vTMM: Tiered Memory Management for Virtual Machines

Abstract

The memory demand of virtual machines (VMs) is increasing, while the traditional DRAM-only memory system has limited capacity and high power consumption. The tiered memory system can effectively expand the memory capacity and increase the cost efficiency. Virtualization introduces new challenges for memory tiering, specifically enforcing performance isolation, minimizing the costly context, and providing resource overcommit. However, none of the state-of-the-art designs consider virtualization and thus address these challenges; we observe that a VM with tiered memory incurs up to a 2× slowdown compared to a DRAM-only VM.

This paper proposes *vTMM*, a tiered memory management system specifically designed for virtualization. *vTMM* automatically determines page hotness and migrates pages between fast and slow memory to achieve better performance. A key insight in *vTMM* is to leverage the unique system characteristics in virtualization to meet the above challenges. Specifically, *vTMM* tracks memory accesses with page-modification logging (PML) and a multi-level queue design. Next, *vTMM* quantifies the page "temperature" and make a fine-grained page classification with bucket-sorting. *vTMM* performs page migration with PML while providing resource overcommit by transparently resizing VM memory through the two-dimensional page table. In combination, the above techniques minimize overhead, ensure performance isolation and provide dynamic memory partitioning to improve the overall system performance.

We evaluate *vTMM* on a real DRAM+NVM system and a simulated CXL-Memory system. The results show that *vTMM* outperforms NUMA balancing, Intel Optane memory mode and *Nimble* (an OS-level tiered memory management) for VM tiered memory management. Multi-VM co-running results show that *vTMM* improves the performance of a DRAM+NVM system by 50% − 140% and a CXL-Memory system by 16% − 40%, respectively.

Keywords: tiered memory, virtual machine, hot set, PML

1 Introduction

Traditional homogeneous CPU-attached DRAM-only memory systems (hereinafter, **DRAM-only systems**) have limited capacity and high power consumption, which can no longer meet the increasing memory demand of virtual machines (VMs) in data centers. In fact, DRAM memory consumes about 40% of power in modern data centers [17, 34, 39]. This has motivated several memory technique alternative proposals. Large-capacity non-volatile memory (NVM) is an effective alternative to DRAM. NVM has no flush and idle power consumption and supports byte-addressable access [43]. In addition, Compute Express Link (CXL) [10], as a new protocol, allows a new memory bus interface to link memory to the CPU [10]. In the future, CXL will be an effective means to expand memory capacity. However, compared to the directly attached DRAM memory, both NVM and CXL-Memory come with lower bandwidth and longer latency [18, 29, 36, 40, 45]. From an operating system (OS) perspective, either NVM or CXL-Memory can be a CPU-free NUMA node, with memory features (such as capacity, bandwidth, latency, etc.) independent of the DRAM memory directly attached to the CPU [10, 25].

The fast memory (FMem) and slow memory (SMem) make up a tiered memory system. The researchers focus on putting active (hot) pages in FMem and inactive (cold) pages in SMem for the best performance. The classic management approach achieves the goal by page tracking, classifying, and migration. Following this design, many advanced techniques have been developed, such as *Thermostat* [1], *HeMem* [40], *Nimble* [44], *HeteroOS* [30], *RAMinate* [22], etc. Unfortunately, all of them cannot provide efficient management for virtualization. Tiered memory management suffers from new challenges in virtualization. First, multi-VM co-running can generate intensive memory competition while performance isolation between VMs is essential. For example, by using total DRAM as a direct-mapped cache, the Intel Optane memory mode (*MM*) [25] can hide the latency of PMem access (§2.1). However, the experiments show that multi-VM co-running can increase DRAM cache misses by 2× to 4× than a standalone VM, because of severe DRAM cache pollution (§4.9.2). Second, the virtualization introduces VM context, and we should minimize expensive VMTraps (i.e., context switch between guest and host). For example, the write protection-based page migration adopted by *HeMem*, would incur heavy VMTraps due to write exception handling. Third, cloud applications in mutiple co-run VMs can exhibit more diverse memory access patterns, which poses new challenges for page placement policy of multi-VMs. Finally, VM memory overcommit [21] in a tiered memory system needs to handle a new design space when compared to the traditional DRAM-only system. The large-capacity SMem guarantees sufficient VM memory capacity, but the FMem is still scarce. Ideal management should dynamically balance FMem across multiple VMs for better overall performance.

Meanwhile, virtualization introduces new opportunities for tiered memory management. (1) We leverage hardware-assisted virtualization technologies to optimize our designs.

Intel page-modification logging (PML) is designed for tracking dirty pages for VM live migration [4, 27]. By using PML, we can trace the accessed (dirty) page tables rather than the entire page tables to improve memory access tracking efficiency. Moreover, PML also can efficiently handle dirty pages with few VMTraps when migrating pages between FMem and SMem. (2) Memory virtualization introduces two-dimensional (2D) address mapping. By manipulating the mappings of guest physical addresses to host physical addresses, we can transparently resize VM FMem according to the memory access patterns. Particularly, we can utilize page migration and remapping rather than resorting to the conventional inefficient ballooning mechanism [23] to achieve dynamic memory partitioning between the multiple VMs.

In this paper, we present *vTMM*, a tiered memory management system for virtualization. Figure 1 shows the high-level overview of *vTMM*. *vTMM* manages tiered memory in the hypervisor and follows the classic flow (i.e., page tracking, classifying, and migration) to manage tiered memory, but with novel designs in every step, specifically designed for virtualization. First, *vTMM* tracks VM page accesses by periodically scanning guest page tables (GPTs) to record and clear accessed/dirty (A/D) bits. *vTMM* only scans dirty GPTs logged by PML instead of the entire GPTs, which effectively reduces the scanning time. Even with this optimization, scanning page tables can still incur a non-negligible overhead because of setting A/D bits and TLB shootdowns for pages. *vTMM* adopts a multi-level queue, where more frequently accessed pages are placed in higher levels and tracked less frequently. This locality-based design can further reduce tracking cost while preserving memory access patterns.

Second, the application access patterns in the VMs are more diverse. Hence to place a page, *vTMM* screens out the hottest pages and flexibly determines the hot set according to the VM FMem size by sorting pages. *vTMM* adopts a fine-grain approach that measures the *page-degree* (i.e., the "temperature" of a page) and bucket-sorts VM pages based on *page-degree*s. The *page-degree* of a page is a weighted combination of its read and write frequency where the weights are derived from read and write costs, respectively.

Next, vTMM exchanges the cold pages in FMem with the hot pages in SMem to carry out the page placement policy. By leveraging PML, *vTMM* copies pages in parallel and handles dirty pages with a minimal VM page access pause. In addition, *vTMM* actively fills the new mappings of Intel extended page table (EPT) after migration to avoid VMTraps caused by the EPT page faults [27, 37].

Finally, *vTMM* manages co-running VMs and partitions memory dynamically by utilizing a shared a memory pool. *vTMM* extracts the surplus FMem from some VMs and allocates it to the VMs with insufficient FMem. *vTMM* hence improves FMem utilization and overall performance. *vTMM* utilizes page migration instead of ballooning to efficiently and transparently adjust VM memory for supporting memory overcommit in the tiered memory system.

We implement and evaluate *vTMM* in both a real DRAM+NVM system and a simulated CXL-Memory system. In addition, we extend *vTMM* to support transparent huge page (THP) management. The multi-VM experimental results show that in the regular page system, compared with *AutoNUMA* where VMs only enable Linux NUMA balancing [20] without other management, *vTMM* provides 51% and 16% higher performance for DRAM+NVM system and CXL-Memory system, respectively. In the huge page system, *vTMM* achieves 143% (DRAM+NVM system) and 40% (CXL-Memory system) higher performance than *AutoNUMA*. *vTMM* outperforms *Nimble Page Management* [44] by 615% (DRAM+NVM) and 50% (CXL-Memory). Meanwhile, *vTMM* outperforms *MM* by 31% and 14%, respectively, for regular page and huge page in DRAM+NVM system.

In summary, the contributions of our work are:

- We propose a PML-based GPT scanner to track the pages of the active processes only and design a multi-level queue to reduce page tracking overhead.
- We propose a hot/cold page classifier based on sorting, which can adapt to various memory access patterns.
- We migrate pages in parallel with a minimal access pause and handle dirty pages cleverly by utilizing PML.
- We propose a memory pool to dynamiclly partition memory among VMs to achieve higher performance for VMs co-running.
- We extend *vTMM* to support THP, as well as, extensively evaluate and compare it with several state-of-the-art designs on a DRAM+NVM system and a CXL-Memory system, respectively.

2 Background and Motivation

2.1 Tiered Memory Management

A tiered memory system consists of different memory (such as DRAM, NVM, CXL-Memory) with different features (bandwidth, latency, power consumption, etc.). We take DRAM+NVM system and CXL-Memory system as examples to discuss tiered memory management.

DRAM+NVM system. Intel Optane DC PMem[1] is the first and only commercially-available NVM product [25]. PMem increases total memory capacity by up to 8×, but as shown in Table 1, it comes at cost of up to 6× lower bandwidth and up to twice the latency compared with DRAM [18, 29, 40, 45]. Moreover, NVM latency and bandwidth show significant asymmetry for read and write.

CXL-Memory system. CXL is a new industry-supported Cache-Coherent interconnection protocol for processors, memory expansions, and accelerators [10]. CXL-Memory supports expanding main memory for the CPU (including

[1]Although Intel has shut down the PMem production, it does not affect our research and evaluation with PMem.

DRAM and NVM), but with lower bandwidth and higher latency than the directly connected memory. The access latency and bandwidth to CXL-Memory is similar to the remote latency and bandwidth on a dual-socket system [36].

Table 1. DRAM and PMem latency and bandwidth.

Memory	R/W Latency(ns)	R/W BW(GB/s)
DRAM	81 / 82	120 / 82
CXL-Memory	153/ 162	97 / 63
Intel PMem	310 / 94	37 / 13

To explore the potential of tiered memory system in virtualization, we execute popular applications on a VM configured with 8GB DRAM and 32GB PMem. The configuration of the programs is described in §4. As shown in Table 2, we observe that there is a more than 2× slowdown compared to the DRAM-only system when just enabling *AutoNUMA* in VM without other management. The result indicates the traditional Linux memory management, designed for DRAM-only system, is no longer suitable for the tiered memory system.

Table 2. Slowdown of tiered-memory VM with AutoNUMA.

Benchmarks	DRAM-only	AutoNUMA	Slowdown
649.fotonik3d_s	262 s	885 s	237.79%
random access	801 s	2404 s	200.12%
graph500	378 s	893 s	136.24%
redis	29296 opt/s	22110 opt/s	24.53%

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, classifiying hot/cold pages and automatically migrating pages between FMem and SMem.

2.2 State-of-the-art and limitations

Table 3 compares *vTMM* with several state-of-art designs of tiered memory management and analyzes key components including page tracking, classifying and migration, etc.

Page Tracking: Page tracking captures memory access information for memory management. Page table (PT) scanning traces memory access by checking and clearing accessed/dirty bits periodically. In particular, targeting a virtualized system, *RAMinate* [22] scans the EPT, called EPT scanner. However, without a proper optimization, PT scanning can cause unacceptable overhead because of the extra MMU pressure. In addition, *DAMON* [31] is a Linux kernel memory data access monitoring framework, which also checks accessed bits to monitor pages. It adopts region sampling to reduce overhead, but it cannot determine the number of regions for different applications to control the optimal accuracy and overhead. *Thermostat* [1] tracks pages using page faults (via BadgerTrap [19]). However, it only randomly profiles 0.5% of total memory and it causes up to several times slowdown if all pages are profiled [19]. Similarly, *autotiering* [32] tracks memory accesses for multi-tiered memory through NUMA page fault monitoring. For *HeMem* [40], it traces pages via Intel PEBS [26] that is not friendly for virtualizaion. Due to the security of the virtualization, the hypervisor cannot trace VM memory access by using PEBS. The PEBS facility cannot store guest linear addresses on the host side, because guest cannot write to the host's PEBS buffer (indexed by the host virtual address). In addition, the PEBS overhead is unacceptable for regular page online tracing [1].

Page Classification: We classify pages based on the access patterns to determine the hot set to be placed in FMem. Existing designs distinguish hot and cold pages by using the LRU algorithm [22, 30, 32, 44] or a fixed threshold method [1, 40]. LRU is used to swap out inactive (cold) pages in the Linux memory subsystem. However, it is not suitable for tracking hot pages because it focuses on the sequence of page access rather than the access frequency. We will show this through experiments in §4.8. We observe that using a fixed threshold cannot adapt to the diverse memory access patterns (§4.5). For anonymous page classification, *autotiering* uses access count ranking, an idea similar to our design. However, it treats reads and writes uniformly. We should weigh reads and writes according to their different performance features especially for the bandwidth.

Page Migration: It is the enforcer of the page placement policy. Raw Linux serially migrates pages and *Nimble* provides high THP migration throughput through parallel and page exchange. Autotiering [32] adopts the same design as Nimble. Both of them incurs long page access pause during migration because of unmapping pages before copying data. *HeMem* uses page write-protection to reduce page access pause during migration, but in virtualization, write-protection triggers expensive VMTraps, like shadow paging synchronization [42].

Dynamic FMem Partitioning: For multi-VM tiered memory management, we should consider memory partitioning to improve performance and utilization. Although *HeterOS* [30] discusses extending the ballooning mechanism for memory overcommit, it does not implement the dynamic partitioning. In addition, the slow ballooning is no longer appropriate for tiered memory management, especially for adjusting regular pages [23], and gOS needs to be customized to distinguish FMem and SMem.

Summary: Overall, none of the existing work is specifically designed for virtualization. (1) They do not support virtualization or they trigger lots of overhead for memory tracking and during page migration; (2) They lack a more adaptable strategy for distinguishing between hot and cold pages for diverse VM applications. (3) They do not perform efficient memory dynamic partitioning. In contrast, *vTMM* is designed for VM tiered memory management and utilizes virtualization features to optimize page manipulations and reduce management overhead. *vTMM* tracks pages with

Table 3. Comparison of tiered memory management

Designs	page tracking	classifying (weight R/W)	page migration	D-partitioning
RAMinate	raw PT scanning	LRU queues (N)	low-speed (raw Linux migration)	N
HeteroOS	ditto	LRU of Linux (N)	ditto	N
Thermostat	page fault	fixed threshold (N)	ditto	N
Nimble	raw PT scanning	LRU of Linux (N)	high-speed but long access pause	N
Autotiering	NUMA page fualt	LRU,sorting (N)	ditto	N
HeMem	Intel PEBS	fixed threshold (Y)	high-speed but heavy VMTraps	N
vTMM	optimized PT scanning	adaptive sorting (Y)	high-speed and few VMTraps	Y

PT scanning, but leverages a multi-level queue to reduce overhead. It classifies hot/cold pages based on page "temperature" sorting and migrates pages using PML for minimal access pause and little virtualization performance loss. Finally, *vTMM* dynamically partitions small FMem among VMs by leveraging page migration and rmapping for improving utilization and overall performance when multi VMs co-run.

2.3 Page-Modification Logging (PML)

In our work, the Intel PML mechanism is utilized twice: capturing the A/D bit-settings of GPTs (§3.2) and optimizing dirty handling of page migration like VM live migration (§3.4). Intel PML is a hardware-assisted virtualization technology, which tracks dirty pages for VM live migration [4, 12]. Before the advent of PML, the hypervisor write-protects all VM pages, and modifying migrated pages triggers expensive VMTraps to trace dirty pages.

Before a guest-physical access, the processor may determine that it first needs to set an accessed or dirty bit for its EPT entry (EPTE) [27]. If PML is enabled, when the dirty bit is set, the MMU logs the guest physical address (GPA) of the accessed regular/huge page to a PML buffer. There is a page-modification log-full event and a VM Exit occurs [27]. In particular, the hypervisor flushes the PML buffer to update the dirty page bitmap in every VM Exit [7]. The hypervisor needs to clear the dirty bits in EPT first to ensure that writing a page will set its dirty bit. In addition, the hypervisor also needs to flush the corresponding TLBs (including regular TLB and ETLB [27]) to enforce the 2D page walk. All in all, Intel PML can track dirty pages of a VM in hardware without triggering expensive VMTraps.

3 vTMM Design and Implementation

3.1 Overview of vTMM

vTMM is a hypervisor-based tiered memory management system. Figure 1 shows the high-level overview of *vTMM*. We focus on exploiting the performance potential of the tiered memory system by utilizing virtualization hardware support (like PML) and software features (like 2D mappings). *vTMM* tracks VM page access by scanning GPTs and reduces the cost by utilizing Intel PML and a multi-level queue. Then, according to the read/write frequency, *vTMM* sets the "temperature"s of the pages and sorts them to guide the VM page placement. *vTMM* then exchanges the cold pages on FMem with the hot pages on SMem using PML-optimized page migration. *vTMM* minimizes page access pauses in page migration with little additional virtualization overhead (aka VMTraps). Further, *vTMM* provides performance isolation for co-running VMs, as well as, dynamically partitions memory to achieves higher utilization and overall performance.

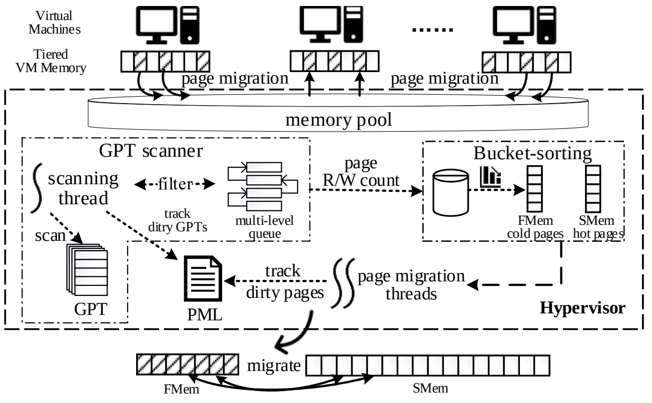

Figure 1. *vTMM* system design (including the GPT scanner with a multi-level queue for page tracking, the bucket-sorting policy for page classifying, PML-based parallel page migration threads and the memory pool for dynamic partitioning).

3.2 PML-based VM Page Tracking

In the page memory management, setting A/D bits in the page tables is the classic mechanism to transfer memory access information from hardware to software. However, traditional page table scanning, such as the EPT scanner adopted by *RAMinate*, suffers from two challenges: (1) The large memory footprint of the VM results in massive page table to scan every time regardless of the working set size of the applications running on the VM. Because the EPT scanner has no idea about which pages are accessed, it has to scan the entire VM mappings to examine A/D bits of pages, which results in a long scan time. (2) Setting A/D bits and flushing TLB can slow down memory accesses notably, especially in virtualization with costly 2D address translation.

3.2.1 PML-optimized GPT scanner. To address challenge (1), *vTMM* utilizes PML to track accessed GPT pages, rather than scanning the entire EPT of the VM. *vTMM* designs a scan kernel module in the VM without modifying the gOS native code. The scan module first scans the GPTs of the processes of the VM and stores the last-level guest page table (LL-GPT) pointers to a buffer, call the LL-GPTP buffer. We use hashing to filter duplicated pointers due to page sharing. With a *hypercall*, the address of the LL-GPTP buffer is passed to the hypervisor.

At initialization, the GPT scanner first clears the A/D bits of each guest page table entry (PTE) by accessing the last-level GPT pages. Then, the GPT scanner clears the corresponding dirty bits in the EPT and flushes TLBs for those GPT pages. This is to ensure that modification of GPTs can be logged by PML, as described in §2.3. The GPT scanner in the hypervisor scans the GPT periodically and the interval between two scans is called a monitoring window. We conduct a sensitivity study of the monitoring window size (MWS) in §4.3. During a monitoring window, the GPT scanner captures the PML log from the PML buffers in real time, and extracts the GPT page addresses from the log. At the end of the monitoring window, the GPT scanner scans those GPT pages logged by PML, records A/D bits that are set and clears A/D bits for only a subset of pages determined by the multi-level queue filter discussed next. Just like initialization, the GPT scanner clears the corresponding dirty bits in EPT and flush TLBs for those GPT pages. After several rounds of monitoring (8 in our evaluation), the GPT scanner records the number of reads and writes per VM page. The pages that are not accessed do not incur A/D bits setting. Thus, the GPT scanner only scans the active GPT pages, which greatly reduces the amount of page table scanning. In addition, VM OS may check A/D bits for memory management and PT scanning may interfere with mechanisms like disk swap, but they are rarely used in VMs.

3.2.2 Multi-level queue.

In order to reduce the overhead of setting A/D bits and flushing TLB (challenge (2)), we design a multi-level queue algorithm. Frequent setting A/D bits and flushing TLB for hot pages are major overhead. According to locality [24, 38], a page that has been accessed is likely to be accessed several times in the future. Based on this motivation, the multi-level queue relaxes monitoring of hot pages based on their access history.

In a monitoring window, *vTMM* filters each page through a multi-level queue which determines whether or not to clear the page's A/D bits. The multi-level queue handles time events in the unit of monitoring window. At the beginning, all pages are in the level 0. The A/D bits of the pages in level 0, need to be cleared right now. For level x ($x > 0$), we define the *do-not-disturb* (DND) time is 2^{x-1}, which that means the A/D bits of the pages in level x will not be cleared until 2^{x-1} intervals later. The pages in their DND time are treated as continuous visits. When a page's DND time ends, its A/D bits should be cleared in this interval. In the next interval, if this page is set A/D bits again, it suggests the page is active (hot). Thus, we upgrade its level to make its DND time longer. Otherwise, it means that the page has not been accessed continuously, so we degrade its level to reduce its DND time. If the level of a page drops to 0, we clear the A/D bits for the page. The hot pages have a long DND time, and the queue filters out their A/D bits setting, and estimates access counting, while the cold pages in low-level queues have a short DND time, which avoids being mistakenly filtered out and missing counts.

3.2.3 Intermittent monitoring.

It is not necessary to enable memory tracking all the time, especially when the VM memory access is stable. Therefore, *vTMM* turns on VM page tracking periodically through a clock switch. It is not practical to determine a perfect interval. Experimentally, we turn on every minute for regular page VMs and 2 minutes for VMs enabling THP. In addition, *vTMM* also collects the history of page migration with counters for VMs to determine whether tracking is enabled next period. When the memory migrated is under a threshold (256 MB in our evaluation), *vTMM* increases the counter up to an upper limit (3 in our evaluation). When the clock switch is on, the counter decreases by one. The page tracking of a VM is enabled only if its counter is 0. The feedback mechanism avoids unhelpful page tracking and migration when most pages have been placed correctly.

3.3 Hot Page or Cold Page?

Page tracking counts read and write frequency for each VM page. Then, *vTMM* quantifies the "temperature" of a page (i.e., page-degree) and bucket-sorts VM pages based on *page-degree* in order to determine the VM hot set.

3.3.1 Page-degree.

As shown in Table 1, there is an significant gap between read and write on bandwidth and latency, especially in NVM (PMem). To better distinguish between hot pages and cold pages, we weight read and write differently. We use micro-benchmarks (including random read/write and sequential read/write) to estimate the weights of read and write. We use the ratio of read-slowdown to write-slowdown as the ratio of the read and write weights. Specifically, read (or write) slowdown is calculated by the difference of execution time of the read (or write) benchmarks running on FMem and SMem. We also repeat the experiment for programs with different working set sizes. With a DRAM+PMem memory system, we calculated the weights $read\_wight : write\_wight = 1 : 3$; In the NUMA-simulated CXL-Memory system composed of local DRAM and remote DRAM, the result is 2 : 3. Further, we define *page-degree* to be $read\_wight * read\_count + write\_wight * write\_count$. Thus, we quantify page access.

3.3.2 Degree-sorting: distinguishing hot/cold pages.

vTMM sorts all pages of the VM based on the *page-degree*. According to the formula above, the *page-degree* is a integer with finite range. Thus, *vTMM* applies the bucket sorting algorithm [6]. The sorting is done by scanning all the pages at once ($O(n)$). The hot set is built according to the FMem capacity of the VM. Then, by using **PML-based page migration**, *vTMM* exchanges SMem pages in the hot set with the same number of the coldest pages of FMem.

3.4 Page Migration Based on PML

Page migration focuses on two issues: the migration speed and the slowdown caused by page access pause during page migration. From a software perspective, either NVM or CXL-Memory can be mounted as a CPU-free NUMA node to expand main memory [25, 36]. In fact, Linux kernel provides a system API [8] for migrating pages between two NUMA nodes. However, it adopts serial migration thus has low speed. *Nimble*'s parallel and concurrent THP migration significantly increases throughput. However, *Nimble* unmaps pages before copying data to ensure consistency, which results in long pauses of page accesses. Then, *HeMem* write-protects pages to be migrated, and reading pages will be not interrupted during the migration. However, write protection is not virtualization friendly because of triggering heavy VMTraps.

In contrast, *vTMM* migrates pages in parallel (4 threads in our evaluation) with a negligible page access pause, and there are few VMTraps because of leveraging PML. As described in §2.3, the PML can efficiently track dirty pages for VM live-migration. Similarly, we utilize PML to track dirty pages for page migration between of FMem and SMem.

After migration preparation, *vTMM* first applies for new pages from the target NUMA node. *vTMM* pre-applies some free pages in parallel when tracking VM pages, which speed up allocation of the new pages, especially for THP. Second, *vTMM* cleans the D bits in EPT for the old pages to be migrated and flushes the corresponding TLB entries to ensure that the PML can track them. Third, *vTMM* copies the data from the old pages to the new pages. Then, *vTMM* unmaps the old pages and update the dirty page bitmap by checking the PML logs. For non-dirty pages, *vTMM* sets up their new mappings directly. For dirty pages, *vTMM* recopies them. By now, the mappings of dirty pages have been removed, so recopying does not incur dirty pages again. After that, *vTMM* sets up their new mappings. In addition, we carefully control the number of pages migrated in parallel to mitigate the impact on VMs due to bandwidth usage.

We actively fill the mappings of new pages in EPT after migration. To build the new page mappings, the system has to remove old mappings in the page table of QEMU and the EPT. However, remapping new pages with system API only restores the mappings in the page table of QEMU, but the mappings in EPT will be only established in the hypervisor when the VM accesses the pages again, which leads to expensive VMTraps. Thus, *vTMM* actively fills the mappings of new pages in EPT.

3.5 Memory Pool for Dynamic Partitioning

Memory overcommit in a virtualized tiered memory system suffers from new challenges. The large-capacity SMem guarantees the total VM memory capacity, but fast FMem is scarce. Efficient use of FMem is the key to maintain overall performance. Static memory allocation may incur unbalancing and low utilization. *vTMM* achieves FMem partitioning across multiple VMs with a memory pool that holds free FMem. A VM can obtain FMem from the pool or release FMem to the pool.

The first step in page placement strategy is to determine the hot set for each VM. *vTMM* obtains the *page-degree* distribution of each VM by bucket sorting. *vTMM* defines the hot set as the top 80% of pages whose page-degree is above a base threshold (We use an empirical value of 3 in evaluation). Sorting ensures that the selected pages are always the hottest, and the base threshold ensures that a VM with many inactive pages will not be identified to have a big hot set. *vTMM* sets the upper and lower FMem limits for each VM, which can be flexibly configured by the administrator. In our experiment, we define 75% and 125% of the initial FMem size as the lower and upper limits, respectively. *vTMM* specifies that the hot page size (*hss*) is limited between the upper and lower limits. If the *hss* is bigger than the FMem size, *vTMM* increases the FMem size to the *hss*, when the FMem of pool is enough. If not enough, *vTMM* takes as much as the pool can supply and increases the FMem size accordingly. By migrating pages of VM SMem to the FMem pool and remapping the new FMem pages to the VM, *vTMM* increases the VM FMem capacity. Because the pages of pool are free, the migration is one-way. If the *hss* of the VM is less than current FMem size, *vTMM* reduces the FMem size to the *hss* and releases FMem to the memory pool. Similarly, *vTMM* migrates VM FMem pages to the SMem and remaps the SMem pages to the VM.

3.6 Implementation

vTMM is a pure software system and we implement *vTMM* in 6500 lines of C code. *vTMM* adapts the QEMU/KVM [9] as virtualization architecture. We implement *vTMM* in the KVM module of Linux kernel (5.4.142) except that an initialization scan module of page tracking is implemented in the VM as a kernel module.

THP Support. THP accelerates memory access, especially for programs with a big memory footprint. We extend *vTMM* to support THP and evaluate big memory VMs enabling THP (§4.8). PML supports VMs that enable THP, so the design of *vTMM* can be directly extended to huge page-grained system by simply upgrading the tracked page tables and page granularity in page tracking and page migration. In addtion, huge page issues such as memory bloat and fragmentation are beyond the scope of this paper.

4 Evaluation

4.1 Experimental Setup

We run our evaluation on a dual-socket Intel Cascade Lake-SP system running at 2.2 GHz with 24 cores/48 threads per socket. All VMs' CPUs are pinned to a single socket. Both the

physical machine and the VMs run Ubuntu 18.04 with Linux Kernel 5.4.142. In addition, for the experiments in regular page and huge page systems, the VMs are configured with 8-cores and 16-cores, respectively. We pre-allocate memory for each VM with two NUMA nodes. Initially, VM memory is allocated from the FMem and SMem node of the physical machine, respectively. We build two tiered-memory systems.
(1) **Real DRAM+NVM system** has 32 GB of local DRAM (FMem) and 192 GB PMem (SMem).
(2) **NUMA-simulated CXL-Memory system** has 32GB local DRAM (FMem) and 192 GB of remote DRAM (SMem).

4.2 Experimental Approach

We evaluate *vTMM* modularly and compare the key components with existing designs summarized in §2.2. We evaluate *vTMM* in both real DRAM+NVM system and NUMA-simulated CXL-Memory system. The results are basically the same. Limited by space, we mainly show the results of the former, but we also present the multi-VM co-running results of CXL-Memory system.

Page tracking. We evaluate two key attributes of page tracking: overhead and accuracy (§4.3). As described in §2.2, related designs include PT scanning, page fault and PEBS. PEBS is difficult to service virtualization and tracing pages by using page fault will cause a huge overhead for fine-grained page management [41]. We hence choose to compare our design to the PT scanning (called EPT scanner) .

Page classification. It is not suitable for evaluating online separately but should be integrated into the system. We compare our design with fixed threshold mechanism through an ablation study (§4.5). We demonstrate the advantages of our design over LRU by comparing *vTMM* with *Nimble Page Management (Nimble)* [44] which relies on LRU for page classifying (§4.8 and §4.9.2).

Page migration. Page migration pursues two points: faster migration speed and less VM performance slowdown. We compare our design with raw Linux page migration and write protection-based page migration (§4.4).

Dynamic memory partitioning. In order to verify the memory pool mechanism for dynamic memory partitioning of multiple VMs, we validate the performance improvement when memory pool is enabled (§4.9).

Then we perform the system-level evaluation of *vTMM* and compare *vTMM*'s performance with that of *AutoNUMA* and Intel *MM* [25], as well as the *Nimble*. *AutoNUMA* indicates that NUMA balancing is enabled in VMs without other management. In particular, we configure *AutoNUMA* to allocate FMem preferentially by using libnuma [16, 20]. *MM* must configure all DRAM (32GB in our configuration) as the DRAM cache. Thus, to be fair, we compared *vTMM* with *MM* in scenarios where the same DRAM usage can be guaranteed. *Nimble* is a two-tiered memory management design for the THP system and we implement it in guest OS for comparison, though it is designed for non-virtualization.

4.3 Page Tracking Efficiency

4.3.1 Page table scanning efficiency. To evaluate the page table scanning efficiency of PML-based GPT scanner, we execute a `random access` workload on a VM with 64 GB of memory. The workload has 10 GB of resident set size (RSS, i.e., memory footprint). After warming-up, the VM establishes 64 GB memory mappings. Figure 2 [a] shows the result (GPT scanner vs. EPT scanner). The X-axis represents 90% access touching $X\%$ (i.e., hot page ratio) of pages and the Y-axis shows the amount of VM memory scanned.

The results show that the GPT scanner is affected by the locality of memory accesses. The better the locality is, the fewer GPTs are involved. The GPT scanner uses PML to accurately capture GPTs accessed recently, which greatly reduces the number of pages to be scanned. In contrast, because the EPT scanner is not aware which pages are accessed, it has to scan the whole EPT (with 64 GB of mappings) regardless of the hot page ratio. It will cause long time of page tracking when the VM has a huge mapping.

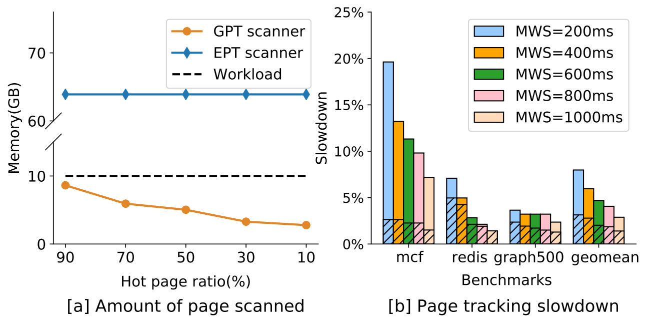

Figure 2. [a] Comparison of page scanned (dotted line represents the RSS of the workload); [b] Comparison of page tracking slowdown (solid line columns represent EPT scanner and shadow columns represent GPT scanner).

4.3.2 Page tracking overhead. We choose `429.mcf` (with 3.2 GB RSS) of SPEC CPU2006 [13], `graph500` (with 19 GB RSS) [15] and `redis` [35] (with 32 GB RSS) as the workloads to test the slowdown caused by page tracking. `Graph500` iterate 8 times with BFS and SSSP. Both `graph500` and `429.mcf` use execution time as performance metric. We test `redis` with YCSB [11] and each key-value size is 4 KB. *READ* and *UPDATE* account for 50% respectively and the workload follows the *hotspot* distribution, where 80% of access operations occur on 20% of data. `Redis` uses the throughput as performance metric. We choose the EPT scanner as a comparison. In this experiment, both the GPT scanner and the EPT scanner are enabled all the time.

Frequent page table scanning results in a significant address translation overhead because of TLB flush and page table A/D bits setting. As shown in Figure 2 [b], the EPT scanner causes up to a 20% slowdown for `429.mcf`, but the GPT scanner with multi-queue causes only a 2.6% slowdown.

We increase the monitoring window size (MWS) to 1000 ms (adopted by *RAMinate*), and the slowdown of `429.mcf` with the EPT scanner is still up to 8%. In contrast, the slowdown of the GPT scanner is no more than 2%. For applications, like `429.mcf`, with memory intensive, the GPT scanner achieves a lower slowdown by utilizing the multi-level queue to filter large amount of hot page tracking. `Redis` accesses memory sparsely with low frequency and the GPT scanner still has performance advantages than the EPT scanner. By geometric mean, our design can reduce the page tracking slowdown by more than 50% over the EPT scanner. In particular, when $MWS \geq 600ms$, the slowdown is less than 2%. Thus, we choose 600 ms as MWS for subsequent experiments.

In addition, we also test the additional VMTraps caused by using the PML. Each VCPU has a separate PML buffer with 512 entries. When the buffer is full, an additional VMTrap, *PML BUFFER FULL*, is triggered. We calculate the proportion of VMTraps generated due to *PML BUFFER FULL* event during the execution of `random access` workload with working set from 4 GB to 16 GB. The larger the working set of the workload, the more VMTraps are generated due to *PML BUFFER FULL*. The results show that no more than 0.34% of additional VMTraps are generated. By leveraging PML, we capture dirty GPT pages rather than all dirty pages. The hypervisor actively flushes the PML buffer every time when any type of VMTraps is triggered.

4.3.3 Multi-level queue.

The locality-based multi-level queue estimates the page access frequency. We choose multiple benchmarks to test whether the multi-level queue approach can keep the memory access pattern. Due to space limitations, we only present the result of `redis`. The RSS is 10 GB and `redis` has a fixed hot set (2 GB), which makes it easy to observe. We monitor 100 cycles and the maximum queue level is configured as 7.

Figure 3 shows distribution of sorted page access frequency and the monitor window size (MWS) changes from 200 ms to 1000 ms. In all the tests, the PML-based GPT scanner can capture the hot set (20%) accurately. The solid and dashed lines have the same shape, indicating that the memory access pattern is described correctly when enabling multi-level queue. We observe that the frequency of hot pages is higher in the distribution when enabling multi-level queue. Hot pages enter a higher-level queue and their A/D bits are not be cleared in the next 2^{i-1} monitor windows, so that they are projected to accessed 2^{i-1} times although the actual frequency might be less. Cold pages are not affected because they stay in the low level queues and prediction error is much smaller. The multi-level queue makes it more efficient to distinguish between hot and cold pages.

In addition, when the MWS is increased, the frequency increase of hot pages caused by multi-level queue becomes more significant. According to the throughput measured by YCSB, `redis` accesses memory pages with very low frequency. Thus, with a small MWS, a page cannot be captured in multiple continuous monitor windows, which makes the level upgrade difficult. Therefore, when MWS is $200ms$, there is no difference whether multi-level queue is enabled or not. Conversely, with bigger MWS, accesses to a same page are more likely to be captured in consecutive windows and the level upgrade will be faster. Combined with the results of §4.3.2, we can conclude that multi-level queue can effectively reduce the overhead caused by page table scanning while maintaining the application memory access pattern.

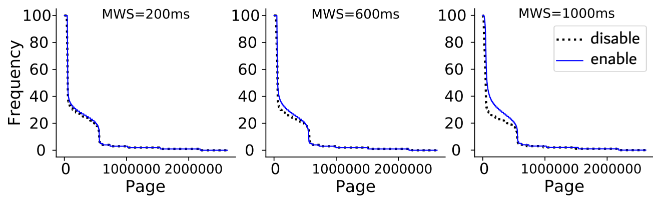

Figure 3. Access frequency of tracked pages with/without multi-level queue.

4.4 Page Migration Efficiency

In this section, we compare PML-based page migration (PM_PML) to raw Linux page migration (PM_Linux), and write protection-based page migration (PM_WP) adopted by *HeMem*. We should focus not only on migration speed but also on the slowdown caused by migration. We configure an VM with 20 GB DRAM and 20 GB NVM. We choose benchmarks with uniform random memory access, including `random read` and `random write`. We run a benchmark on the NVM first and then migrate all pages of the benchmark to the DRAM.

4.4.1 Page migration speed.

Obviously, the sooner we can finish the migration, the sooner hot pages can be accessed on high-speed DRAM. We vary the RSS of the workload from 2 GB to 16 GB and observe that the migration speed (throughput) is stable. The results show that benefiting from parallel migration, PM_PML and PM_WP have basically the same speed and both are twice as fast as PM_Linux. PM_PML and PM_WP migrate a write-workload is slower than a read-workload because of handling dirty pages.

4.4.2 Slowdown caused by page migration.

We use the execution time of the workload as a metric to measure the slowdown of the workloads caused by page migration. Table 4 and Table 5 show the workload slowdowns (compared to DRAM performance) after migration by PM_PML, PM_WP, PM_Linux. We present the slowdowns (compared to DRAM performance) of running on NVM without migration.

For `random read`, both PM_PML and PM_WP have better performance than PM_Linux. Both PM_PML and PM_WP migrate pages faster and do not need to unmap pages before copying page data, which reduces the pause time for VM page access during page migration. Also, when running `random read`, there are few dirty pages to handle. PM_PML causes lower slowdowns than PM_WP because PM_PML actively

Table 4. Slowdown caused by migration (random-read).

RSS (GB)	NVM	PM_Linux	PM_WP	PM_PML
2	62.8%	6.9%	0.7%	0.1%
4	60.6%	7.0%	2.8%	1.5%
8	72.1%	7.6%	3.1%	1.7%
16	78.0%	8.0%	3.3%	2.2%

fills mappings in the EPT, which reduces the cost of VMTraps to handle EPT page faults. The results of VMTraps caused by page migration are shown in Table 6. For `random read`, both PM_Linux and PM_WP generate VMTraps that are roughly equivalent to the number of migrated pages. PM_PML, in contrast, incurs few VMTraps.

Table 5. Slowdown caused by migration (random-write).

RSS (GB)	NVM	PM_Linux	PM_WP	PM_PML
2	188.2%	4.8%	0.7%	0.3%
4	190.6%	5.0%	3.6%	1.8%
8	189.8%	5.4%	5.7%	2.5%
16	183.5%	10.0%	8.4%	3.4%

For `random write`, both PM_Linux and PM_WP cause much a higher slowdown than PM_PML. For example, when migrating the 16 GB workload, PM_Linux and PML_WP cause 3.2x and 2.5x slowdowns over PM_PML, respectively. Compared to PM_Linux, PM_PML finishes migration faster due to parallel processing. As shown in Table 6, compared to migrating `random read`, PM_WP doubles the number of VMTraps when migrating write workloads of the same RSS. PM_WP triggers a large number of VMTraps because of write-protection exceptions during page migration. PM_PML, in contrast, eliminates this overhead by leveraging PML.

Table 6. VMTraps caused by page migration (thousand)

RSS	PM_Linux		PM_WP		PM_PML	
(GB)	read	write	read	write	read	write
2	523	524	526	1030	<1	<1
4	1039	1049	1051	1988	<1	<1
8	2097	2080	2102	3927	1	1
16	4188	4196	4203	7721	2	2

4.5 Ablation Study

In this section, we evaluate the performance improvements generated by the three main components (i.e., PML-based page tracking, bucket sort-based hot/cold classifier and PML-based page migration) of *vTMM*. We replace *vTMM* components with EPT scanner, a fixed threshold classifier, and PM_Linux for comparison. In particular, the fixed threshold classifier compares a page's access count with a fixed threshold to directly determine whether the page is a hot page. Following the method of *HeMem* [40], we select the value of fixed threshold by profiling a `random access` benchmark. In addition, we run benchmarks in a DRAM-only VM for comparison. We configure the VM with 8 GB DRAM and 32 GB NVM except the DRAM-only system.

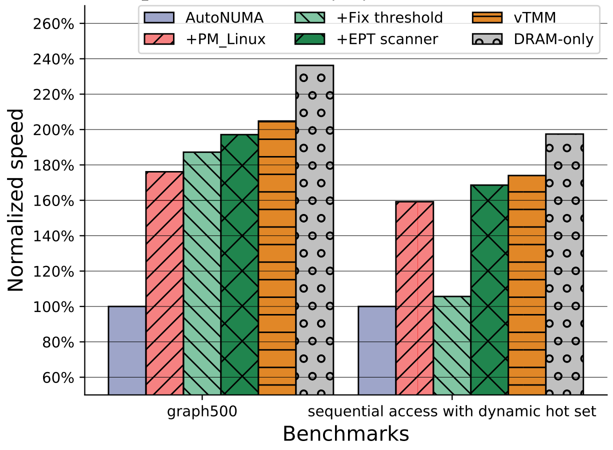

Figure 4. Benchmark execution performance (the performance of *AutoNUMA* is normalized as 1).

Figure 4 shows the relative performance speedup of these six configurations over *AutoNUMA*. We choose `graph500` (with 19 GB RSS) and `sequential access` (with 36 GB RSS) as workloads. The `sequential access` consists of 50% read and 50% write and its hot set size is fixed (7.2 GB), but the hot data distribution changes periodically. For `graph500` and `sequential access`, *vTMM* outperforms *+EPT scanner* by 3.9% and 3.2%, respectively. This indicates that the PML-based GPT scanner with multi-level queue has less page tracking overhead than the EPT scanner. For `graph500`, *vTMM* outperforms *+fixed threshold* by 9.4% and for `sequential access`, the performance speedup is about 65%. Distinguishing hot/cold pages based on sorting adopted by *vTMM* is more adaptable than the fixed threshold classifier. *vTMM* provides 16.3% (`graph500`) and 9.3% (`sequential access`) higher performance than *+PM_Linux*. With PML-based page migration, *vTMM* benefits from high page migration speed and efficient dirty page handling mechanism that minimizes the pause time for VM page access during page migration. Because of more intensive memory write access, for *vTMM*, `sequential access` costs more to handle dirty page than `graph500`. In addition, *vTMM* outperforms *AutoNUMA* 105% and 74% for `graph500` and `sequential access`, respectively. *vTMM* can provide more than 87% DRAM VM system performance with only 20% DRAM both in two tests.

4.6 Fixed/Dynamic Hot Set

vTMM aims to place the hot set into DRAM accurately. We verify *vTMM*'s adaptability to the VM hot set. We use a `random access` micro-benchmark and set 90% of operations to access hot objects while the remaining 10% of operations uniformly access the entire memory footprint. The experimental VM is configured with 8 GB DRAM and 16 GB NVM.

4.6.1 Fixed hot set.

First, we configure the micro-benchmarks with fixed hot sets of different size. As shown in Figure 5 [a], we vary the hot set ratio of the workload (with 16 GB RSS) from 10% to 80%. When the hot set can fit in the DRAM, *vTMM* can keep more than 93% performance compared to the DRAM-only system, which indicates that *vTMM* can identify the hot set and migrate hot pages into DRAM memory rapidly. The performance gap comes from approximately 10% of accesses must go to slow NVM. On the contrary, the performance of *AutoNUMA* suffers an average loss more than 25% than the DRAM system. When the hot set exceeds the DRAM capacity, the VM performance with *vTMM* and *AutoNUMA* converges. But *vTMM* still outperforms *AutoNUMA* by 28%. When there are no cold pages in DRAM to exchange hot pages in NVM, *vTMM* stops page migration.

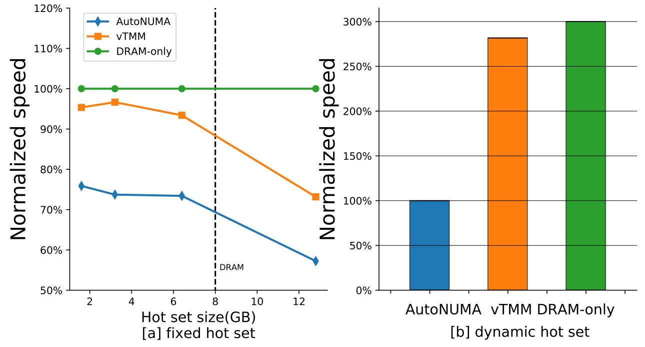

Figure 5. Micro-benchmark performance.

4.6.2 Dynamic hot set.

We configure the `random access` with a dynamic hot set. The benchmark has four stages and in each stage, the hot data is distributed in a different 4 GB virtual address space randomly. As shown in Figure 5 [b], *vTMM* achieves 94% performance of the DRAM-onlyDRAM system, but *AutoNUMA* suffers from a 2× slowdown over the DRAM-only system. *vTMM* can accurately track hot pages and migrate them into DRAM in time even when the hot set changes dynamically.

4.7 Sensitivity to DRAM Memory Size

Effective use of DRAM is the key to ensure the performance of heterogeneous memory system. In practice, VMs may be configured with different DRAM memory capacities for cost-performance ratio. In this section, we vary the VM DRAM size and show the adaptability of *vTMM*. We choose the AutoNUMA as a baseline. We select four workloads: `649.fotonik3d_s` of SPEC CPU 2017 [14], `graph500`, `redis` and `page rank` of GAPS [3]. As shown in Figure 6, different types of applications have different sensitivity to DRAM size.

As a parallel CPU and memory test benchmark, `fotonik3d_s` intensively accesses the entire working set partition. Both *vTMM* and *AutoNUMA* incur a high slowdown when the memory footprint (10 GB) exceeds the DRAM size. This indicates that `649.fotonik3d_s` has a very urgent and huge

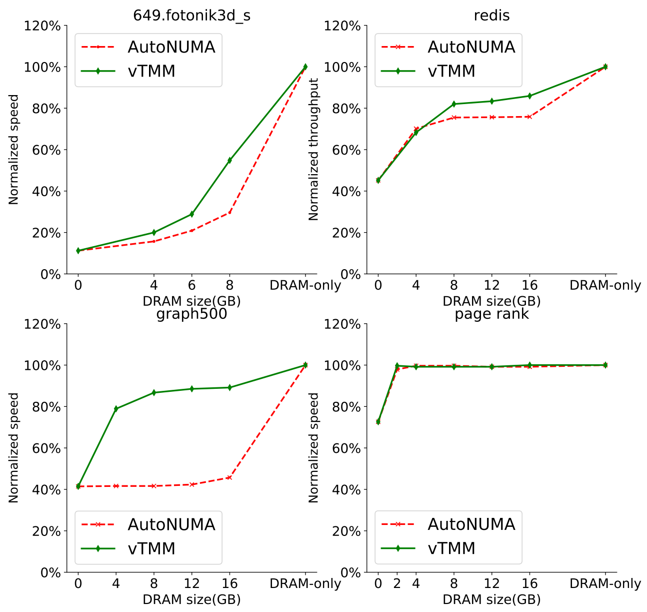

Figure 6. VM performance sensitivity to DRAM size.

demand for fast memory (DRAM). For `graph500`, when the DRAM size varies from 4 GB to 16 GB, *vTMM* reaches 79% to 90% performance of that of the DRAM-only VM. *vTMM* outperforms *AutoNUMA* 95.6% on average. This benefits from *vTMM*'s ability to determine hot set accurately and migrate hot pages from NVM to DRAM quickly. We configure `redis` with 32 GB RSS and 20% hot data. As an in-memory database with a large memory footprint, `redis` has a sparse memory access pattern. When the DRAM size is smaller than hot set size, both *vTMM* and AutoNUMA provide about 70% DRAM performance due to frequent NVM access. Once DRAM could fit hot page set, *vTMM* achieves better performance (more than 10%) than AutoNUMA. `Page rank` mainly computes the contribution of each node by sparsely walking the graph node arrays in sequence. Thus, it has poor locality. The memory footprint of `page rank` is configured as 19 GB. *vTMM* achieves 100% DRAM performance when $DRAM \geq 2GB$. By analyzing the access distribution of VM pages with *vTMM*, we observe that most memory accesses are located in the top of address space, a small amount of memory ($\leq 1GB$). *AutoNUMA* uses fast memory preferentially, so hot set fit perfectly in DRAM and it also achieves 100% of DRAM performance.

4.8 Transparent Huge Page (THP) Support

This section evaluates *vTMM* performance when the VM enables THP mechanism. We compare the performance of *vTMM* with that of *AutoNUMA*, *MM* and *Nimble*. *MM* must use 32GB of DRAM as DRAM cache. To be fair, for *vTMM*, *AutoNUMA* and *Nimble*, we configure VM with 32 GB DRAM. Benchmarks used include page rank algorithm (PR) and betweenness centrality (BC) algorithm of GAPS [3], `graph500` and redis. The parameters and memory footprints of benchmarks are shown in Table 7. We provide two sets of parameters for each benchmark. For example, we configure page rank with 2^{27} vertices (fits in DRAM) and 2^{29} vertices (exceeds DRAM), respectively, called PR_S and PR_L.

Table 7. Parameters and RSSes of benchmarks.

	PR[+]		BC[+]		graph500		redis	
Parameter.[*]	2^{27}	2^{29}	2^{27}	2^{29}	2^{25}	2^{27}	8M	25M
RSS (GB)	18	70	20	76	19	76	32	100

[*] For PR, BC, graph500, the parameter represents the number of vertices; For redis, it represents the number of K-V pairs.

[+] For PR and BC, we present the RSS during the stable iteration period, and building the graph requires doubling the RSS.

Figure 7 presents the performance comparison. When memory footprint can fit in DRAM, *vTMM* remains the workload's all memory in DRAM, achieving performance close to the DRAM-only system. *vTMM* outperforms *MM* by 14.7% on average. With a direct-mapped DRAM cache, MM can suffer conflict misses incurring costly NVM accesses and the frequent NVM write back is also expensive, especially for write-intensive programs like graph500 and BC. We configure *AutoNUMA* allocating DRAM memory preferentially by using libnuma. However, graph500_S tries to allocate memory from two NUMA nodes equally for CPU balancing. As a result, *AutoNUMA* provides only half the performance of the DRAM system. In addition, like *vTMM*, *Nimble* migrates all workload pages and achieve performance close to the DRAM-only system.

When the memory footprint exceeds the DRAM capacity, the workloads behave differently. For PR_L, *vTMM* and *Nimble* provide the same of performance with *AutoNUMA*. This result matches the analysis of sensitivity to DRAM size in §4.7. However, *MM* suffers from a 26% slowdown due to heavy conflict cache misses. For BC_L, *Nimble* outperforms *AutoNUMA* by 30% and *vTMM* achieves 48.4% higher performance than *Nimble*. *vTMM* achieves page tracking with low overhead, and benefiting migration feedback mechanism (§3.2.3), *vTMM* can suspend unnecessary page scans based on history. In contrast, we observe that for program with large working set and intensive memory access pattern, the performance of *Nimble* is limited due to the high frequency of page scanning without optimization. Compared with *MM*, *vTMM* provides 13% higher performance for BC_L. And graph500_L has basically the same results as BC_L.

Redis_L is configured with 20 GB hot set, which can fit in DRAM. *vTMM* outperforms *AutoNUMA* by more than 10%. *vTMM* identifies hot pages and migrate into DRAM. *MM* outperforms *vTMM* by about 10%. The hot keys of Redis distribute randomly in entire address space and Redis accesses memory dispersedly. Therefore, when memory footprint exceeds DRAM capacity, *MM*, a fine-grained (cache line) hardware management, is able to exploit more locality

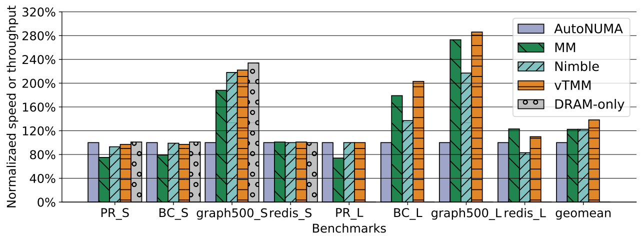

Figure 7. Benchmark performance when VM enables THP (the performance of *AutoNUMA* is normalized as 1).

than page-level management. In addition, *Nimble*'s performance is even 19% lower than the baseline (*AutoNUMA*). Nimble adopts active/inactive LRU lists of the Linux memory reclamation mechanism [44]. *Two page states ("active" and "inactive') are not sufficient to describe all possible access patterns* [5]. For example, although a page of DRAM is "inactive" for the most of time, one recent access makes it "active", thus denying migrating the page into NVM, even if it is not going to be accessed for a long time. Redis has this kind of pattern.

4.9 Multi-VM Co-Running

This section evaluates *vTMM* when multiple VMs co-run, including regular page and huge page. We configure *vTMM* as two modes: *vTMM* island mode and *vTMM* pool mode. In island mode, each VM has a fixed FMem capacity and *vTMM* independently manages pages for each VM. In pool mode, *vTMM* enables memory pooling (§3.5) and FMem is dynamic partitioned on demand. In particular, we present the results of both DRAM+NVM system and CXL-Memory system.

4.9.1 VMs with Regular Page. We configure four VMs and each VM has 40 GB memory including 8 GB FMem and 32 GB SMem. For pool mode, and we set 75% of the initial FMem size as the default (6 GB) to ensure basic VM performance. This is the lower bound of FMem size. In addition, the benchmarks are configured the same as §4.7.

We first analyze the results of DRAM+NVM system (Figure 8[a]). Using *AutoNUMA* as baseline, by geometric mean, *MM* provides a 15% performance improvement. However, *MM* suffers from a 30% slowdown for the VM4 (page rank), which matches the result of §4.8. Moreover, the performance degradation is exacerbated because multi-VM co-running causes heavier DRAM cache pollution. When multiple VMs co-run, *vTMM* still achieves distinguishing hot/cold pages and relocating pages effectively. And *vTMM* effectively avoids the interference of DRAM access of multiple VMs.

vTMM pool mode outperforms *MM* and *vTMM* island mode on every VM. We observe that with management of pool mode, VM2, VM3 and VM4 release 1.5 GB, 1.5 GB and 2 GB to the pool respectively, while VM1 gets 1.5 GB DRAM memory from the pool. Like the analysis in §4.7, 649.fotonik3d_s, with 10 GB RSS, makes intensive access to the most address space, which requires more DRAM memory. `Page rank` running in VM4 requires little DRAM, so the DRAM is down to the default value (6 GB). The hot set of `redis` is 6.4 GB, so it frees nearly 1.5 GB of DRAM. VM2 (`graph500`) also progressively releases 1.5 GB DRAM. Thus the VMs actually consume a total of 28.5 GB DRAM. In particular, the pool mode achieves *higher* performance but with *less* DRAM for VM2 and VM3 than the island mode. For VM2 and VM3, the remaining DRAM is sufficient to cover their hot sets. Without sufficient DRAM, under *vTMM* island mode, VM1 suffers frequent page migration, which affects the performance of other VMs by consuming memory bandwidth. In summary, with pooling management, *vTMM* achieves higher performance by judiciously allocating precious DRAM memory.

Overall, in DRAM+NVM system, *vTMM* island mode outperforms *AutoNUMA* and *MM* by 35% and 17% (geomean), respectively. The overall performance of *vTMM* pool mode is 12% higher than *vTMM* island mode. In the CXL-Memory system, *vTMM* still improves system performance by 15%(island mode) and 16% (pool mode), though remote DRAM performs much better than PMem.

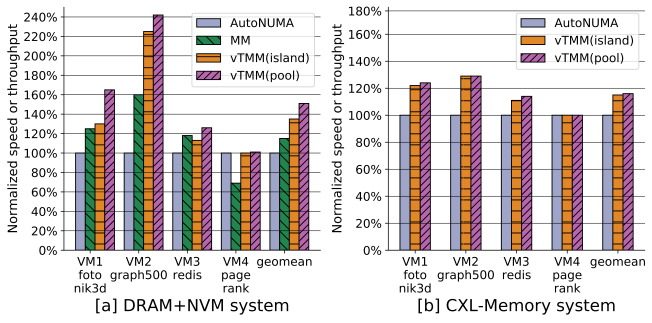

Figure 8. Performance of co-running VMs (regular page).

4.9.2 VMs with Huge Page.

We evaluate *vTMM* when VMs enable THP and the result is shown in Figure 9. Each VM has 16 GB FMem and 64 GB SMem. For pool mode, we also set 75% of the initial FMem size as the default (i.e., $12GB$). VM1 runs `redis` with 50 GB RSS (including 20% hot data) and VM2 executes `graph500` with 76 GB RSS.

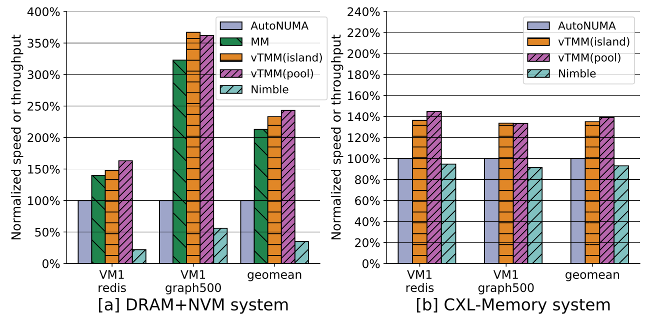

Figure 9. Performance of co-running VMs (huge page).

We also take the results of DRAM+NVM system as an example to analyze the performance of *vTMM*. When the VMs enable THP, *vTMM* still achieves higher performance than MM. We observe that when running `redis` in a single VM, MM performs better than *vTMM* island mode (see §4.8). But when two VMs co-run, *vTMM* (island) outperforms MM 5.3% on `redis`. Similarly, the performance gap between MM and *vTMM* (island) widens for `graph500`. We measure the DRAM cache load miss ratio for MM by detecting two hardware events[2]: *MEM_LOAD_RETIRED.LOCAL_PMM* (PMM_C) and *MEM_LOAD_L3_MISS_RETIRED.LOCAL_DRAM* (DRAM_C) [28]. The former counts retired load instructions with local PMem as the data source and the data request missed the DRAM cache in MM. The latter counts retired load instructions with data serviced from local DRAM. Thus, the DRAM cache load miss ratio is $PMM\_C/(PMM\_C + DRAM\_C)$. We compare miss ratio when the workload runs in single VM and double VMs. For `redis`, the miss ratio increases from 6.52% to 16.89% and for `graph500`, it increases from 3.88% to 14.87%. The result indicates that when multiple VMs co-run, *MM* suffers heavy (conflict) cache misses.

In particular, *Nimble* provides a lower performance on `graph500` compared to *AutoNUMA*, which is the opposite of single-VM result (§4.8). The DRAM size of the single-VM experiment is double that of the multi-VM experiment. The larger the DRAM size, the higher the wrong page placement tolerance. *Nimble*'s LRU-based policies cannot accurately identify hot set when DRAM size is relatively limited. However, *vTMM* achieves more than 5× performance over *Nimble*, which suggests that *vTMM* still works well under the condition of tight DRAM size, due to our page monitoring and bucket sort-based hot/cold page classifier.

Overall, in the DRAM+NVM system, *vTMM* outperforms *AutoNUMA*, *MM* and *Nimble* 1.4×, 14.1% and 5.9×, respectively. Similarly, *vTMM* still performs best on CXL-Memory system and outperforms *AutoNUMA* and *Nimble* 39.5% and 49.4%, respectively.

5 Conclusion

This paper proposes *vTMM*, a novel tiered memory management system for virtualization. *vTMM* holds hot pages in fast memory and cold pages in slow memory through page tracking, classification, and migration. In particular, *vTMM* optimizes performance by utilizing hardware-assisted virtualization. In addition, it adopts dynamic partitioning to balance fast memory between multiple VMs for higher utilization and performance. We have evaluated *vTMM* in a DRAM+NVM system that supports Intel Optane DC PMem and a NUMA-simulated CXL-memory system. The results show that *vTMM* outperforms *AutoNUMA*, *MM* and *Nimble* for multiple applications.

[2]Intel PMU does not provide similar events for store instruction

References

[1] 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.

[2] Dmytro Apalkov, Alexey Khvalkovskiy, Steven Watts, Vladimir Nikitin, Xueti Tang, Daniel Lottis, Kiseok Moon, Xiao Luo, Eugene Chen, Adrian Ong, et al. 2013. Spin-transfer Torque Magnetic Random Access Memory (STT-MRAM). *ACM Journal on Emerging Technologies in Computing Systems (JETC)* 9, 2 (2013), 1–35.

[3] S. Beamer, K. Asanovi, and D. Patterson. 2015. http://arxiv.org/abs/1508.03619. The GAP Benchmark Suite. *arXiv e-prints* (2015. http://arxiv.org/abs/1508.03619).

[4] Stella Bitchebe, Djob Mvondo, Laurent Réveillère, Noël De Palma, and Alain Tchana. 2021. Extending Intel PML for Hardware-assisted Working Set Size Estimation of VMs. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 111–124.

[5] Daniel P Bovet and Marco Cesati. 2005. *Understanding the Linux Kernel.* " O'Reilly Media, Inc.". (Section 17.3.1.1 of page 828).

[6] Apostolos Burnetas, Daniel Solow, and Rishi Agarwal. 1997. An Analysis and Implementation of An Efficient In-place Bucket Sort. *Acta Informatica* 34, 9 (1997), 687–700.

[7] Linux community. 2021. Intel PML Buffer Flush. https://elixir.bootlin.com/linux/v5.4/source/arch/x86/kvm/vmx/vmx.c.

[8] Linux community. 2022. migrate_pages() Function of Linux Kernel. https://elixir.bootlin.com/linux/v5.4/source/mm/migrate.c#L1399.

[9] Qemu community. 2021. QEMU. https://www.qemu.org/.

[10] CXL Consortium. 2022. CXL. https://www.computeexpresslink.org/.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. 2010. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing, SoCC 2010, Indianapolis, Indiana, USA, June 10-11, 2010.*

[12] Intel Corporation. 2022. Page Modification Logging for Virtual Machine Monitor White Paper. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/page-modification-logging-vmm-white-paper.pdf.

[13] Standard Performance Evaluation Corporation. 2022. SPEC CPU 2006 benchmarks. http://www.spec.org/cpu2006.

[14] Standard Performance Evaluation Corporation. 2022. SPEC CPU 2017 benchmarks. http://www.spec.org/cpu2017.

[15] Graph500 developers. 2022. Graph500. http://graph500.org/.

[16] NUMA developers. 2022. A NUMA API for Linux. http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf.

[17] G. Dhiman, R. Ayoub, and T. Rosing. 2009. PDRAM: A hybrid PRAM and DRAM main memory system. In *IEEE Design Automation Conference.*

[18] Mingkai Dong, Heng Bu, Jifei Yi, Benchao Dong, and Haibo Chen. 2019. Performance and Protection in the ZoFS User-space NVM File System. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles.* 478–493.

[19] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. 2014. BadgerTrap: a tool to instrument x86-64 TLB misses. *Acm Sigarch Computer Architecture News* 42, 2 (2014), 20–23.

[20] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *2014 {USENIX} Annual Technical Conference ({USENIX} {ATC} 14).* 231–242.

[21] Abel Gordon, Michael Hines, Dilma Da Silva, Muli Ben-Yehuda, Marcio Silva, and Gabriel Lizarraga. 2011. Ginkgo: Automated, application-driven memory overcommitment for cloud computing. *Proc. RESoLVE* (2011).

[22] Takahiro Hirofuchi and Ryousei Takano. 2016. RAMinate: Hypervisor-based Virtualization for Hybrid Main Memory Systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*. 112–125.

[23] Jingyuan Hu, Xiaokuang Bai, Sai Sha, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2018. HUB: Hugepage Ballooning in Kernel-based Virtual Machines. In *Proceedings of the International Symposium on Memory Systems*. 31–37.

[24] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Chen Ding, and Zhenlin Wang. 2016. Kinetic Modeling of Data Eviction in Cache. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. 351–364.

[25] Intel Inc. 2021. Intel Optane DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[26] Intel Inc. 2022. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3B. https://www.intel.cn/content/www/cn/zh/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3b-part-2-manual.html.

[27] Intel Inc. 2022. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3C. https://www.intel.cn/content/www/cn/zh/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3c-part-3-manual.html.

[28] Intel Inc. 2022. PerfMon Events. https://perfmon-events.intel.com/#.

[29] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *arXiv preprint arXiv:1903.05714* (2019).

[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.

[31] The kernel development community. 2022. DAMON. https://damonitor.github.io/doc/html/latest-damon/vm/damon/index.html.

[32] Jonghyeon Kim, Wonkyo Choe, and Jeongseob Ahn. 2021. Exploring the Design Space of Page Management for Multi-Tiered Memory Systems. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021*, Irina Calciu and Geoff Kuenning (Eds.). USENIX Association, 715–728. https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon

[33] Benjamin C Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory as A Scalable DRAM Alternative. In *Proceedings of the 36th annual international symposium on Computer architecture*. 2–13.

[34] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. 2003. Energy Management for Commercial Servers. *Computer* 36, 12 (2003), 39–48.

[35] Redis Ltd. 2022. Redis. https://redis.io/.

[36] Hasan Al Maruf, Hao Wang, Abhishek Dhanotia, Johannes Weiner, Niket Agarwal, Pallab Bhattacharya, Chris Petersen, Mosharaf Chowdhury, Shobhit O. Kanaujia, and Prakash Chauhan. 2022. TPP: Transparent Page Placement for CXL-Enabled Tiered Memory. *CoRR* abs/2206.02878 (2022).

[37] Timothy Merrifield and H Reza Taheri. 2016. Performance Implications of Extended Page Tables on Virtualized X86 Processors. In *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 25–35.

[38] Cheng Pan, Xiameng Hu, Lan Zhou, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2018. PACE: Penalty Aware Cache Modeling with Enhanced AET. In *Proceedings of the 9th Asia-Pacific Workshop on Systems*. 1–8.

[39] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. *Computer architecture news* (2009).

[40] 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.

[41] Sai Sha, Jingyuan Hu, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2020. Huge Page Friendly Virtualized Memory Management. *J. Comput. Sci. Technol.* 35, 2 (2020), 433–452. https://doi.org/10.1007/s11390-020-9693-0

[42] Sai Sha, Yi Zhang, Yingwei Luo, Xiaolin Wang, and Zhenlin Wang. 2021. Swift Shadow Paging (SSP): No Write-protection but Following TLB Flushing. In *Proceedings of the 17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 29–42.

[43] X. Wang, X. Liao, H. Liu, and H. Jin. 2018. Big Data Oriented Hybrid Memory Systems. *Big Data Research* (2018).

[44] 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.

[45] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. atrixKV: Reducing Write Stalls and Write Amplification in LSM-tree Based KV Stores with Matrix Container in NVM. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 17–31.