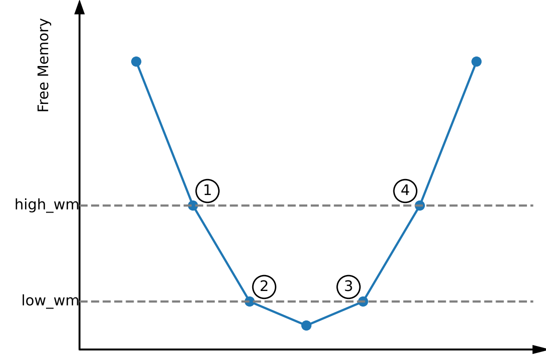


Fig. 4: Demotion watermarks. ① Periodic demotion starts; ② Direct demotion starts; ③ Direct demotion stops; ④ Periodic demotion stops.

the anonymous active list is not empty. Data Warehouse uses anonymous pages for computation. The file pages are used for writing intermediate computation data to the storage device. Previous research has demonstrated that almost all hot pages are anonymous whereas almost all file pages remain cold [10]. Therefore, the demotion fallback path is inactive file LRU, active file LRU, and inactive anonymous LRU. Pages on active anonymous LRU are not candidates for demotion. Each time the page access tracker cools down its metadata, it will notify the demotion thread. The demotion thread will move 50% of the pages at the tail of the active anonymous LRU to the inactive LRU.

IV. IMPLEMENTATION

TieredMMS is implemented by a user program and a kernel module. The total lines of code (LOC) is 5696, including 4077 lines of user code and 1619 lines of kernel module code.

The metadata size of TieredMMS depends on the memory size we are interested in. Each hash bucket has a 4-byte counter and a 8-byte physical address and the total size is 12 bytes. Take 1TiB of memory as an example, 256M (1TiB/4KiB) buckets are required. The metadata requires a total of 3072MiB. In the worst case, the memory overhead is 0.293%. Considering the memory footprint, static memory allocation is not a good choice. Instead, we use the *map* container in the C++ Standard Template Library (STL) to record the page access count. It is implemented as a Red-Black tree and memory is dynamically allocated/deallocated by the C++ runtime.

The page migration worker is implemented as a kernel module to minimize kernel intrusions. There are two main challenges in the implementation of the migration module. First, the Linux LRU lists and many kernel APIs are not exposed to the kernel module by default. We obtain the global

TABLE I: Benchmarks

Benchmark	RSS	Description
Graph500	67GB	Generate a graph and perform BFS search for 64 keys
Liblinear	45GB	We run the multi-core Liblinear benchmark with KDD12 [28] dataset

variables and critical kernel APIs by *kprobe* mechanism. Second, the kernel module is not in the kernel's memory allocation path, so it cannot be aware of the upcoming memory pressure in time. We hooked the kernel's *alloc_pages_vma* function by eBPF and obtained the memory size to be allocated from its entry parameter. Third, the communication between the page access tracker and the page migration worker. The userspace page access tracker needs to inform the page migration module of sampled hot pages. We use ring buffers to send the hot/cold pages to the kernel module. Besides, when the free memory falls below the *low_wm*, the page access tracker immediately wakes up the page migration thread by a system call.

V. Evaluation

Our evaluation answers the following questions:

- How effective is TieredMMS in identifying hot pages of memory-intensive applications?
- How does TieredMMS perform against the state-of-the-art memory tiering system?
- What is the performance gap between TieredMMS and the DRAM-only system?

A. Experimental Setup

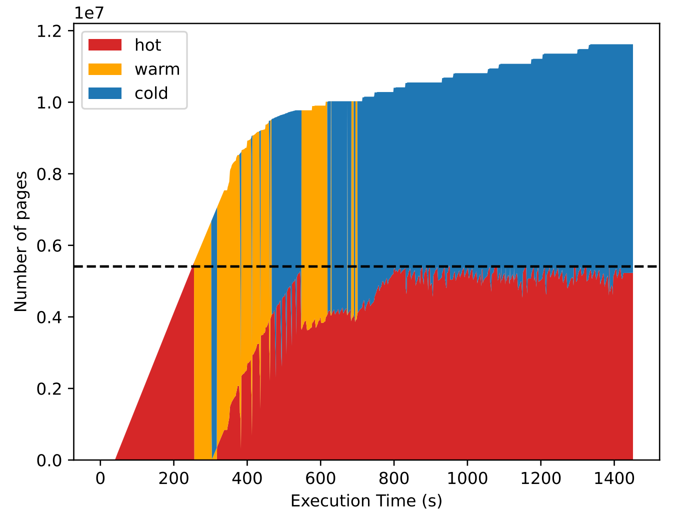
The experiments were conducted on a physical server with two 20 cores/40 threads Xeon Gold 6230N @ 2.30GHz processors. The testbed has 2 NUMA nodes, each with 128GB DRAM and 256GB Optane memory. The baseline and TieredMMS run on CentOS Stream 9 with Linux kernel 5.15.0. For Memtis, the kernel version is v5.15.19.

We choose two representative memory-intensive applications, graph processing (Graph500) and machine learning (Liblinear). Table I shows the brief description and its Resident Set Size (RSS).

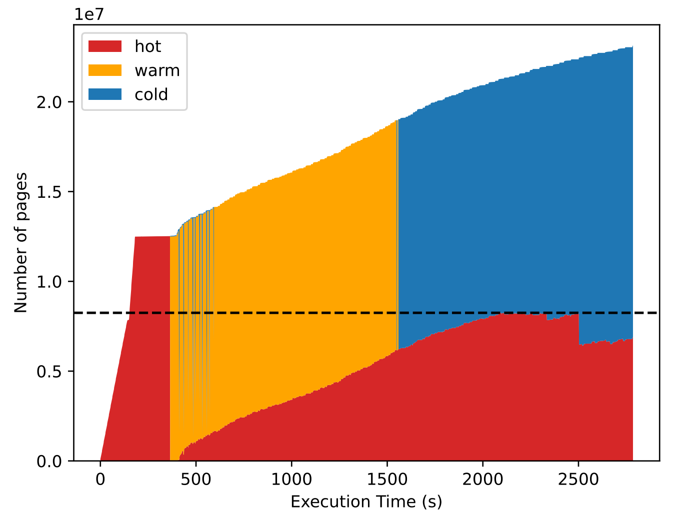
For a tiering memory system, the ratio of the fast memory to slow memory size is an important factor affecting the performance. We explore 3 configurations (1:1, 1:3, 1:7). In the 1:3 configuration, the fast memory size is 25% of the application's resident set size. We modify the ratio by adjusting the kernel boot argument (*memmap* GRUB option [29, 30]) to set a restriction on the size of DRAM. We turn off the transparent huge page and use a single socket for our evaluations to avoid NUMA effects, which are out of the scope of this paper.

B. Effectiveness of Page Classification

Under the 1:1 configuration, we collect the amount of hot, warm, and cold pages reported by TieredMMS. The results are shown in Figure 5. The dashed line indicates the



(a) Liblinear



(b) Graph500

Fig. 5: Amount of hot, cold, and warm pages with page migration.

available memory on the DRAM node. For both Liblinear and Graph500, we can see that the number of hot pages is close to the DRAM size, which is consistent with the design. The difference is that Graph500's hot pages grow very slowly. Because Graph500's hot page ratio is lower than Liblinear's. This is why Graph500's performance improves by 4-5× compared with AutoNUMA, while Liblinear only improves by 2-3× (Section V-C). The warm set, a buffer between hot and cold sets, will see temporary large fluctuations because many hot pages become warm pages before adjusting the thresholds.

Compared with anonymous pages, file pages are accessed much less frequently. Liblinear consumes about 45GB of anonymous pages and about 21GB of file pages. Given their low access frequency, few file pages are collected. This aligns with Figure 5a. Graph500 consumes 67GB of memory, all of

which are anonymous pages. Figure 5b shows that the total size of sampled pages is larger than 67GB due to page freeing and reallocation.

C. Performance Comparison

We compared TieredMMS with AutoNUMA and Memtis [31], a state-of-the-art heterogeneous memory system. AutoNUMA served as the baseline and all results were normalized to it. We ran each test three times and averaged the results. To better understand the performance results, we collected DRAM hit rates in some test cases.

1) Machine Learning: We ran Liblinear [19, 32, 33] on KDD2012 dataset. During the warm-up phase, Liblinear loads the KDD2012 dataset into memory, which performs lots of file I/O, and generates file caches. The Liblinear consumes ∼67GB of memory, of which ∼21GB is the page cache. Since Linux uses a local allocation policy, the page cache consumes a good proportion of local DRAM capacity. When the free memory of the DRAM node is lower than the *high_wm*, TieredMMS migrates the file pages to the non-DRAM node. This ensures that anonymous pages are allocated in DRAM nodes. Previous research has shown that anonymous pages are more critical to application performance than file pages. This is one reason for performance improvement. Another reason is that TieredMMS identifies the hottest anonymous pages and places them on the DRAM node. Figure 6 shows that TieredMMS is 2.08-2.74× the performance of AutoNUMA.

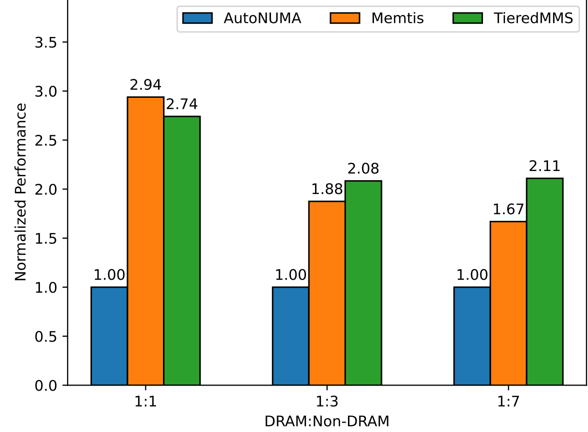


Fig. 6: Liblinear performance.

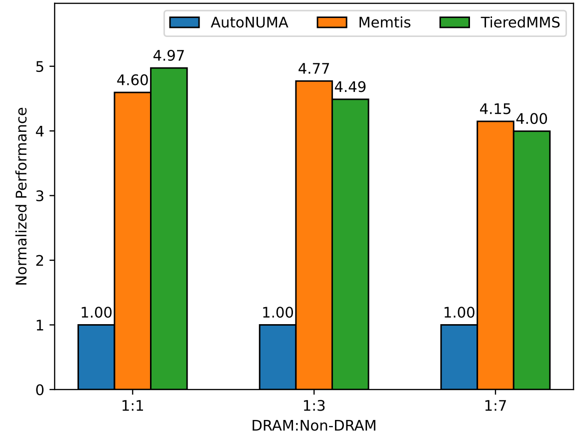


Fig. 7: Graph500 performance.

Compared with Memtis, TieredMMS shows 6.8% worse performance in the 1:1 configuration. In this case, the DRAM capacity is 31GB (OS consumes a part of memory). Half of the anonymous pages fit in the DRAM node. TieredMMS ensures that hot pages remain at the head of the LRU list by frequently adjusting page positions within the LRU. This process involves locking the page by the kernel. For the DRAM node, this mechanism does not offer significant assistance. In the 1:3 and 1:7 configurations, most anonymous pages fit into the non-DRAM, and TieredMMS achieves 10.6% and 26.3% performance improvements, respectively, as shown in Figure 6. For the non-DRAM node, keeping the hottest page at the head of LRU lists helps promotion candidates selection. At the same time, TieredMMS has a more efficient demotion policy. TieredMMS keeps the hottest pages at the head of LRU lists and directly selects the rear 70% of the LRU list for demotion candidates. Since the sampled pages predominantly consist of hot pages, fewer pages are filled into Memtis's demotion list. The majority of elements in the demotion list are derived from traversing the LRU lists and querying the metadata, which involves expensive reverse mapping.

2) Graph Processing: Graph500 [20] generates a graph and conducts a BFS search for 64 randomly selected keys. During the runtime of Graph500, the majority of pages stay in the inactive anonymous list. Unlike Liblinear, Graph500 does not have file pages but a large number of anonymous pages. On the DRAM node, TieredMMS moves hot pages from the inactive list to the active list. TieredMMS periodically isolates pages from the anonymous inactive list and migrates them to the non-DRAM node. In the cooling stage, cold pages are moved from the active list back to the inactive list. On the non-DRAM node, TieredMMS isolates pages from the anonymous active list and migrates them to the DRAM node. Figure 7 shows the performance for Graph500. Compared with AutoNUMA, TieredMMS improves performance by 3.97×, 3.49×, and 3.00× in 1:1, 1:3, and 1:7 configurations, respectively.

Figure 7 shows that TieredMMS outperforms Memtis by 8% in the 1:1 configuration, while in 1:3 and 1:7 configurations, TieredMMS exhibits 5.8% and 3.6% performance deficits, respectively. The primary factor contributing to the performance difference stems from the watermarks and the demotion policy. In the 1:1 configuration, the DRAM capacity is about 42GB (the OS consumes a part of the memory). The *high_wm* and *low_wm* are about 1272MB (3% of DRAM capacity) and 848MB (2% of the DRAM capacity), respectively. Graph generation quickly consumes a large amount of memory. With a 1272MB *high_wm*, TieredMMS starts migrating pages

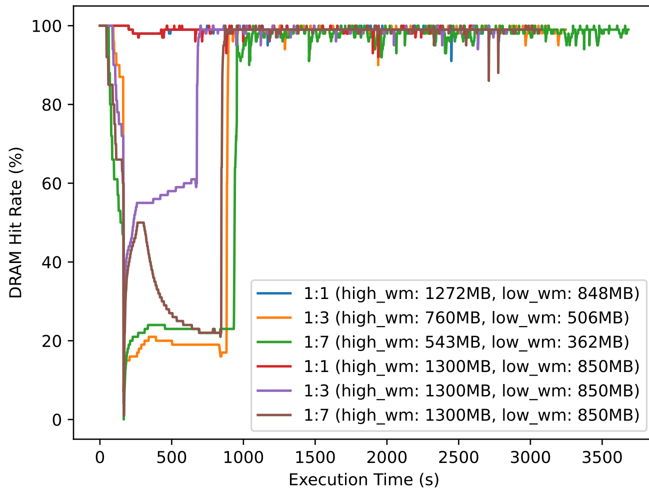


Fig. 8: Graph500 DRAM hit rate.

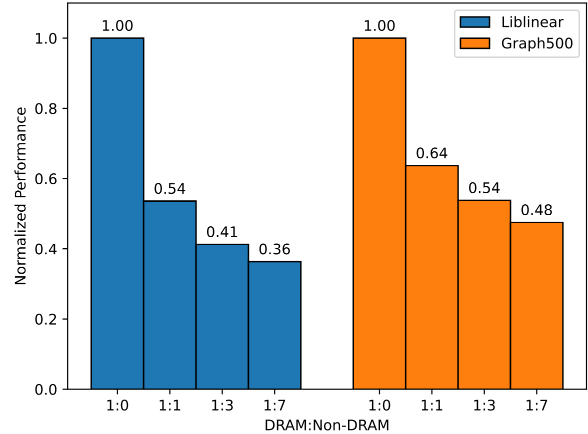


Fig. 9: Comparison with DRAM-only (1:0) system.

to the non-DRAM node at an earlier stage, making space for subsequent anonymous page allocation. Another reason is that TieredMMS employs a page migration strategy that enables faster responsiveness compared with Memtis. Since TieredMMS maintains the most frequently accessed pages at the head of the LRU list, it directly selects the rear 70% of the list for migration to the non-DRAM node during demotion. For Memtis, this process is relatively time-consuming, which is explained when analyzing Liblinear's performance. In the 1:3 and 1:7 configurations, the DRAM capacity is about 25GB and 17GB, respectively. The *high_wm* is about 760MB and 543MB, respectively, and TieredMMS does not have much time to free up space for quickly generated data in the graph generation phase. Consequently, the newly generated data falls into the non-DRAM node. To validate our analysis, we compared the impact of different watermarks on DRAM hit rates. Figure 8 shows that increasing the watermarks results in higher DRAM hit rates during graph generation. When raising watermarks, the performance for the 1:3 and 1:7 configurations increased by 8% and 14%, respectively.

D. Comparison with DRAM-only system

The ultimate performance of a tiered memory system is equivalent to that of a DRAM-only system. This section shows the gap between TieredMMS and capped performance. Since we currently do not have actual CXL-based memory devices, our experiments are based on Intel's Optane DIMM. It can be seen from Figure 9 that there is nearly half or more performance loss when using Intel's Optane DIMM. Previous research has shown that there is a big performance gap between Intel's Optane DIMM and DRAM. The bandwidth gap between DRAM and Optane DIMM is approximately 3-5× and the read latency gap is about 3× [21]. The access latency of CXL-memory is similar to the remote NUMA node [3, 10]. We therefore infer that this gap will continue to narrow as CXL memory devices become more popular.

VI. Related Work

The stock Linux memory management reuses the NUMA infrastructure to cope with multi-tiered memory systems. The NUMA management has some limitations. First, pages are never moved to a CPU-less node and promotion is discarded when the DRAM node is under pressure. Second, Linux divides memory into local and remote without considering the performance differences among heterogeneous memory media. When the local DRAM is exhausted, the fallback path is the local non-DRAM node, even though the remote DRAM node performs better [34]. When local DRAM is under pressure, the *kswapd* becomes busy, consuming significant CPU resources. Even when the swap partition is disabled, the *kswapd* is occupied with writing file-backed memory to the disk. For LRU lists, the oldest pages in the inactive lists are cold but the youngest pages in the active lists are just the most recently used but may not be the most frequently used.

To remedy the shortcomings, previous works have explored various techniques to unlock the potential of the tiered memory system. According to the page access tracking method, these works are classified into several categories: software-based scanning, hardware-based sampling, and a combination of both. Of course, there is also a kind of hardware tiered memory system, such as memory mode Intel Optane DC [1], which is outside the scope of our research.

TPP [10], AutoTiering [11], Nimble [5], Thermostat [35] are software-based solutions, which augment current OS page tracking and page migration mechanism. To solve the problems of existing memory mechanisms, AutoTiering performs page promotion with page eviction if the target node is fully occupied or finds the next best memory node when the DRAM node is fully occupied. TPP defines a new watermark that is higher than *WMARK_HIGH* and has *kswapd* reclaiming pages until free pages reach such watermark. With this watermark, TPP could reserve some memory space for promotion. Nimble introduces efficient two-sided migration, multiple migration

threads, and larger data sizes. Thermostat precisely detects the access frequency of huge pages using page faults, which incur significant tracking overhead. All these solutions are intrusive to the Linux kernel. Moreover, page table scanning incurs performance degradation with large memory sizes. To balance the performance, it requires a long time scan period, which affects accuracy.

Leveraging the processor's hardware event-based sampling to track memory access on tiered memory systems is another approach used by recent works [8, 31, 36, 37]. HeMem [37] and Memtis [31] use hardware-based sampling to analyze the temperature of pages. HeMem is implemented as a user-space library that is transparently linked to applications. It handles page faults using *userfaultfd* which introduces memory copy between userspace and kernel space. Apart from this, two *userfaultfd* patches are required to implement write-protection faults and page-missing faults. Although Memtis outperforms many other tiering memory systems, it is implemented in the kernel and tightly coupled with the Linux kernel. Besides, Memtis samples all processes on all CPU cores. In fact, many memory access operations are not necessary for identifying hot pages, such as kernel objects.

TMTS [8] combines both page table scanning and hardware mechanism. The cold pages are identified by scanning the page access bits. The hot pages are selected by precise event sampling coupled with proactive and periodic scanning of page tables. TMTS is also intrusive to the Linux kernel. On one hand, it defines a custom system call for page promotion. On the other hand, it extends in-kernel page access bits scanner to track page hot age. The vTMM [38] proposed the optimized page table scanning method based on the Intel Page Modification Logging [39] mechanism. However, it only works for the extended page table (EPT) entry.

VII. Discussion

Comparison with Memtis. TieredMMS and Memtis have distinct design objectives that could potentially complement each other. Memtis primarily focuses on page size determination while TieredMMS focuses on easy deployment and maintenance. Memtis modifies several critical kernel objects for its metadata. TieredMMS maintains metadata in userspace, achieving a separation between mechanism and policy, which is not intrusive to the Linux kernel and facilitates maintenance. Memtis samples all events on all CPU cores, which will incur unnecessary overhead. TieredMMS samples the specified process and its child task. Memtis maintains a promotion list for the non-DRAM node and a demotion list for the DRAM node. TieredMMS enhances the LRU lists by leveraging page access information sampled by the hardware sampling mechanism.

Comparison with caching software. OpenCAS [40] and CacheLib [41] are two typical caching software. OpenCAS is a block caching software. It leverages a fast cache device to accelerate the slow backend block devices. CacheLib accelerates applications by providing APIs to manage key-value pairs in the cache. It supports DRAM, NVM, and a mixture of them as the cache device. They are different from TieredMMS. First, they have different interfaces. OpenCAS provides the application with a block interface. CacheLib provides the application with a key-value interface. TieredMMS performs data placement asynchronously without direct interaction with applications. Second, they expose different capacities. OpenCAS and CacheLib use the fast media as cache, and hence the exposed capacity equals the backend device. TieredMMS treats all devices that support load/store semantics as memory devices. The exposed memory capacity is the sum of all types of memory media.

VIII. Conclusion

We propose TieredMMS, a hardware-based and modular tiered memory management system. It comprises a userspace page access tracker and a page migration kernel module, enabling easy deployment. Page access tracker, utilizing hardware sampling, identifies hot/cold pages and signals the page migration module via ring buffers. The page migration module migrates the hottest pages to the DRAM node, optimizing the utilization of diverse memory types. The evaluation shows that TieredMMS outperforms state-of-the-art tiering systems by up to 26.3% in 3/6 tests.

Acknowledgment

We thank the anonymous reviewers and our shepherd, Erez Zadok, for their constructive comments.

References

[1] (2019) Intel optane dc persistent memory. [Online]. Available: http://www.intel.com/optanedcpersistentmemory

[2] (2023) Compute express link™: The breakthrough cpu-to-device interconnect cxl™. [Online]. Available: https://www.computeexpresslink.org/

[3] H. Li, D. S. Berger, L. Hsu, D. Ernst, P. Zardoshti, S. Novakovic, M. Shah, S. Rajadnya, S. Lee, I. Agarwal *et al.*, "Pond: Cxl-based memory pooling systems for cloud platforms," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 2023, pp. 574–587.

[4] S. Bergman, P. Faldu, B. Grot, L. Vilanova, and M. Silberstein, "Reconsidering os memory optimizations in the presence of disaggregated memory," in *Proceedings of the 2022 ACM SIGPLAN International Symposium on Memory Management*, 2022, pp. 1–14.

[5] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, "Nimble page management for tiered memory systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019, pp. 331–345.

[6] A. Maruf, A. Ghosh, J. Bhimani, D. Campello, A. Rudoff, and R. Rangaswami, "Multi-clock: Dynamic tiering for hybrid memory systems," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE Computer Society, 2022, pp. 925–937.

[7] D.-J. Oh, Y. Moon, E. Lee, T. J. Ham, Y. Park, J. W. Lee, and J. H. Ahn, "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*, 2021, pp. 24–36.

[8] P. Duraisamy, W. Xu, S. Hare, R. Rajwar, D. Culler, Z. Xu, J. Fan, C. Kennelly, B. McCloskey, D. Mijailovic *et al.*, "Towards an adaptable systems architecture for memory tiering at

warehouse-scale," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 727–741.

[9] J. Weiner, N. Agarwal, D. Schatzberg, L. Yang, H. Wang, B. Sanouillet, B. Sharma, T. Heo, M. Jain, C. Tang *et al.*, "Tmo: transparent memory offloading in datacenters," in *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2022, pp. 609–621.

[10] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "Tpp: Transparent page placement for cxl-enabled tiered-memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 742–755.

[11] J. Kim, W. Choe, and J. Ahn, "Exploring the design space of page management for multi-tiered memory systems," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 715–728.

[12] S. R. Dulloor, A. Roy, Z. Zhao, N. Sundaram, N. Satish, R. Sankaran, J. Jackson, and K. Schwan, "Data tiering in heterogeneous memory systems," in *Proceedings of the Eleventh European Conference on Computer Systems*, 2016, pp. 1–16.

[13] C. Wang, H. Cui, T. Cao, J. Zigman, H. Volos, O. Mutlu, F. Lv, X. Feng, and G. H. Xu, "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*, 2019, pp. 347–362.

[14] J. Ren, J. Luo, K. Wu, M. Zhang, H. Jeon, and D. Li, "Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2021, pp. 598–611.

[15] M. Hildebrand, J. Khan, S. Trika, J. Lowe-Power, and V. Akella, "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*, 2020, pp. 875–890.

[16] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, "Heteroos: Os design for heterogeneous memory management in datacenters," *ACM SIGOPS Operating Systems Review*, vol. 52, no. 1, pp. 13–26, 2018.

[17] V. Gupta, M. Lee, and K. Schwan, "Heterovisor: Exploiting resource heterogeneity to enhance the elasticity of cloud platforms," *ACM SIGPLAN Notices*, vol. 50, no. 7, pp. 79–92, 2015.

[18] Z. Li and M. Wu, "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*, 2022, pp. 72–80.

[19] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin, "Liblinear: A library for large linear classification," *the Journal of machine Learning research*, vol. 9, pp. 1871–1874, 2008.

[20] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, vol. 19, pp. 45–74, 2010.

[21] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor *et al.*, "Basic performance measurements of the intel optane dc persistent memory module," *arXiv preprint arXiv:1903.05714*, 2019.

[22] T. Hirofuchi and R. Takano, "A prompt report on the performance of intel optane dc persistent memory module," *IEICE TRANSACTIONS on Information and Systems*, vol. 103, no. 5, pp. 1168–1172, 2020.

[23] X. Wei, X. Xie, R. Chen, H. Chen, and B. Zang, "Characterizing and optimizing remote persistent memory with rdma and nvm," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 2021, pp. 523–536.

[24] Intel. (2023) Intel® 64 and ia-32 architectures software developer's manual. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[25] (2023) A thorough introduction to ebpf. [Online]. Available: https://lwn.net/Articles/740157/

[26] (2023) Bpf documentation. [Online]. Available: https://www.kernel.org/doc/html/latest/bpf/index.html

[27] N. Amit and M. Wei, "The design and implementation of hyperupcalls," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 97–112.

[28] (2023) Libsvm data: Classification (binary class). [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/binary.html

[29] (2023) Using the memmap kernel option. [Online]. Available: https://docs.pmem.io/persistent-memory/getting-started-guide/creating-development-environments/linux-environments/linux-memmap

[30] (2023) The kernel's command-line parameters. [Online]. Available: https://www.kernel.org/doc/html/latest/admin-guide/kernel-parameters.html

[31] T. Lee, S. K. Monga, C. Min, and Y. I. Eom, "Memtis: Efficient memory tiering with dynamic page classification and page size determination," in *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023, pp. 17–34.

[32] (2023) Liblinear – a library for large linear classification. [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/liblinear/

[33] (2023) Multi-core liblinear. [Online]. Available: https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/multicore-liblinear/

[34] (2023) What is numa? [Online]. Available: https://www.kernel.org/doc/Documentation/vm/numa

[35] N. Agarwal and T. F. Wenisch, "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*, 2017, pp. 631–644.

[36] J. Choi, S. Blagodurov, and H.-W. Tseng, "Dancing in the dark: Profiling for tiered memory," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2021, pp. 13–22.

[37] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, "Hemem: Scalable tiered memory management for big data applications and real nvm," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 392–407.

[38] S. Sha, C. Li, Y. Luo, X. Wang, and Z. Wang, "vtmm: Tiered memory management for virtual machines," in *Proceedings of the Eighteenth European Conference on Computer Systems*, 2023, pp. 283–297.

[39] (2023) Page modification logging for virtual machine monitor white paper. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/page-modificationlogging-vmm-white-paper.pdf

[40] (2024) Opencas. [Online]. Available: https://open-cas.github.io/

[41] (2024) Cachelib. [Online]. Available: https://cachelib.org/docs/Cache_Library_User_Guides/About_CacheLib