SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

September 2021, Vol. 64 192104:1–192104:16
https://doi.org/10.1007/s11432-019-2729-5

HMvisor: dynamic hybrid memory management for virtual machines

Dang YANG, Haikun LIU*, Hai JIN & Yu ZHANG

National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab, Cluster and Grid Computing Lab,
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074*, China*

Received 10 March 2019/Revised 13 May 2019/Accepted 12 November 2019/Published online 19 August 2021

Abstract Emerging non-volatile memory (NVM) technologies promise high density, low cost and dynamic random access memory (DRAM)-like performance, at the expense of limited write endurance and high write energy consumption. It is more practical to use NVM combining with the traditional DRAM. However, the hybrid memory management such as page migration becomes more challenging in a virtualization environment because virtual machines (VMs) are unaware of the memory heterogeneity. In this paper, we propose HMvisor, a hypervisor and VM coordinated hybrid memory management mechanism to better utilize DRAM and NVM resources. HMvisor exposes the memory heterogeneity to VMs by mapping virtual NUMA nodes to different physical NUMA nodes. We propose a lightweight and efficient page migration mechanism by decoupling page hotness tracking from page migration. HMvisor performs those operations in the hypervisor and VMs separately, without disrupting the execution of VMs. We also propose a memory resource trading policy to adjust the capacity of DRAM and NVM for each VM, with the monetary cost unchanged. We implement our prototype system based on QEMU/KVM and evaluate it with several benchmarks. Experimental results show that HMvisor can reduce 50% of write traffic to NVM with less than 5% performance overhead. Moreover, the hybrid memory adjustment scheme in HMvisor can significantly improve application performance by up to 30×.

Keywords hypervisor, virtual machine, non-volatile memory, hybrid memory management

Citation Yang D, Liu H K, Jin H, et al. HMvisor: dynamic hybrid memory management for virtual machines. Sci China Inf Sci, 2021, 64(9): 192104, https://doi.org/10.1007/s11432-019-2729-5

1 Introduction

With the continuously increasing memory footprint of data center applications, dynamic random access memory (DRAM) scaling is unable to meet the heavy demand of memory capacity due to scalability issues in terms of density and power consumption [1]. Emerging byte-addressable non-volatile memory (NVM) [2] technologies, such as phase change memory (PCM) [3,4], 3D XPoint [5], promise high memory density, low cost per bit, and near-zero standby power consumption, at the expense of low performance and limited write endurance. Particularly, the typical PCM technologies offer approximate an order of magnitude higher capacity and lower cost per bit than DRAM, but show 2× higher read latency, up to 8× higher write latency, and about 5×–10× lower bandwidth [3,6,7]. The advantages of NVM make it to be complementary to DRAM, and possibly replace existing DRAM technologies in the future. Currently, it is more practical to use NVM in conjunction with DRAM in hybrid memory systems [8,9]. However, the challenging problem is how to fully exploit the advantages of NVM and DRAM and overcome their drawbacks.

There have been many studies on improving performance and energy efficiency of hybrid memory systems in non-virtualization environments. Owing to the performance gap between DRAM and NVM, many proposals use DRAM as an inclusive/exclusive cache to NVM, and place hot data in DRAM and leave cold data in NVM. This simple policy is often achieved through static data placement approaches [6,

* Corresponding author (email: hkliu@hust.edu.cn)

10–13] and dynamic memory migration approaches [7, 14–18]. The former approaches usually use offline profiling technologies to analyze applications' memory access patterns, and then direct programmers or compilers for static memory allocations. This kind of application-level memory allocation requires that applications are aware of the heterogeneity of hybrid memories. However, in a virtualization environment, the hardware abstraction layer (i.e., hypervisor) hides the heterogeneity of underlying hardware resources. This fundamental conflict makes the static hybrid memory allocation hard to implement in virtualization environments.

Dynamic memory migration approaches usually rely on operating systems (OSes) or memory controllers (hardware) to track page accesses at runtime, and swap hot pages between NVM and DRAM periodically [7, 14–18]. However, most current computer architectures assume that the main memory is homogeneous. They do not yet provide sufficient hardware support for fine-grained memory access monitoring owing to the hardware complexity and scalability. On the other hand, memory references are not necessarily perceived by OSes because the translation lookaside buffer (TLB) handles a large portion of virtual-to-physical address translations. A few OS-level page migration policies [19, 20] track page accesses by invalidating TLBs, resulting in unacceptable performance overhead at the software layer. The cost of page access counting even exceeds the benefit of page migration in hybrid memory systems. Interestingly, memory virtualization technologies such as shadow paging and extended page tables (EPT) provide a new opportunity to track page accesses in the hypervisor layer, resulting in much less software overhead of page access counting compared with the non-virtualization environments.

For hybrid memory management in a virtualization environment, there still remains an open problem whether the hypervisor should expose the memory heterogeneity to the guest OSes (i.e., VMs). A few studies [21, 22] proposed to migrate guest OSes' memory in the hypervisor layer, hiding the memory heterogeneity to the guest OSes. Those approaches mainly rely on the hypervisor to track the hotness of pages in VMs, and must interrupt the execution of VMs to guarantee data consistency during the page migrations. The other studies [10, 13] exposed the memory heterogeneity to VMs, and exploited application semantics to optimize data placement in the guest OSes. Although these approaches require to re-design the memory management subsystem in guest OSes to adapt to hybrid memory systems, application-level semantics can exploited to better utilize the hybrid memories.

In this paper, we propose a hypervisor and VM coordinated hybrid memory management system (called HMvisor) to optimize data placement in VMs. HMvisor leverages the non-uniform memory access (NUMA) abstraction in the hypervisor to map the pseudo physical memory space of a VM to both DRAM and NVM. This provides the guest OS an opportunity to perform intra-VM page migrations. HMvisor performs page access counting and identifies the top hot pages in the hypervisor. We develop a lightweight loadable driver in the guest OS to communicate with the hypervisor, and to collect the guest physical page numbers that should be migrated to fast DRAM. Then, the driver in the guest OS performs process-level page migrations between DRAM and NVM. Our approach does not have to disrupt the VMs, and HMvisor is transparent to the applications running in the VMs. We have made the following novel designs to support lightweight hybrid memory management in virtualization environments.

(1) We provide an NUMA abstraction mechanism in the hypervisor to expose memory heterogeneity to VMs. This is achieved by dynamic mapping a portion of DRAM and NVM to the guest physical memory space of VMs.

(2) We also develop a lightweight and efficient page hotness tracking mechanism in the hypervisor, which coordinates the guest OS to perform intra-VM page migrations. In contrast to the previous approaches that need to disrupt the guest OS during page migrations, we decouple page hotness tracking from page migration, and thus do not suffer the performance penalty of VM pausing during page migrations.

(3) Most public cloud providers offer many fix-sized VMs to users. Once a VM is chosen, its memory resource is not allowed to scale at runtime owing to the static resource tenancy policy in clouds. We demonstrate that trading a portion of DRAM for a larger size of NVM (vice versa) in a VM can significantly improve the performance of memory-hungry or latency-sensitive applications, while the total monetary cost of hybrid memories in a VM is kept unchanged. Thus, we propose a DRAM-NVM resource trading mechanism to adjust the hybrid memory allocation, so that the dynamic memory demands of workloads can be satisfied to some extent.

We implement our prototype HMvisor based on a widely-used hypervisor QEMU/KVM [23]. We evaluate HMvisor with several benchmarks. Experimental results show that HMvisor can significantly reduce 50% of write traffic to NVM with less than 5% performance overhead. Moreover, the hybrid memory adjustment in HMvisor can effectively improve application performance by up to 30×.

The remainder of this paper is organized as follows. We first introduce the background and motivation of this work in Section 2. We present the design and implementation of HMvisor in Section 3. We evaluate HMvisor in Section 4. We discuss the related work in Section 5 and conclude this paper in Section 6.

2 Background and motivation

In this section, we introduce the hybrid memory architectures and management mechanisms. We also discuss the feasibility and necessity of dynamic memory adjustment for VMs in public cloud environments.

2.1 Hybrid memory architectures

NVM technologies such as 3D XPoint [5] have illustrated the potential of significantly improving the main memory capacity with lower cost compared to DRAM. However, owing to the disadvantages of limited endurance and relatively low performance, a number of studies have proposed to organize DRAM/NVM in a cache/memory hierarchy [3, 7, 16]. These approaches need to design special hardware modules to fetch data from NVM to the DRAM cache. In contrast, many studies also advocate flat-addressable hybrid memory architectures [6,13,22,24,25] in which NVM and DRAM are organized in a single address space. NVM is attached to memory bus using the standard dual in-line memory module (DIMM) interface. In practice, it is feasible to organize DRAM/NVM in an NUMA architecture. In a virtualization environment, this approach is more convenient to expose the memory heterogeneity to VMs through an NUMA abstraction in the hypervisor layer.

2.2 Data placement in hybrid memory systems

Data placement mechanisms are crucial to improve the performance and energy efficiency of hybrid memory systems. These mechanisms generally can be categorized into static memory allocation and dynamic memory migration. The prior approaches try to optimize data placement in hybrid memories at the stage of memory allocation, and they need to excessively modify the existing memory management system in OSes and application programs. The latter approaches aim to migrate frequently-accessed pages to fast DRAM based on page hotness monitoring mechanisms. As page migrations are transparent to applications, these approaches do not have to change the source code of applications. This is particularly useful for legacy applications to run in a cloud environment with hybrid memory supported.

One challenging problem is how to efficiently track page accesses through a software approach. Previous studies such as clock-based [26] or LRU-based [27] algorithms can only monitor the most recently accessed data, and are not able to track page access frequency. The software-based page access counting mechanisms often lead to intolerable performance overhead [19, 20]. The advance of hardware virtualization technologies, such as Intel's EPT and AMD's NPT (Nested Page Tables), provides an opportunity to monitor page accesses in the hypervisor layer with much less software overhead. However, page migrations performed in the hypervisor [21, 22] should disrupt VMs to guarantee data consistency during data movement. In order to eliminate the VM downtime, we decouple page hotness tracking from page migration, and perform these operations in the hypervisor and guest OSes, respectively. In this way, VMs are not necessarily disrupted during page migrations. Guest OSes only need to forbid the writing operations on the on-the-fly pages by adding locks. This decoupled design can significantly reduce the performance overhead owing to page migrations in the virtualization environment.

2.3 Dynamic memory resource adjustment

Cloud applications are diversifying, and workloads may be dynamically changed over time. Thus, memory resource allocation in VMs should be adjusted to meet dynamic requirements of workloads [28, 29]. Memory ballooning [30–32] is a memory dynamic allocation technique for VMs supported by most virtualization platforms. It provides an effective way to balance memory utilization among VMs, and thus can improve the overall performance of multiple VMs. With the ballooning mechanism, a hypervisor can easily reconfigure the memory size of VMs by inflating or deflating the balloon driver within guest OSes. When the NVM resource is adopted in the cloud, several problems arise. The ballooning mechanism is designed for homogeneous memory systems, however, the cost and performance of DRAM and NVM are different, this may lead to performance interference in the cloud. For memory-hungry workloads, a fixed size of memory capacity may lead to a larger number of major page faults owing to page swapping

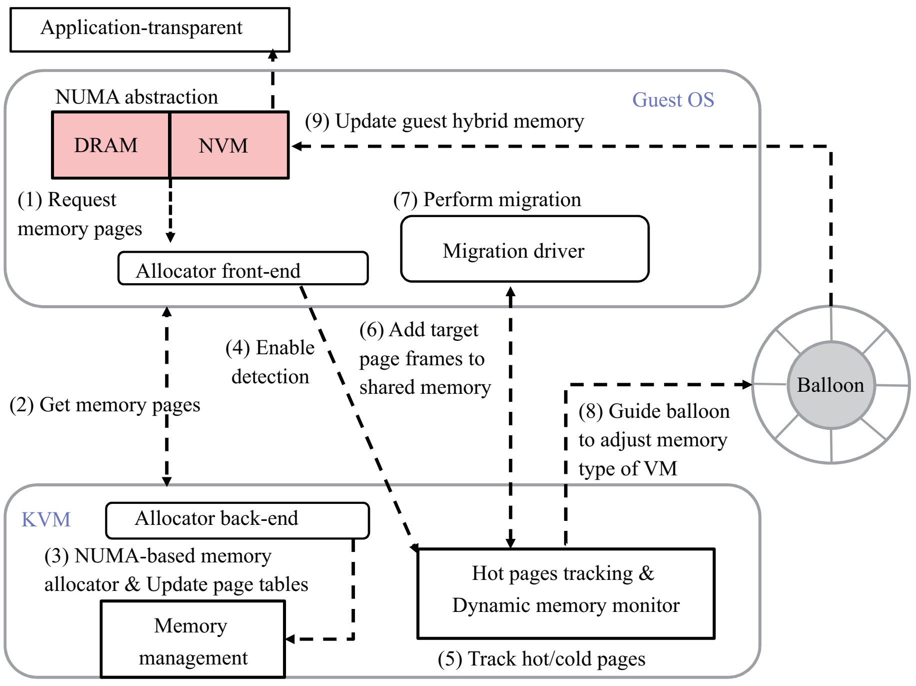

Figure 1 (Color online) System overview of HMvisor.

between memory and Disk, significantly degrading the performance of VMs. Thus, a larger size of memory capacity can significantly improve VM performance. For latency-sensitive workloads, the memory access latency rather than memory capacity has a significant impact on the application performance. It is advisable to use more DRAM instead of NVM. Therefore, it is important to let the balloon driver know the memory heterogeneity when adjusting the memory capacity of VMs.

On the other hand, current public clouds such as Amazon EC2 only support automatic resource scaling in a scale-out model. This implies the memory resource of VMs cannot be scaled up owing to the fixed monetary cost of VMs, although the ballooning mechanism is applicable in practice. We think this constraint can be relaxed in a hybrid memory system. For example, we can trade 1 GB DRAM for 4 GB NVM when the VM is running a memory-hungry application, while the total cost of memory is kept unchanged. This motivates us to develop a memory heterogeneity aware balloon driver to adjust VMs' memory allocation based on the resource trading mechanism [28, 29]. In this manner, we can meet dynamic memory requirements of workloads at runtime, and thus improve the performance of VMs.

3 Design and implementation

In this paper, we explore hybrid memory management mechanisms for virtualization environments. Figure 1 shows an overview of hybrid memory management in HMvisor. Unlike previous approaches, our design exposes the memory heterogeneity to VMs by mapping virtual NUMA nodes to different physical NUMA nodes, which are equipped with different types of memory. We employ an NVM-aware ballooning mechanism to support dynamical adjustment of hybrid memory mappings for VMs on the fly. Thus, the balloon driver can customize VMs' hybrid memory allocations to satisfy dynamic memory demands of different workloads. In order to mitigate write traffic to NVM, we also design a hypervisor and virtual machine coordinated page migration mechanism to dynamically relocate DRAM/NVM pages within VMs.

3.1 Support hybrid memory in VMs

At first, hypervisor should be aware of the memory heterogeneity and allocate different types of memory to a single VM. We exploit an NUMA abstraction in the hypervisor to dynamically map guest physical memory space to machine physical memory space, so that a VM can have both DRAM and NVM as its main memory. We implement HMvisor in a very popular virtualization platform, i.e., QEMU/KVM.

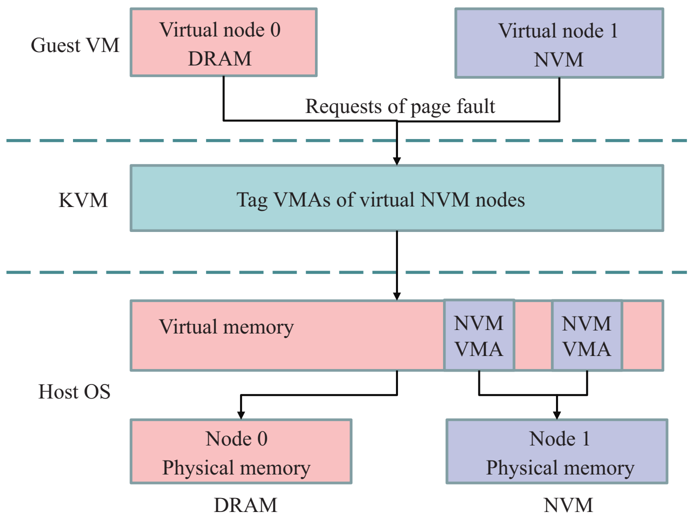

Figure 2 (Color online) Virtual-to-physical NUMA node mapping.

KVM handles page faults of VMs and delivers memory requests to the memory subsystem of Linux kernel. Thus, we provide an NUMA feature for VMs by using the Linux Fake NUMA patch [33]. This patch provides NUMA abstraction for VMs and does not modify the memory allocation policy of guest OSes. In practice, we use different physical NUMA nodes to emulate the underlying hybrid memory system. To make VM be aware of the memory heterogeneity, we use a simple yet effective way to map virtual NUMA nodes in a VM to different physical NUMA nodes.

We modify QEMU/KVM to support hybrid memories. QEMU sets up each VM with two virtual NUMA nodes. Figure 2 illustrates the mapping of virtual NUMA nodes to physical NUMA nodes. The virtual NUMA topology in a VM is initialized by KVM hypervisor when a VM is created. KVM hypervisor captures memory requests (minor page faults) from VMs at runtime. According to the mapping between virtual NUMA nodes and physical NUMA nodes, KVM can distinguish the page faults from different virtual NUMA nodes, and returns the corresponding machine memory address to the guest OS. This mechanism also exposes the memory heterogeneity to guest OSes.

With the virtual-to-physical NUMA mapping mechanism, guest OSes are able to optimize memory allocation using traditional NUMA memory management policies and application-level hybrid memory programming models [11]. Also, it provides an opportunity to swap memory pages belonging to different virtual NUMA nodes inside guest-OSes.

3.2 Hypervisor-VM coordinated page migration

We propose a hypervisor and virtual machine coordinated page migration scheme to optimize the performance of hybrid memory systems. It consists of a front-end driver for memory hotness tracking in the hypervisor, and another back-end driver for process-level page migration in the VMs. The hypervisor communicates with a VM through a shared memory region. The former driver delivers frequently-accessed page frame numbers to the shared memory region, and then the driver in VMs migrates the pages denoted by the shared memory region. Unlike traditional hypervisor-based page migration mechanisms, our approach does not cause data consistency issues during page migration in VMs. Moreover, our approach can also maintain the configuration of NVM/DRAM allocation for VMs, while the hypervisor-based page migration mechanisms implicitly change the ratio of NVM size to DRAM size in VMs. That is in conflict with the public clouds in which the VM' resource configuration is fixed.

3.2.1 *Page access counting*

A primary challenge of memory migration is to detect frequently-accessed pages in the NVM effectively and efficiently. A simple approach is to peer the access/dirty bits of page table entries periodically in the OS layer. However, this approach lacks sampling accuracy because a page may be accessed many times in a short interval. Some prior studies instrument x86-64 TLB misses to detect hot pages in the OS level, such as BadgerTrap [20]. Another approach is binary instrumentation [34], which traps

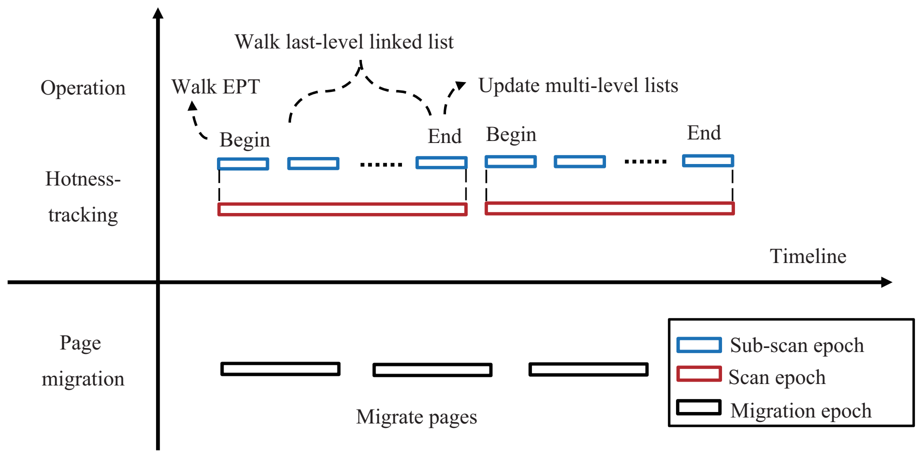

Figure 3 (Color online) Hotness tracking.

CPU load/store instructions to track memory accesses. Although these methods are able to provide more accurate page access counts than sampling the access/dirty bits of page table entries (PTEs), they usually incur extremely high software/hardware overhead.

Fortunately, modern processors provide some hardware support such as Intel's EPT for accelerating the address translation between guest frame numbers (GFNs) and machine frame numbers (MFNs). It can be exploited to track memory accesses of VMs in the hypervisor layer, with trivial additional performance overhead. From the processors of Haswell generation, each EPT entry contains access and dirty bits. Once a processor accesses the guest physical page, the access bit of associated EPT entry is set. Similarly, the dirty bit is set when a guest physical page is modified. Our goal is to place hot pages in DRAM and cold pages in NVM. To determine whether a page is frequently accessed, we should take both the recency and frequency [35] of memory accesses into account. In order to avoid unnecessary page migrations, we should accurately monitor the page accesses and carefully set the page hotness threshold.

We record memory read/write operations using 8-bit page access counters, which are used to identify hot pages of NVM and cold pages of DRAM. Specifically, we periodically scan the access/dirty bits, and the corresponding NVM read/write operations have different weights on page access counts. For example, if the weight of read is 1, the weight of write becomes the ratio of NVM write latency to NVM read latency. If the access/dirty bits are set, we increase the page access counts and then reset the bits. We create an EPT object to record the access information of GFNs. The EPT object has four items: (1) the guest frame number, (2) the page access counts in the previous monitoring epoch, (3) the time of the most recent update, and (4) the level of the multi-level linked list (see below). For memory allocation of EPT objects, we use the slab-based memory allocation to reduce memory fragmentation and the initialization overhead of small objects.

We propose multi-level linked lists to manage the EPT objects. Figure 3 shows the diagram of hot/cold page identification. Intuitively, all EPT entries should be traversed to sort the hot/cold pages according to their access counts. However, the traversal of the whole EPTs is extremely costly if the memory footprint of workloads is very large. Thus, we accelerate page hotness tracking by organizing the last-level EPTs in a linked list, and the traversal of this linear list is much faster than that of the tree-like EPTs. The linked list only stores the pointers which point to the last-level EPTs. At the beginning of each scan epoch, we traverse the whole EPT and maintain the last-level EPTs in a linked list. We scan the linked list to update 8-bit page access counters in each sub-scan epoch, which only takes several milliseconds. At the end of scan epoch, we create EPT objects to record guest page frames whose access counts exceed the hotness threshold and update the multi-level linked lists, which may take tens to hundreds of milliseconds. In our implementation, we empirically set the sub-scan epoch to 100 ms, and a full scan epoch covers 50 sub-scan epochs. In this way, we reduce the overhead owing to frequently sorting the page access counts and updating multi-level linked lists. In the page migration epoch, we select the hot pages from the multi-level linked lists and place the page frame numbers in a shared memory region. At the same time, we remove the corresponding EPT objects from the multi-level linked lists, and the data consistency of the multi-level linked lists is protected by spin-locks.

Similar to the MQ algorithm, we use multi-level linked lists to record the hotness of pages. As shown

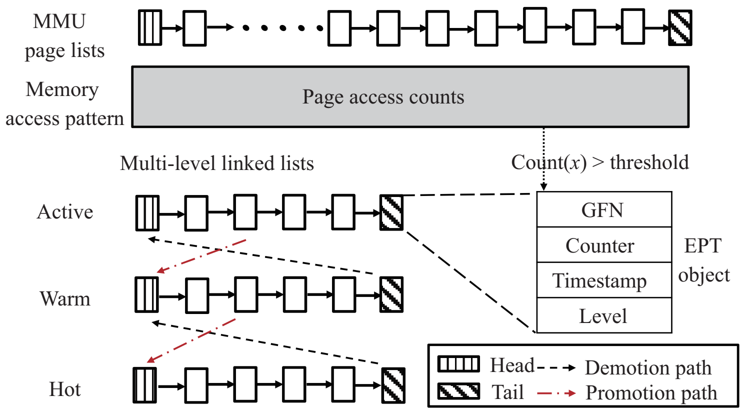

Figure 4 (Color online) Management of EPT objects.

in Figure 4, we use three levels of linked lists (i.e., active, warm, hot) to manage the candidates of hot pages. The page promotion and demotion in the three lists are illustrated as follows. In the beginning, the EPT object whose access counts exceed the hotness threshold is added to the head of the active list. If the interval of two successive accesses of a page is lower than a given threshold (called decay_time), we promote it to the head of the warm list. If this trend continues, the EPT object is promoted to the head of the hot list. In contrast, if the interval of two successive accesses of the EPT object exceeds the decay_time, we demote it to the head of the lower level linked list. At the end of scan epoch, we scan the multi-level linked lists from the tail to the head to perform the demotion operation. Once the elapsed time of last update of EPT object exceeds the decay_time, we move it to the head of the lower level linked list. For page migration, we walk the highest level linked list to migrate at most MAX_GFNS (currently 1024) pages in each migration epoch (currently 1 s). For cold pages in DRAM, we manage them in the same manner. As the cold pages in NVM and hot pages in DRAM are not managed in the linked list, the promotion and demotion operations are performed efficiently. In this way, our page hotness tracking mechanism takes the recency and frequency of page accesses into consideration to reduce the performance and storage overhead.

We can tune the parameters of page hotness tracking mechanism to improve the effectiveness and efficiency of page migrations. For example, the scan epoch determines the accuracy of hotness tracking, and the migration epoch determines the memory traffic owing to page migrations. The corked multi-queue (CMQ) algorithm in RAMinate [22] sets the scan epoch to 1 s and mainly adjusts the migration epoch to reduce the overhead of page migrations. We improve the accuracy of hotness tracking by setting the scan epoch to 100 ms. As the frequently-written pages are migrated to DRAM, our scheme also significantly optimizes the write endurance of NVM. Also, the hotness threshold can have a significant impact on the memory space consumed by the multi-level linked lists of EPT objects and memory traffic to the DRAM. We carefully configure the hotness threshold by extensive experiments.

Our page hotness tracking mechanism is different from CMQ [22] in two folds. (1) High-precision page access counting. In RAMinate, it takes about 800 ms to scan the page table when the application memory footprint is 1.6 GB. Therefore, the scan interval cannot be shorter than the time cost of page table traversal. To solve this problem, we maintain all last-level EPTs in a linked list, and significantly reduce the time for traversing the linked list to only several milliseconds. As a result, the scan interval in CMQ is set to 1 s, while it can be as short as 0.1 s in HMvisor. (2) Our approach can identify the recently-accessed pages more quickly than RAMinate. We promote the memory pages whose access counts exceed the hotness threshold in a linear mode rather than the logarithmic mode of CMQ. Therefore, our hotness tracking algorithm allows HMvisor to migrate recently and frequently accessed pages to DRAM node more quickly.

3.2.2 *Page migration*

As mentioned above, page migration in HMvisor is performed cooperatively by a front-end driver in the hypervisor and a back-end driver in the VM. The front-end driver delivers target page frames to the back-end driver through a shared memory region, which is implemented in a producer-consumer model. We develop a system call named heter_migrate_pages to migrate pages among NUMA nodes. HMvisor

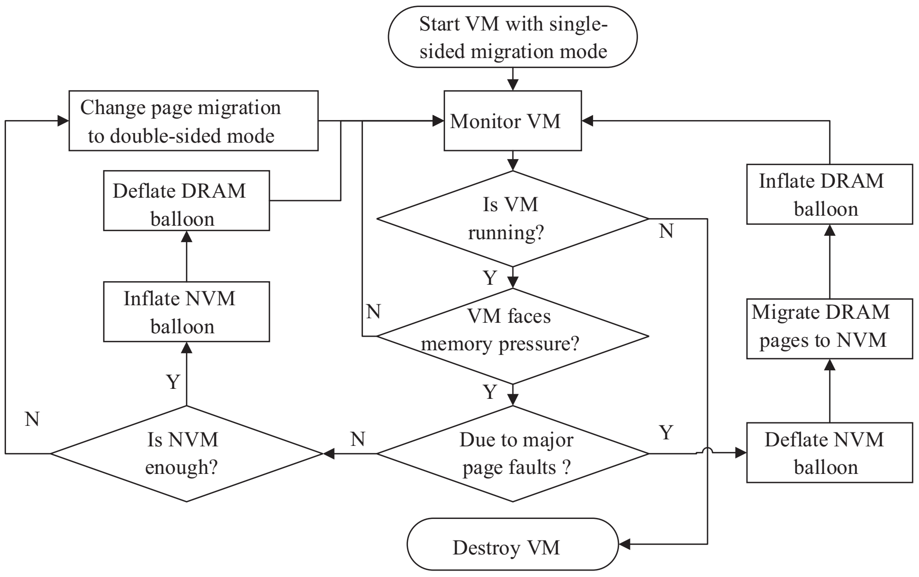
Start VM with single-sided migration mode
Monitor VM
Is VM running?
N
Y
VM faces memory pressure?
N
Y
Due to major page faults ?
Y
N
Is NVM enough?
Y
N
Inflate NVM balloon
Deflate DRAM balloon
Change page migration to double-sided mode
Deflate NVM balloon
Migrate DRAM pages to NVM
Inflate DRAM balloon
Destroy VM

Figure 5 Flow chart of dynamic memory adjustment.

performs memory hotness-tracking in the hypervisor and migrates only hot pages inside VMs, without disrupting the VMs.

We propose two strategies for page migration. One is single-sided migration and another is double-sided migration. The single-sided migration means that all hot pages in NVM and cold pages in DRAM can be migrated without constraints. Once a kind of memory is used up, the NVM-aware ballooning mechanism can adjust the DRAM-to-NVM ratio of hybrid memory systems. The double-sided migration means that the hot pages in NVM and cold pages in DRAM are swapped equivalently in each epoch. We combine the single-sided migration with an NVM-aware ballooning mechanism to perform dynamic memory adjustment among multiple VMs. When the NVM-aware ballooning mechanism does not work owing to severe memory pressure, the performance optimization can only rely on the double-sided migration for a single VM. These two migration strategies can be applied to different scenarios with dynamic memory demands.

The hotness tracking mechanism has a significant impact on the effectiveness of page migration. We propose a utility function $\mathrm{DRAM}_{\mathrm{utility}}$ to evaluate the efficiency of page hotness tracking mechanism. Assume the total amount of write traffic to DRAM and NVM are W_d and W_n, respectively. Let the usage of DRAM and NVM are M_d and M_n respectively. The utility function is defined as

$$\mathrm{DRAM}_{\mathrm{utility}} = \frac{W_d}{(W_d + W_n)} \Bigg/ \frac{M_d}{(M_d + M_n)}.$$

This function denotes the proportion of write traffic on a single unit of DRAM (e.g, 1 MB), and reflects the efficiency of DRAM usage. It can evaluate whether the hotness tracking mechanism can effectively migrate the most hot pages from NVM to DRAM. According to the definition of $\mathrm{DRAM}_{\mathrm{utility}}$, the value of this function tends to be 1 for the NUMA-Interleave policy. Therefore, we use the NUMA-Interleave policy as a baseline to analyze the efficiency of different hotness-tracking mechanisms. The difference of $\mathrm{DRAM}_{\mathrm{utility}}$ between a hotness-tracking mechanism and the NUMA-Interleave policy reflects the effectiveness of hotness-tracking mechanisms, and larger is better.

3.3 Dynamic memory adjustment

In order to meet dynamic memory requirements, we propose a dynamic memory adjustment mechanism to improve performance and energy efficiency of VMs in hybrid memory systems. Figure 5 shows the flow chart of our dynamic memory adjustment mechanism. VMs usually are configured with fix-sized hybrid memories and the default single-sided migration mode is used for page migrations. When a VM faces memory pressure owing to lack of main memory, we should trade a portion of DRAM for a larger size of NVM because the cost of NVM is expected to be much lower than DRAM. For example, if the price of

DRAM-to-NVM ratio is 4, and a VM has 2 GB DRAM and 4 GB NVM while the total memory demand increases to 9 GB, we can trade 1 GB DRAM for 4 GB NVM, and thus increase the VM's total memory capacity while keeping the VM's monetary cost unchanged. The resource trading between DRAM and NVM is achieved through the ballooning mechanism. When the memory resource in the host machine is insufficient, HMvisor uses the double-sided page migration mode to keep the hybrid memory allocation in VMs unchanged. We describe the details of dynamic memory adjustment for different workloads in the following. For memory-hungry workloads, if the VM faces high memory pressure owing to lack of main memory, we increase the total capacity of main memory by trading some DRAM for more NVM. First, we deflate the NVM balloon to increase the capacity of NVM. Second, a number of pages in DRAM are migrated to NVM. Finally, we inflate the DRAM balloon to return the portion of DRAM to the hypervisor.

To avoid the potential memory waste owing to over adjustment, we develop a simple yet efficient policy based on the denotation of major page faults, which reflect the intensity of page swapping between main memory and disk. The target balloon size is estimated by base$\times$(the number of page faults)$\times$(page size), where the initial value of base is 1. We periodically monitor the major page faults to calculate the miss rate in 5 s. While the miss rate continues to increase, we exponentially increase the value of base to provide more memory for the VM rapidly. When the miss rate declines, we decrease the value of base linearly. For latency-sensitive workloads, memory access latency rather than memory capacity has a significant impact on the application performance. It is advisable to use more DRAM instead of NVM. Hence, we trade some NVM for more DRAM to place more hot pages in the fast DRAM. When the memory demand is stable, HMvisor swifts to the double-sided page migration mode. In this way, HMvisor enables dynamic memory adjustment under a fixed monetary cost constraint.

4 Evaluation

4.1 Methodology

As the NVM device is still not commercially available, we use an emulation approach to emulate the performance characteristics of NVM with DRAM. In order to emulate a hybrid memory system, memory access trace based simulation approaches have been widely used in previous studies. Fast x86-64 multi-core simulators such as Zsim [36] can collect memory access trace of workloads, and another cycle-accurate memory simulator such as NVMain [16] can simulate the memory accesses using the trace. However, these simulation approaches are extremely slow. There have been some DRAM-based memory emulators for NVMs [37–39]. Quartz [38] and HME [39] both model application perceived NVM access latency by injecting software created delays periodically. They exploit the DRAM thermal control interface provided by commodity Intel CPUs to limit the maximum memory bandwidth. However, Quartz and HME cannot work for QEMU/KVM platform. Therefore, we developed a lightweight evaluation method for hybrid memory systems in virtualization environments. The key idea is to measure the amount of read/write operations of each memory channel by monitoring the performance monitor counters of Intel CPUs. Intel also provides performance counter monitor (PCM) tools to count memory read/write throughput for each memory channel.

We implement HMvisor in physical machines equipped with two Intel Xeon E5-2650 v3 processors (Haswell, 20 CPU cores and 25 MB Last Level Cache) and 64 GB DDR4 DRAM. To mitigate the performance fluctuation in different experiments, we disable the dynamic voltage and frequency scaling (DVFS) control mechanism in the processors. The host operating systems are CentOS with Linux kernel 4.10.2 and the modifications of HMvisor. Each physical machine has two NUMA memory nodes and each node uses 32 GB DRAM. Node 0 and Node 1 are deemed as the DRAM node and NVM node, respectively. We launch a VM with 2 vCPU and 2 virtual NUMA nodes. The VM runs CentOS with Linux kernel 4.12 extended by NVM-ware memory balloon technology. We resize the capacity of hybrid memory in a VM on the fly by sending balloon instructions to QEMU. QEMU communicates with a guest OS driver to adjust the capacity of hybrid memory through Virtio, which is an I/O virtualization framework for Linux OS. We also develop QEMU interfaces to enable/disable the policies in HMvisor. In our experiments, the data are placed in the NVM node at first, and then HMvisor performs page migrations in 1 s.

We evaluate HMvisor using several workloads with different memory access patterns. Mcf, sjeng, wrf,

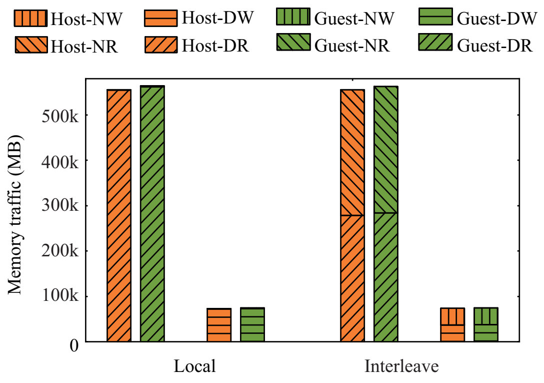

Figure 6 (Color online) Memory read/write traffic of mcf in host and guest OSes for two NUMA policies.

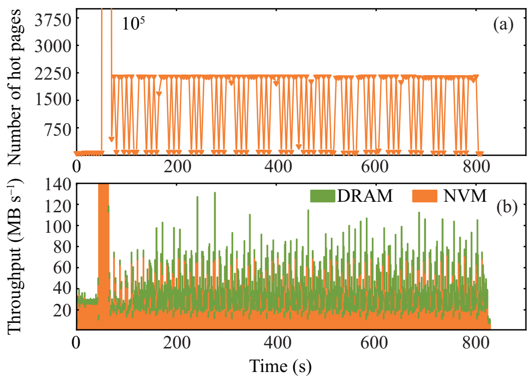

Figure 7 (Color online) (a) The number of hot pages in Graph500 varies over time; (b) the DRAM/NVM write traffic of Graph500.

bwaves, libquantum, GemsFDTD are selected from SPEC CPU2006. Mcf shows relative large memory footprint (1.7 GB) among those benchmarks. Graph500 is a data-intensive workload that searches large-scale, in-directed graph. In our experiments, we set the graph parameters as follows (scale=21, edgefactor=30), and the working set of Graph500 is approximately 1.5 GB. The execution of Graph500 consists of two phases. One is the generation of graph tree, and another is graph searching. We observe that there is a large amount of write traffic during the phase of graph tree generation, and the periodical write traffic is about 80 MB s^{-1} during the phase of graph searching.

4.2 Memory heterogeneity of VM

To verify the effectiveness of HMvisor for hybrid memory allocation to VMs, we check whether the total memory traffic of applications running in a bare-metal machine and a VM are equal. For the VM, its virtual NUMA nodes are mapped to different NUMA nodes, so that its main memory is composed of both DRAM and NVM. We run the same benchmarks in real machines and VMs with various NUMA policies. We count the memory read/write traffic from different memory channels. In Figure 6, we run mcf workload in host and guest operating systems with two NUMA memory allocation policies (local and interleave). The legend “Host-NR” denotes NVM read traffic when the workload runs on the host machine. The total memory read/write traffic in the VM is only 3% higher than the execution in the host owing to the virtualization overhead in all experiments. For the NUMA-interleave policy, the read/write traffic is evenly distributed on NVM and DRAM for both the two executions in the host and guest operating systems, respectively. These results have validated that the virtual NUMA nodes in the VM correctly map to the DRAM/NVM NUMA nodes. Therefore, HMvisor exposes the memory heterogeneity to VMs without any modification of guest OSes.

4.3 Performance studies of HMvisor

Unlike the traditional page hotness tracking mechanisms, HMvisor uses a lightweight approach to identify hot pages in VMs. To illustrate how HMvisor can effectively reduce write traffic to NVM, we obtain memory write trace of workloads in a short time window periodically, and then dynamically change the hotness threshold till the reduction of NVM write traffic is minimized with a fix-sized DRAM configuration.

Figure 7(a) shows the memory write pattern of Graph500. As we peer at the dirty bit of EPT entries in an interval of 0.1 s, the maximum value of the counter in each scan epoch that covers 50 sub-scan epochs is 50. Figure 7(a) only presents the number of guest pages whose write counts exceed 30 in each scan epoch (5 s). We observe that the amount of frequently-accessed pages is extremely large at the phase of graph tree generation, and the burst write throughput of those hot pages is up to 300 MB s^{-1}. After the initial phase, the memory write throughput declines. Another interesting observation is that the number of hot pages in the VM becomes stable. This motivates us to check whether these frequently-accessed pages have a significant impact on the total write traffic of Graph500. We ignore the initial phase and only migrate the hot pages during the normal execution. Figure 7(b) presents the DRAM/NVM write traffic of Graph500 when the hotness threshold is set to 30, and we enable the HMvisor mechanism

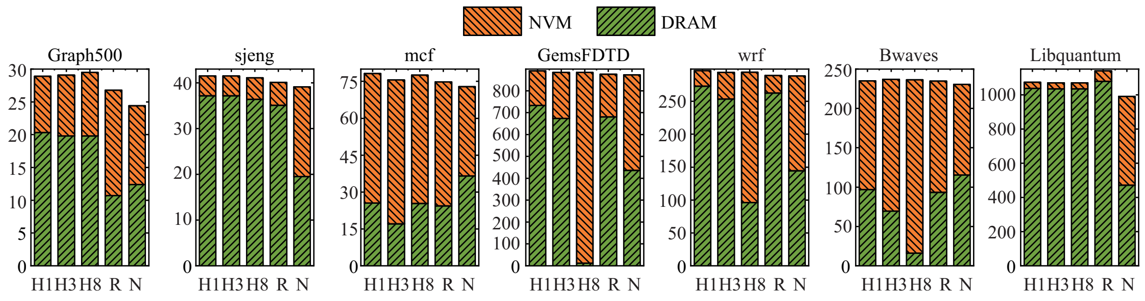

Figure 8 (Color online) The total amount of write traffic (in GB) on DRAM and NVM with different policies. "R" represents CMQ in RAMinate, "N" represents NUMA-Interleave, "H1, H3, H8" represent HMvisor with different hotness thresholds (1, 3, 8).

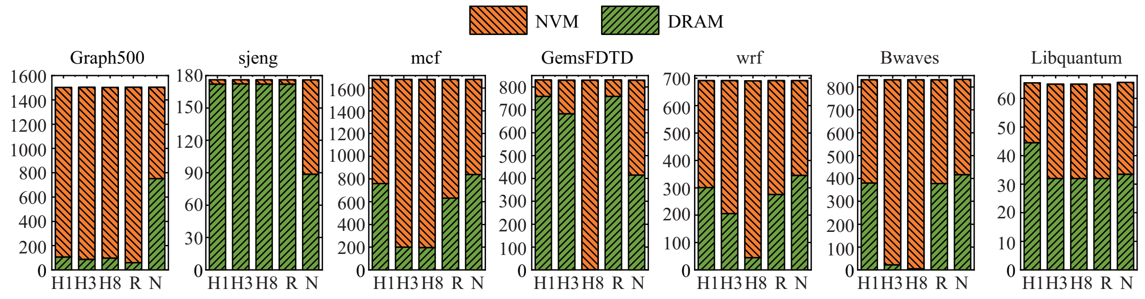

Figure 9 (Color online) The total amount of DRAM and NVM usage (in MB) with different policies. "R" represents CMQ in RAMinate, "N" represents NUMA-Interleave, "H1, H3, H8" represent HMvisor with different hotness thresholds (1, 3, 8).

Table 1 The values of $\mathrm{DRAM}_{\mathrm{utility}}$ with different policies

Policy	Graph500	sjeng	mcf	GemsFDTD	wrf	bwaves	Libquantum
HMvisor-1	9.88	0.93	0.72	0.90	2.12	0.89	1.43
HMvisor-3	11.7	0.93	1.9	0.93	2.9	11.27	1.94
HMvisor-8	10.3	0.93	2.79	0	5.07	11.33	1.94
RAMinate	10	0.9	0.87	0.85	2.27	0.87	1.89
Interleave	1	1	1	1	1	1	1

after the initial phase of Graph500. After about 140 s from the beginning, HMvisor migrates a total of 4098 memory pages in NVM to DRAM in 2 migration epochs. We find that almost half of the memory write traffic has been diverted to the DRAM node, although HMvisor only uses 16 MB DRAM. This demonstrates that HMvisor can effectively detect the hot pages in the guest OS and migrate them to the DRAM node.

We also compare the hotness tracking mechanism of HMvisor with the CMQ algorithm in RAMinate, and the baseline is the NUMA-Interleave policy. We note that a careful setting of the hotness threshold has a significant impact on the number of hot pages, and finally affects the reduction of memory write traffic to NVM. We evaluate HMvisor using several workloads with different memory access patterns. For the CMQ algorithm in RAMinate, we set the scan and migration epoch as 1 s, and set the decay_time as 5 s. For the hotness tracking algorithm in HMvisor, the sub-scan epoch is 0.1 s and scan epoch covers 10 sub-scan epochs. We use three different hotness thresholds (1, 3, 8) to evaluate the effectiveness of hot page identification. We set the number of multi-level linked lists as 8 for both algorithms.

Figure 8 shows the total amount of DRAM/NVM write traffic for different workloads under different policies. Correspondingly, Figure 9 shows the total amount of DRAM/NVM memory usage in each experiment. Table 1 shows the $\mathrm{DRAM}_{\mathrm{utility}}$ for different workloads. We can find that both HMvisor and RAMinate are much better than the NUMA-interleave policy, and the hotness tracking algorithm can efficiently identify frequently-accessed pages. Although a smaller hotness threshold leads to a high proportion of DRAM write traffic, it increases the total number of page migrations, which may cause non-trivial performance overhead owing to unnecessary page migrations. For Graph500, HMvisor diverts 68% of total write traffic to DRAM, while RAMinate only achieves 40%. The reason is that HMvisor can identify more frequently-accessed pages compared to RAMinate, although the $\mathrm{DRAM}_{\mathrm{utility}}$ values of

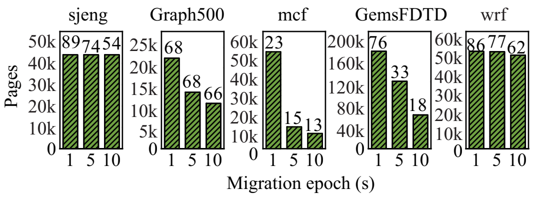

Figure 10 (Color online) The number of pages migrated in different migration epochs. The numbers over each bar represent the percentage of DRAM write over total write traffic.

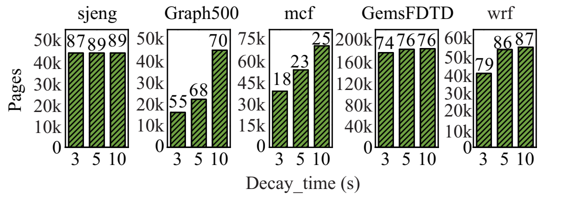

Figure 11 (Color online) The number of pages migrated with different decay_time. The numbers over each bar represent the percentage of DRAM write traffic over total write traffic.

Table 2 The normalized execution time with different policies.

Policy	Graph500 (%)	sjeng (%)	mcf (%)	GemsFDTD (%)	wrf (%)	bwaves (%)	Libquantum (%)
HMvisor-1	101	100	103	105	100	105	100
HMvisor-3	101	100	100	103	100	102	100
HMvisor-8	101	100	100	100	100	101	100
RAMinate	101	100	100	105	100	104	100
Interleave	100	100	100	100	100	100	100

both approaches are close. We find the proportion of NVM write traffic for mcf is still very high even when the hotness threshold is set to 1. Because the memory accesses of mcf are uniformly distributed on its total memory footprint, the ratio of DRAM write traffic to NVM write traffic is almost equal to the DRAM-to-NVM ratio for all policies. The DRAM write traffic of "H8" is 10% higher than that of "H3" for mcf, even though both policies have the same memory usage. This implies that the frequently-accessed pages can be migrated to DRAM more quickly with a larger hotness threshold. As sjeng and libquantum are typical memory-intensive workloads, HMvisor diverts most of memory write traffic to DRAM for both HMvisor and RAMinate. For GemsFDTD, when the hotness threshold is set to 8, HMvisor does not migrate any NVM page to DRAM. This implies that the page hotness in GemsFDTD is rather moderate, unlike libquantum. For wrf and bwaves, the hot page distribution is very sparse, with only a few extremely hot pages, and thus the DRAM write traffic declines with the increase of the hotness threshold. For bwaves, although the ratio of DRAM write traffic in "H8" or "H3" is smaller than that of RAMinate, the $\text{DRAM}_{\text{utility}}$ value of HMvisor is much higher than that of RAMinate. We also observe that the $\text{DRAM}_{\text{utility}}$ values of "R" and "H1" are smaller than that of NUMA-interleave policy, because both "R" and "H1" fail to identify hot pages in this workload. These experiments suggest that high accuracy of page access counting is very important for hot page identification. The hotness tracking algorithm in HMvisor performs much better than the CMQ algorithm in RAMinate. Moreover, we should carefully set the page hotness threshold for different workloads to balance the number of page migrations and the reduction of NVM write traffic.

Moreover, we also study how the decay_time and migration epoch impact on the number of NVM pages migrated to DRAM and the proportion of DRAM write traffic. Figure 10 shows the number of migrated pages vary with three migration epochs. We find that a smaller migration epoch usually leads to more page migrations, such as Graph500, mcf, and GemsFDTD. Also, the hot pages can be migrated to DRAM more quickly, leading to a higher proportion of DRAM write traffic. Figure 11 shows how the decay_times impact on hot page identification. Generally, a larger decay_time usually identifies more pages as hot pages and leads to more page migrations, such as Graph500 and mcf.

4.4 Performance overhead of page migrations

Table 2 shows the execution time of applications for different policies, all normalized to the NUMA-Interleave policy. In order to evaluate the performance overhead of page migrations, we ignore the difference of read/write latency between DRAM and NVM. The experimental results clearly reflect the performance overhead owing to page migrations in HMvisor. Unlike the studies [21,22] that perform page remapping in the hypervisor, HMvisor incurs slight performance degradation because it does not have to disrupt the VMs during page migrations. When the value of hotness threshold is set to 1, HMvisor only introduces about 5% of performance degradation for bwaves and GemsFDTD, because a large number of page migrations also lead to non-trivial performance overhead. Overall, the page migrations in HMvisor

Table 3 Hybrid memory configurations of VM1 (sjeng) and VM2 (mcf)

DRAM-to-NVM ratio	VM1-DRAM(MB)	VM1-NVM(MB)	VM2-DRAM(MB)	VM2-NVM(MB)
1:1	100	100	900	900
1:3	100	300	400	1200
1:4	100	400	300	1200

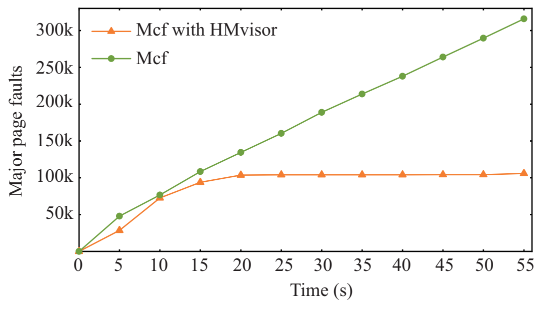

Figure 12 (Color online) The number of major page faults of mcf running in VMs with and without the support of HMvisor.

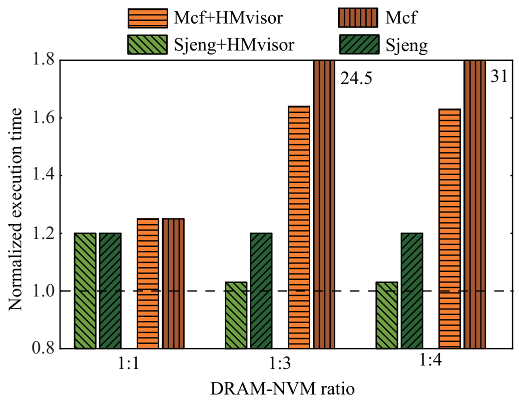

Figure 13 (Color online) Normalized application execution time of two VMs with and without HMvisor.

only result in very slight performance degradation when the page hotness threshold is set cautiously.

4.5 Dynamic memory adjustment

In the following, we evaluate the effectiveness of dynamic memory adjustment mechanism in HMvisor. In order to demonstrate our mechanism is able to improve the performance of applications with different memory access patterns, we start two VMs to run a memory-hungry workload (mcf) and a latency-sensitive workload (sjeng), respectively. The working set size of mcf is approximate 1.7 GB, and the memory accesses of mcf are uniformly distributed on its total memory footprint. In contrast, the working set size of sjeng is only 178 MB, and its memory pages are frequently accessed. The total memory allocated to the two VMs is limited to 2 GB, which is close to the total memory demand of the two workloads. Table 3 shows the initial hybrid memory allocation of VM1 (mcf) and VM2 (sjeng) with different DRAM-to-NVM ratios. We assume that the DRAM-to-NVM ratios after adjustment are equal to the initial settings. For example, we should trade 100 MB DRAM for 400 MB NVM when the DRAM-to-NVM ratio is 1:4. This ratio reflects the monetary cost of DRAM and NVM in the cloud environment. When all workloads begin to run, HMvisor performs dynamic memory adjustment among multiple VMs.

Figure 12 shows the accumulative major page faults of mcf in a VM with and without using the dynamic memory adjustment mechanism. The DRAM-to-NVM ratio of the VM is configured as 1:3. We observe that our NVM-aware ballooning mechanism is able to reduce the major page faults effectively and quickly. The major page faults do not increase after 20 s, implying that the memory demand is successfully satisfied through dynamic memory adjustment.

Figure 13 shows the application execution time under different settings of DRAM-to-NVM ratios with and without the dynamic memory adjustment mechanism in HMvisor, all normalized to the normal execution in a system with plenty of DRAM (called baseline). When the DRAM-to-NVM ratio is 1:1, both VMs can satisfy the applications' memory demands, and HMvisor does not perform resource trading between DRAM and NVM. However, because half of total data are placed on NVM, the application execution time is about 1.20×–1.25× relative to the baseline system. When the DRAM-to-NVM ratio is 1:3 and 1:4, the execution time of mcf (VM2) significantly increases owing to the large amount of major page faults. However, the dynamic memory adjustment mechanism in HMvisor can significantly reduce the execution time of mcf by up to 30×. Because VM1 migrates hot pages to DRAM, leaving a large portion of NVM unused, VM2 can trade some DRAM for more NVM to meet its memory demand. Finally, VM2 has traded 100 MB DRAM for 300 MB NVM with VM1. The dynamic memory adjustment only causes about 3% performance overhead. Apparently, the resource trading mechanism is able to benefit both VMs. For VM1, the execution time of sjeng decreases because the NVM is changed to DRAM and the DRAM capacity can meet its memory requirement. For VM2, the application performance of mcf is also improved by significantly eliminating the major page faults. Overall, our dynamic hybrid memory adjustment mechanism can best utilize the hybrid memories and improve the performance of all VMs.

5 Related work

There have been many studies on hybrid memory management for bare-metal machines. Dhiman et al. [24] extended the memory controller to perform hot page migrations between PCM and DRAM. Meswani et al. [14] exploited the stacked DRAM as a part of physical memory address, and proposed OS-level hybrid memory management. Chou et al. [15] modified the memory controller to manage DRAM and stacked 3D-DRAM in both cache and flat modes. Ham et al. [40] proposed disintegrated memory controllers for heterogeneous memory system with hardware-assisted page migration. Ramos et al. [17] exploited memory access pattern obtained from memory controller and proposed a hardware-driven page placement policy for hybrid memory systems. There are also a number of studies on software-managed hybrid memory systems. Lin et al. [11] presented user/kernel interfaces to place data in different memory regions. Kannan et al. [10] proposed an OS-level hybrid memory management system, and provided user-level NUMA libraries for users. Gandhi et al. [20] provided a software-based tool to distinguish hot pages by instrumenting x86-64 TLB miss in bare-metal machines. Dulloor et al. [6] focused on data classification and tiering, and proposed X-Men interfaces to map data structure of applications to different hybrid memories. These studies all focused on data classification and placement for hybrid memory systems in bare-metal machine environments. Most of them cannot be applied directly in virtualization environments.

Recently, there have been a few work of hybrid memory management policies in virtualization environments. Balloonfish [41] converted the memory cell state between MLC and SLC to achieve a balance between performance and memory space under mobile virtualization. Thermostat [19] proposed an online hot/cold data classification mechanism to distinguish the hotspot in huge pages for cloud applications. HeteroOS [13] focused on static data placement in a hybrid memory systems in the OS layer. RAMinate [22] and HeterVisor [21] performed page migrations in the hypervisor layer, and they had to disrupt VMs during the page migrations to maintain data consistency. HMvisor performs page hotness tracking in the hypervisor and migrates hot pages within VMs without disrupting VMs. HMvisor also exposes memory heterogeneity to VM by mapping virtual NUMA nodes to physical NUMA nodes, and thus can facilitate the migration of application-level optimization mechanisms to virtualization environments.

6 Conclusion

In this paper, we propose a hypervisor and VM coordinated hybrid memory management mechanism (called HMvisor). It reduces NVM write traffic by lightweight page migration in hybrid memory system. Our mechanism can detect frequently-accessed pages and dynamically divert more than 50% memory write traffic to DRAM. HMvisor can also perform dynamic memory adjustment among multiple VMs. Our experiments demonstrate that HMvisor is effective and efficient for optimizing memory-hungry and latency-sensitive VMs resided in the same host, and achieves better application performance in hybrid memory systems. The performance overhead of page migration is less than 5%.

Acknowledgements This work was supported jointly by National Natural Science Foundation of China (Grants Nos. 61672251, 61732010, 61825202).

References

1 McKee S. Reflections on the memory wall. In: Proceedings of the 1st Conference on Computing Frontiers, Ischia, 2004. 162–167

2 Meena J, Sze S, Chand U, et al. Overview of emerging nonvolatile memory technologies. Nanoscale Res Lett, 2014, 9: 526

3 Pistol C, Chongchitmate W, Dwyer C, et al. Architectural implications of nanoscale integrated sensing and computing. SIGARCH Comput Archit News, 2009, 37: 13

4 Li J, Lam C. Phase change memory. Sci China Inf Sci, 2011, 54: 1061–1072

5 Hady F T, Foong A, Veal B, et al. Platform storage performance with 3D XPoint technology. Proc IEEE, 2017, 105: 1822–1833

6 Dulloor S R, Roy A, Zhao Z, et al. Data tiering in heterogeneous memory systems. In: Proceedings of the 11th European Conference on Computer Systems, London, 2016. 1–16

7 Qureshi M K, Srinivasan V, Rivers J A. Scalable high performance main memory system using phase-change memory technology. SIGARCH Comput Archit News, 2009, 37: 24–33

8 Black B, Annavaram M, Brekelbaum N, et al. Die stacking (3D) microarchitecture. In: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture, Orlando, 2006. 469–479

9 Radulovic M, Zivanovic D, Ruiz D, et al. Another trip to the wall: how much will stacked DRAM benefit HPC? In: Proceedings of 2015 International Symposium on Memory Systems, Washington, 2015. 31–36

10 Kannan S, Gavrilovska A, Schwan K. pVM: persistent virtual memory for efficient capacity scaling and object storage. In: Proceedings of the 11th European Conference on Computer Systems, London, 2016. 1–16

11 Lin F X, Liu X. Memif: towards programming heterogeneous memory asynchronously. SIGARCH Comput Archit News, 2016, 44: 369–383

12 Coburn J, Caulfield A M, Akel A, et al. NV-Heaps: making persistent objects fast and safe with next-generation. SIGPLAN Not, 2012, 47: 105–118

13 Kannan S, Gavrilovska A, Gupta V, et al. HeteroOS: OS design for heterogeneous memory management in datacenters. SIGOPS Oper Syst Rev, 2018, 51: 13–26

14 Meswani M R, Blagodurov S, Roberts D, et al. Heterogeneous memory architectures: a HW/SW approach for mixing die-stacked and off-package memories. In: Proceedings of 2015 IEEE 21st International Symposium on High Performance Computer Architecture, San Francisco, 2015. 126–136

15 Chou C, Jaleel A, Qureshi M. BATMAN: techniques for maximizing system bandwidth of memory systems with stacked-DRAM. In: Proceedings of International Symposium on Memory Systems, Alexandria, 2017. 268–280

16 Meza J, Chang J, Yoon H B, et al. Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management. IEEE Comput Arch Lett, 2012, 11: 61–64

17 Ramos L E, Gorbatov E, Bianchini R. Page placement in hybrid memory systems. In: Proceedings of International Conference on Supercomputing, Tucson, 2011. 85–95

18 Dong X, Xie Y, Muralimanohar N, et al. Simple but effective heterogeneous main memory with on-chip memory controller support. In: Proceedings of 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, 2010. 1–11

19 Agarwal N, Wenisch T F. Thermostat: application-transparent page management for two-tiered main memory. In: Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems, Xi'an, 2017. 631–644

20 Gandhi J, Basu A, Hill M D, et al. BadgerTrap. SIGARCH Comput Archit News, 2014, 42: 20–23

21 Gupta V, Lee M, Schwan K. BadgerTrap: a tool to instrument x86-64 TLB misses. SIGPLAN Not, 2015, 50: 79–92

22 Hirofuchi T, Takano R. RAMinate: hypervisor-based virtualization for hybrid main memory systems. In: Proceedings of the 7th ACM Symposium on Cloud Computing, Santa Clara, 2016. 112–125

23 Kivity A, Kamay Y, Laor D, et al. KVM: the Linux virtual machine monitor. In: Proceedings of Linux Symposium, Ottawa, 2007. 225–230

24 Dhiman G, Ayoub R, Rosing T. PDRAM: a hybrid PRAM and DRAM main memory system. In: Proceedings of the 46th Annual Design Automation Conference, San Francisco, 2009. 664–669

25 Zhang W, Li T. Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures. In: Proceedings of 2009 18th International Conference on Parallel Architectures and Compilation Techniques, Raleigh, 2009. 101–112

26 Lee S, Bahn H, Noh S H. CLOCK-DWF: a write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures. IEEE Trans Comput, 2014, 63: 2187–2200

27 Dybdahl H, Stenstrom P, Natvig L. An LRU-based replacement algorithm augmented with frequency of access in shared chip-multiprocessor caches. SIGARCH Comput, 2007, 35: 45–52

28 Liu H, He B. Reciprocal resource fairness: towards cooperative multiple resource fair sharing in IaaS clouds. In: Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis, New Orleans, 2014. 970–981

29 Liu H, He B. F2C: enabling fair and fine-grained resource sharing in multi-tenant IaaS clouds. IEEE Trans Parallel Distrib Syst, 2016, 37: 2589–2602

30 Chen H G, Wang X L, Wang Z L, et al. DMM: a dynamic memory mapping model for virtual machines. Sci China Inf Sci, 2010, 53: 1097–1108

31 Russell R. Virtio: towards a de-facto standard for virtual I/O devices. ACM SIGOPS Operat Syst Rev, 2008, 42: 95–103

32 Liu H, Jin H, Liao X, et al. Hotplug or ballooning: a comparative study on dynamic memory management techniques for virtual machines. IEEE Trans Parallel Distrib Syst, 2015, 26: 1350–1363

33 David R. Linux fake NUMA patch. 2007. https://www.kernel.org/doc/Documentation/x86/x86_64/fake-numa-for-cpusets

34 Luk C K, Cohn R, Muth R, et al. Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, 2005. 190–200

35 Zhou Y, Chen Z, Li K. Second-level buffer cache management. IEEE Trans Parallel Distrib Syst, 2004, 15: 505–519

36 Sanchez D, Kozyrakis C. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. SIGARCH Comput Archit News, 2013, 41: 475–486

37 Dulloor S R, Kumar S, Keshavamurthy A, et al. System software for persistent memory. In: Proceedings of the 9th European Conference on Computer Systems, Amsterdam, 2014. 1–15

38 Volos H, Magalhaes G, Cherkasova L, et al. Quartz: a lightweight performance emulator for persistent memory software. In: Proceedings of the 16th Annual Middleware Conference, Vancouver, 2015. 37–49

39 Duan Z, Liu H, Liao X, et al. HME: a lightweight emulator for hybrid memory. In: Proceedings of 2018 Design, Automation Test in Europe Conference Exhibition, Dresden, 2018. 1375–1380

40 Ham T J, Chelepalli B K, Xue N, et al. Disintegrated control for energy-efficient and heterogeneous memory systems. In: Proceedings of 2013 IEEE 19th International Symposium on High Performance Computer Architecture, Shenzhen, 2013. 424–435

41 Long L, Liu D, Liang L, et al. Morphable resistive memory optimization for mobile virtualization. IEEE Trans Comput-Aided Des Integr Circuits Syst, 2016, 35: 891–904