FlexHM: A Practical System for Heterogeneous Memory with Flexible and Efficient Performance Optimizations

BO PENG, Shanghai Jiao Tong University, China
YAOZU DONG, Intel Asia-Pacific Research and Development Ltd., China
JIANGUO YAO, Shanghai Jiao Tong University, China
FENGGUANG WU, Intel Asia-Pacific Research and Development Ltd., China
HAIBING GUAN, Shanghai Jiao Tong University, China

With the rapid development of cloud computing, numerous cloud services, containers, and virtual machines have been bringing tremendous demands on high-performance memory resources to modern data centers. Heterogeneous memory, especially the newly released Optane memory, offer appropriate alternatives against DRAM in clouds with the advantages of larger capacity, lower purchase cost, and promising performance. However, cloud services suffer serious implementation inconvenience and performance degradation when using hybrid DRAM and Optane memory. This article proposes FlexHM, a practical system to manage transparent heterogeneous memory resources and flexibly optimize memory access performance for all VMs, containers, and native applications. We present an open-source prototype of FlexHM in Linux with several main contributions. First, FlexHM raises a novel two-level NUMA design to manage DRAM and Optane memory as transparent main memory resources. Second, FlexHM provides flexible and efficient memory management, helping optimize memory access performance or save purchase costs of memory resources for differential cloud services with customized management strategies. Finally, the evaluations show that cloud workloads using 50% Optane slow memory on FlexHM can achieve up to 93% of the performance when using all-DRAM, and FlexHM provides up to 5.8× improvement over the previous heterogeneous memory system solution when workloads use the same ratio of DRAM and Optane memory.

CCS Concepts: • **Computer systems organization** → **Architecture**; *Secondary storage organization*; • **Software and its engineering** → Virtual machines; Memory management;

Additional Key Words and Phrases: NVM, heterogeneous memory management, two-level NUMA, page hotness, conservative page migration

ACM Reference format:
Bo Peng, Yaozu Dong, Jianguo Yao, Fengguang Wu, and Haibing Guan. 2022. FlexHM: A Practical System for Heterogeneous Memory with Flexible and Efficient Performance Optimizations. *ACM Trans. Arch. Code Optim.* 20, 1, Article 13 (December 2022), 26 pages.
https://doi.org/10.1145/3565885

This work was supported in part by the Program for NSFC (No. 62032008), the STCSM project (No. 20510712400), Ministry of Education-China Mobile Research Foundation (No. MCM20180703), Alibaba Innovative Research (AIR) Program, and SJTU-Enflame Joint Lab.
Authors' addresses: B. Peng, J. Yao, and H. Guan, Shanghai Jiao Tong University, Shanghai, China; emails: {pengbo_michael, jianguo.yao, hbguan}@sjtu.edu.cn; Y. Dong and F. Wu, Intel Asia-Pacific Research and Development Ltd., Shanghai, China; emails: eddie.dong@intel.com, wfg@mail.ustc.edu.cn.

1544-3566/2022/12-ART13 $15.00
https://doi.org/10.1145/3565885

1 INTRODUCTION

Memory is one of the most performance-critical resources in cloud infrastructures relevant to the **Quality of Service (QoS)** of numerous memory-intensive native services, containers, and **virtual machines (VM)**. The **cloud service providers (CSP)** usually keep oversubscribing the memory resources and increasing the density of memory-intensive cloud workloads on each single cloud server for higher profits. Suppose the memory resource demands of the cloud services exceed the DRAM capacity of one cloud server. In that case, the **cloud service providers (CSP)** can only turn to distributed memory systems, while distributed systems seriously degrades the memory resource scalability and utilization [35] compared with one large-scale local memory system. With the commercial release of Intel Optane DC persistent memory (referred to in this article simply as Optane memory) [19], the **slow main memory (SMM)** can provide reliable main memory alternatives to combine with DRAM to form **heterogeneous main memory (HMM)** systems for cloud servers. Optane memory is one implementation of the non-volatile memory that embraces the byte-addressable feature and enriches the memory hierarchy with its advantages of large capacity and low purchase-cost per capacity. Now, the largest Optane is 512 GB per blade while the largest commercially available DRAM is 128 GB, and Optane is roughly 1/3 ∼ 1/2 prices of DRAM per GB [6, 16]. However, Optane shows a non-negligible performance gap with DRAM if used as main memory; for instance, the peak read bandwidth of Optane is 1/3 of DRAM, and the peak write bandwidth is 1/6 of DRAM [20].

Given the performance and prices of heterogeneous memory, the most essential concern of **heterogeneous main memory (HMM)** system design is how the system can optimize the HMM resource utilization to reach a promising service performance with a more affordable memory resource cost. Specifically, the bare-metal cloud native applications, containers, and virtual machines on the cloud raise several fundamental requirements to the HMM systems. For example, (1) They prefer using transparent heterogeneous memory to modifying their source codes and adapt their programming models with some libraries or tools like PMDK [41] for Optane, since PMDK incurs heavy programming and testing works for memory performance optimization on Optane, and it limits the convenience and flexibility of framework transplanting of more future slow memory devices rather than Optane. (2) Their various memory usage purposes and diverse memory access patterns, along with their multiple levels of QoS or **Service Levels of Agreement (SLA)** about the memory capacity and the overall throughput and latency performance, significantly increase the management difficulty and finally influencing their overall memory performance when using HMM. (3) They usually show imbalanced data distribution of memory usage over the long running time so they have differentiated willingness of resource payment. For example, the majority memory accesses of enterprise services or VMs appear in a minority of the daily time [12], and the personal guest machines run idle at the majority of their lifetime.

The memory management is significant for cloud memory performance guarantees and optimization in OS designs, especially for the heterogeneous memory cloud systems. The virtual memory management [7], memory virtualization [58], and **Non-uniform memory access (NUMA)** [28] are fundamental and generic OS memory management techniques. Many previous works studied efficient HMM management solutions for modern systems by using the DRAM-emulated non-volatile memory [1, 4, 10, 25, 30, 33, 38, 55] or memory controller simulators [5, 17, 32, 40, 42, 46, 52]. However, with the release of Optane memory, recent research [34, 49, 57] has proved that the previous emulation-based solutions cannot perform efficiently on the practical Optane hardware as well as on those emulated memory devices. Therefore, building a practical system to support and manage real DRAM and SMM devices (for example, Optane) and optimize performance and utilization for native memory workloads, containers, and guest machines attracts the attention of academic areas and OS developer communities.

In this article, we propose FlexHM, a practical heterogeneous main memory system with a flexible and efficient HMM management mechanism so cloud applications, containers, and virtual machines can use HMM transparently and adopt customized management policies for performance and price optimization. FlexHM builds a two-layer NUMA (distance and heterogeneity peer) based on the traditional one-layer NUMA of mainstream OSes to provide transparent HMM resources and memory virtualization support. Then, FlexHM involves the HMM management based on the periodical interaction between the memory performance monitoring in OS kernel space and the utilization analysis and optimization in the OS user space. Specifically, we study the aggressive and conservative management policies of FlexHM and prove that FlexHM can flexibly and efficiently handling the differential HMM usage management and performance optimization requirements from diverse cloud workloads. In the end, we demonstrate the evaluation results of FlexHM with comprehensive memory-intensive workloads with micro-benchmarks and application benchmarks (in-memory database and graph-computing benchmarks). For example, (1) the micro benchmarks using fixed ratios of DRAM and Optane **slow main memory (SMM)** can achieve up to 5.8 times over Thermostat [1]; (2) the micro benchmark using HMM (DRAM: Optane = 1:1) achieves up to 93% performance of a baseline using all DRAM; (3) the PageRank application benchmark can use lower percentages of DRAM resources (for example, DRAM: Optane = 1:3) to achieve 5.7× performance improvement and save 50% expensive DRAM resources over the state-of-the-art solution (using DRAM: Optane = 1:1.)

In summary, the article makes the following contributions:

- We propose FlexHM, a practical and open-sourced system for flexible and efficient memory performance and utilization optimization in the cloud where all guests, containers, and native applications can transparently and efficiently use heterogeneous main memory.
- We implement a prototype of FlexHM in Linux to manage transparent heterogeneous main memory including DRAM and the real NVM hardware Optane memory and add necessary kernel modules for fundamental HMM management, open-sourced at https://git.kernel.org/pub/scm/linux/kernel/git/wfg/linux.git.
- We design flexible and efficient HMM management on FlexHM that can flexibly adopt customized HMM management policies to efficiently handle the differential HMM usage demands, open-sourced at https://github.com/intel/memory-optimizer for the community to further develop more HMM management strategies.
- We do thorough evaluations to prove that FlexHM system can provide efficient and flexible HMM management for native or guest memory-intensive workloads.
- We discuss the management efficiency when FlexHM chooses aggressive or conservative management policies and the pros & cons of FlexHM with the hardware cache such as Optane two-level Memory Mode.

The rest of this article is organized as follows: We introduce the background and motivation of FlexHM in Section 2. We introduce the overall design of FlexHM system in Section 3, with the two-layer NUMA for HMM in Section 4, and the flexible and efficient HMM management and performance optimization in Section 5. We introduce the implementation details of the Linux prototype in Section 6. We demonstrate the evaluation results in Section 7. Discussions of FlexHM's overhead and the pros/cons between the HMM systems using DRAM as the cache are in Section 8. We conclude this article in Section 9.

2 BACKGROUND AND MOTIVATION

2.1 Heterogeneous Main Memory

Intel Optane DC persistent memory (also known as Optane NVDIMM, referred to in this article simply as Optane memory) [19] is a high-performance commercially available NVM device. Optane memory uses the 3D XPoint technique with a larger density of memory particles and can provide larger-capacity and more cost-affordable main memory resources than DRAM. Previous works [20, 56] have demonstrated that Optane shows similar idle latency but about 5× random read load latency and 2× random sequential load latency when compared with DRAM. We also build experiments with Intel Memory Latency checker [51] and demonstrate the measured sequential and random read or write performance results in Figure 1. From the results, we can find that Optane memory has more deficient random read or write performance than the sequential read or write performance of DRAM, and even Optane is the most advanced slow main memory device. So, it is obvious to infer that using non-volatile memory to replace volatile memory in OSes and rebuilding the current computing architecture is still unrealistic and unacceptable.

Optane memory can significantly increase the local memory capacity of one cloud server and increase the density of services, containers, and virtual machines. We can envision that Optane and some future SMM devices can co-exist with DRAM as heterogeneous main memory of cloud systems in a long term. Currently, there are two main usage modes of Optane, which are the **APP direct (AD)** mode and the **memory (MM)** mode. In AD mode, the OS initially treats Optane memory devices as byte-addressable but persistent devices (not like main memory, but similar to flash and disks). In MM mode, Optane memory uses the DRAM from the same memory channels to build an inclusive hardware cache model [21], and the OS can directly use Optane just like the volatile main memory. So, the MM mode is a preliminary heterogeneous main memory hardware solution for modern system, which can load and flush the DRAM cache line from/to SMM with fixed cache replacement algorithms, such as direct-map or LRU. However, this hardware solution has three main shortcomings. (1) With growing SMM capacity, the hardware cache implementations will be more complicated and impractical [32, 42]. (2) Using DRAM as the inclusive hardware cache sacrifices the total memory capacity of the entire system; for example, when a two-socket server equips 12 64 G DRAM blades and 12 512 GB Optane blades, it will waste over 10% total memory capacity. (3) The hardware cache can only use rigid cache replacement policies (e.g., direct map, pseudo-LRU), which is not flexible in addressing different cloud workload scenarios.

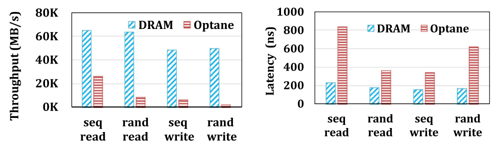
2.2 Heterogeneous Memory Management

The memory management for heterogeneous main memory resources has been a research hot spot for a long time [1, 13, 14, 23, 24, 30, 31, 38, 43, 44, 50, 53–55], and we summarize some significant research works in Table 1. Specifically, Thermostat [1] is a typical system that builds an application-transparent huge-page-aware mechanism to place pages with online page hotness classification on virtual NUMA of QEMU. Thermostat provides an insight into managing SMM with page hotness, but the solution is VM-limited and cannot meet the requirement from native cloud processes and containers. Also, a limitation of these related works is that they usually use DRAM to emulate slow NVM or use memory simulators to build a simulated heterogeneous memory environment, so these works do not fully consider the behaviors and performance of real NVM hardware to provide efficient optimization.

Specifically, we make a preliminary evaluation about how these previous works behave on the practical HMM hardware. We choose an open-sourced implementation of Thermostat from Github [37], build the system, and use the virtual NUMA of QEMU to run a 32-VCPU virtual machine that can manage the real DRAM and Optane memory as its HMM resource. We run a 32-thread

Table 1. Famous Research Works on the System Design and Resource Management of Heterogeneous Memory

Systems	Slow Main Memory	Design Level	Application Usage	VM Usage	Management Policy
PoM [46], BuMP [52], Moneta [5, 32]	Hardware simulator	Hardware Controller	Transparent	Not design	Fixed
Thermostat [1], MemBrain [38, 55], CLOCK-DWF [10, 30]	DRAM emulated	OS	Transparent	Not design	Fixed
HeteroOS [23], Heterovisor [14]	DRAM emulated	Hypervisor	Not Design	Yes	Fixed
CoMerge [9, 50], Ramos [43]	DRAM emulated	Application	Modified	Not Design	Fixed
FlexHM (Our work)	Optane	OS	Transparent	Yes	Flexible



(a) Throughput. (b) Latency.

Fig. 1. MLC Load latency and throughput results of DRAM and Optane in one NUMA node (six DRAM and six Optane blades).

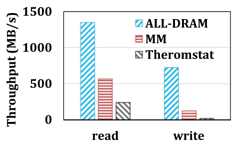


Fig. 2. Sysbench micro benchmark results on Thermostat, MM mode, and all-DRAM baseline.

Sysbench random read and write micro benchmarks in this VM, and we compare the throughput performance between Thermostat, the MM mode (where the server's DRAM capacity is 1/4 of its total Optane capacity), and an all-DRAM baseline. We demonstrate the memory access throughput performance results in Figure 2. From the experimental data, we can find that when we let a guest machine using 25% DRAM and 75% Optane memory as its memory resources (using the same capacity resources as the MM mode environment), the guest machine can only separately provide 18% random read performance and only 3.1% random write performance compared with using all DRAM. The performance results of Thermostat are also much worse than using MM mode to provide heterogeneous memory for this VM.

We further analyze that the reasons why Thermostat cannot work efficiently on real heterogeneous main memory hardware as on the slow memory emulated by DRAM. According to the implementation and evaluation environment in Reference [1], the design of Thermostat is based on an important assumption of the heterogeneous memory performance, which is the slow memory has a latency of $\approx 1\mu s$ but no less throughput than fast DRAM memory. However, even though Optane memory is one of the most advanced slow memory hardware and it can work better in latency performance than the assumption, there is a very significant throughput performance gap between Optane and DRAM, especially random read/write performance. Since the performance optimization of Thermostat is based on migrating the hot data into fast memory and cold data into slow memory in its system kernel, it will generate very large memory page movement requirement between DRAM and Optane memory, so it incurs heavy memory overhead of the kernel. When Thermostat uses very aggressive policy to conduct hotness collection and migration of the heterogeneous memory pages, the throughput performance of Optane may become a bottleneck and may incur more memory access and migration overhead between DRAM and Optane memory than the performance optimization effect it gains. Thermostat can give efficient optimization for some application benchmarks with low system memory throughput of random read/write, which will not bring serious hotness collection and migration overhead in the system kernel. But it cannot flexibly change its policy to reach a better balance between the performance improvement and the overhead when intensive workloads occupy the peak bandwidth of Optane memory.

So, we aim to design and implement FlexHM to provide flexible and efficient HMM management and performance optimization for cloud computing. We infer that FlexHM must have the following functions to handle the sophisticated and differential demands of heterogeneous memory resources from the native services, containers, and virtual machines of cloud systems:

(1) Supporting transparent HMM usage and full HMM virtualization: We aim to provide all native cloud workloads, containers, and guest machines with flexible capacities of transparent HMM resources without any modification to their source codes.

(2) Optimizing the performance of a workload using a fixed ratio of HMM: We aim to provide a runtime memory management for a workload with a fixed ratio of HMM to dynamically move hot data from its SMM area into the DRAM area and swap cold data into the SMM area to achieve a more optimal memory access performance and resource utilization.

(3) Optimizing the HMM utilization to reach more affordable prices with acceptable performance degradation: We aim to save the memory purchase cost of an HMM workload with a slight and acceptable performance sacrifice but meet their QoS. We may slightly reduce the DRAM percentage, increase the SMM (Optane NVM) percentage, and dynamically pick out and swap cold data into SMM.

3 FLEXHM OVERVIEW

We design FlexHM, a practical system to manage heterogeneous main memory with an efficient and flexible runtime memory management mechanism to provide transparent and fast DRAM and other SMM (e.g., Optane memory) and optimize utilization. An overview of the FlexHM architecture is shown in Figure 3, where we introduce the significant designs from the hardware configuration to the system kernel and the user-and-kernel-level-cooperated HMM management.

FlexHM Hardware Configuration: FlexHM is a practical system building on the real heterogeneous memory environments, for example, using DRAM as fast main memory and Optane memory as slow main memory. Unlike the Memory Mode of Optane (Optane uses all the DRAM as its hardware cache), the Optane resources used in the FlexHM system are configured as APP Direct mode in the server BIOS. So, FlexHM wastes no capacity of DRAM, and the kernel of FlexHM can manage all the DRAM and Optane capacity as available memory for all processes. This configuration helps to improve more profits of all the expensive memory resources of cloud servers.

FlexHM kernel: We design and implement the FlexHM kernel functions based on Linux. First, the FlexHM kernel enables transparent resource management of heterogeneous memory hardware (for example, DRAM and Optane). Since Optane is originally supposed to be used as a faster storage device when configured with the APP Direct Mode, we update the original memory management functions in the Linux kernel to manage the Optane as slow but application-transparent memory resources. To keep the kernel clean and concise and to improve the efficiency of HMM management and performance optimization, we design a user-and-kernel-level-cooperated HMM management mechanism to improve the management flexibility and efficiency, and the FlexHM kernel provides fundamental supports for a user-level manager. Specifically, we introduce a “kvm-ept-idle” module for checking the memory access patterns of the bare-metal native services and guest machines into the FlexHM kernel. The module can provide reliable memory usage information but will not

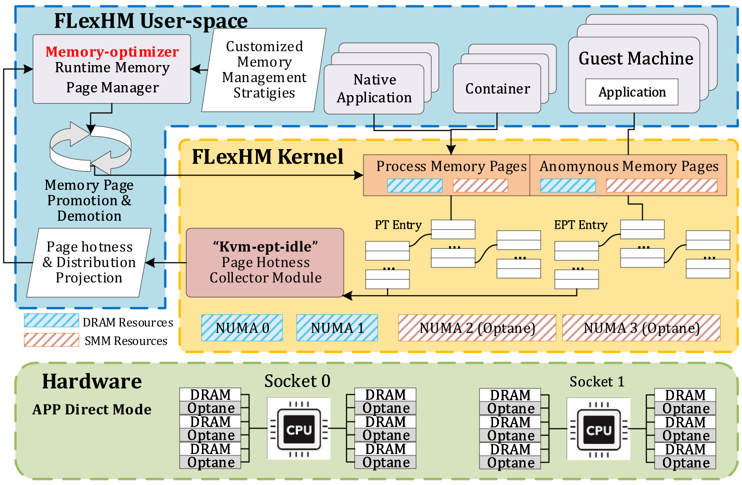


Fig. 3. An overview of FlexHM system design.

directly trigger any memory management action for any system processes. FlexHM can inherit all the basic memory management modules of the current Linux system, and we only introduce a few memory page management policies into the FlexHM kernel to optimize the heterogeneous memory allocation performance of each process.

FlexHM User-level Manager: We further design an HMM resource manager named Memory-optimizer in the user space of FlexHM. The manager can determine any ratio between the fast and slow memory utilization for each system process and provide performance optimization and utilization management. The manager can adopt different customized management policies for different cloud applications, containers, or guests to reach their diverse demands on HMM resources or optimize HMM resource utilization. Specifically, it can use the memory reference information from the kernel and analyze the memory patterns of workloads and trigger memory movement between heterogeneous memory. If the workloads want to optimize performance on a fixed HMM ratio, then the manager can dynamically select its hot in-memory data into DRAM and swap the cold data into SMM. If the workloads want to use more SMM resources to save costs, then it can analyze the memory access patterns and increase the percentage of its SMM usage without missing the QoS.

4 TRANSPARENT HETEROGENEOUS MEMORY IN FLEXHM

In FlexHM system, Optane memory is configured as APP Direct mode in the server BIOS. However, instead of main memory, Optane with APP Direct mode is used initially as a faster storage device than NVMe SSDs when OSes build file systems on Optane. So, we must add necessary supports to the memory management functions of the operating system to manage DRAM and SMM like Optane as transparent main memory.

Considering that the **Non-uniform memory access (NUMA)** can efficiently manage the memory from different hardware sockets of each server, we design a two-layer NUMA in FlexHM to manage all HMM resources as application-transparent heterogeneous memory. The concept of two-layer NUMA is an extension of the traditional NUMA in the mainstream OSes. The traditional NUMA is a one-layer NUMA, because it only considers the CPU access distance, related to the local or remote access latency; that is to say, the traditional NUMA has only one feature to tell

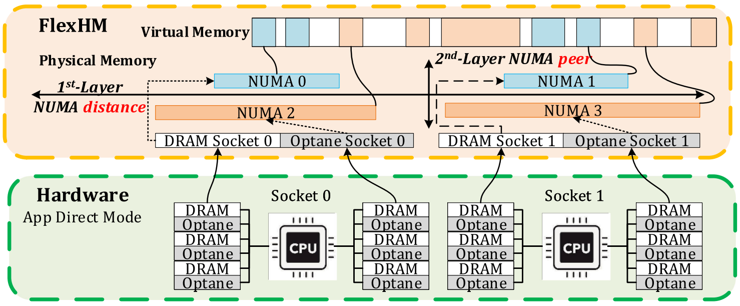


Fig. 4. Transparent memory from two-layer NUMA.

the information about hardware sockets, and it cannot accurately describe memory reference differences considering memory heterogeneity. Therefore, to manage memory heterogeneity, we introduce a concept called "NUMA peer" as the second-layer NUMA feature in addition to the first-layer NUMA distance feature. Therefore, the OS can separate all the heterogeneous memory devices from the memory controller of the same hardware socket into one DRAM peer node and one SMM peer node if there is only Optane NVDIMM, or several different SMM peer nodes if there are more than one SMM hardware installed in future servers. Together, the distance and peer features constitute the two-layer NUMA in FlexHM, and a FlexHM system can have *number-of-socket* * *number-of-memory-hardware* NUMA nodes. For example, we demonstrate how the two-layer NUMA manage DRAM and Optane in a two-socket server in Figure 4 with four (2 $\times$ 2) NUMA nodes; and NUMA 0 and 1 are the DRAM resources from the hardware socket 0 and 1, and NUMA 2, 3 are the Optane resources from the hardware socket 0 and 1. If there are other byte-addressable slow main memory devices in the future, then the two-layer NUMA design can also easily manage a more complicated memory heterogeneity with DRAM, Optane, and other SMM.

Since our FlexHM manages all the DRAM and Optane as transparent memory resources inside different NUMA peer nodes of the two-layer NUMA, we introduce two new notions: promotion and demotion for page management. Specifically, a NUMA peer node can have promotion and demotion targets in FlexHM two-layer NUMA. For example, memory pages can be demoted from a DRAM peer node into an SMM (Optane memory) peer node and finally be reclaimed into a swap area, and the promotion is vice versa. If we have more than one SMM, then a demotion can start from DRAM to a faster SMM node, then to slower SMM nodes, and so on, and the promotion is vice versa. The promotion/demotion is fundamental for memory page management of FlexHM, and it gives a generic interface for the kernel or user-level processes to move the memory pages between different NUMA peer nodes.

The two-layer NUMA in FlexHM can manage all capacities of heterogeneous main memory and provide transparent memory resources for all system processes. Also, FlexHM supports the full HMM virtualization and has no conflict with the usage of hardware-assisted virtualization support (for example, the **Extend Page Table (EPT)**). Therefore, from the aspect of the CSPs, every single server running the FlexHM system can increase the deployment density of cloud applications, containers, and virtual machines using heterogeneous main memory, and CSPs can increase the profits because of the better memory resource scalability. From the aspect of cloud users, FlexHM is very convenient for them to use more affordable heterogeneous memory with no need for any source code modification. Also, FlexHM can give them the choices with more affordable memory resources for their workloads by using some percentage of slow memory to replace fast DRAM.

5 FLEXIBLE AND EFFICIENT HMM MANAGEMENT

FlexHM can now support transparent HMM usage and the full HMM virtualization based on the two-level NUMA design. However, compared with using all fast memory (for example, DRAM), the native applications, containers, and virtual machines using HMM will suffer serious performance degradation if they directly use HMM without efficient memory management. Therefore, by reference to the system requirements we inferred in Section 2.2, FlexHM must flexibly handle the sophisticated and differential requirement on heterogeneous memory usage and optimization targets from diverse cloud service tenants.

One straightforward idea for HMM management is to extend the current kernel-level memory management policies to handle the HMM management and optimization requirement from the memory-intensive cloud workloads. However, this design may have several shortcomings. On the one hand, since the native applications, containers, and virtual machines on cloud systems have various memory resource requirements and differential memory access patterns, it is very hard for the OS kernel to design a universe policy to manage and optimize the HMM differential workloads efficiently. On the other hand, if we design and adopt multiple HMM management policies in the kernel, then it will severely complicate the original kernel memory management functions, sacrifice the code maintenance convenience, and hurt kernel stability eventually.

We choose a user-and-kernel-level-cooperated design for the HMM management and optimization when we design FlexHM. Specifically, we only add several basic HMM management policies in the FlexHM kernel to ensure the correct memory allocation and management function from the view of operating systems (in Section 5.1). The kernel HMM management is usually triggered when new applications, containers, or VMs initialize on the FlexHM system, so this management is coarse-grained for each workload itself and is not a runtime management function. Besides, we design a more fine-grained and runtime HMM management mechanism in FlexHM using an interaction between a user-space manager and the kernel to provide flexible and efficient HMM management that can adopt customized management policies (in Section 5.2).

5.1 Memory Management in Kernel

We add three basic memory management policies in the FlexHM kernel corresponding to the two-level NUMA design so the kernel can wisely do memory allocation and management: (1) If a new application or guest VM initializes, the kernel allocates its memory pages from DRAM peer nodes by default rather than allocate pages by following the NUMA interleave policy. (Specifically, if there is no memory in DRAM nodes, then the page allocations directly fall on SMM nodes.) (2) The native applications or VMs running on one NUMA peer node explicitly allocate pages from the nodes with the same peer node. If there is no memory in this peer, then the kernel will allocate memory from the other peer node with the same NUMA distance. (3) The **page table (PT)** is forced to be allocated and kept in DRAM peer nodes. PT is performance-critical, because it is frequently fetched and updated when applications generate numerous random read/write operations on memory data, and near half of the memory accesses are caused by TLB misses when doing PT translation.

Moreover, the kernel can automatically trigger page demotion when it finds no free space in DRAM peer nodes, similar to that *kswapd* swaps pages from memory to the swap partition on disks. The kernel page demotion has just as low frequency as the *kswapd*, so this will not bring serious overhead to the kernel memory management. FlexHM also leverages the *kswapd* to periodically move the least-frequently used memory data of SMM into swap areas if the OS has any swap area. This helps to ensure that process on FlexHM will not allocate memory resources from the swap area and affect the system memory performance even if all DRAM and SMM resources of the OS are occupied.

These kernel management policies are simple, static, and generic for all processes in systems, which helps keep the OS kernel stable.

5.2 Runtime Page Management

For more efficient HMM management in FlexHM, we need a more fine-grained management mechanism in addition to kernel management. Typically, in an OS with memory pages, the reference of each data on each memory page is proportional to the cost of each CPU hardware cache line loaded and flushed, and the cache lines are filled with an interleave policy when cloud applications generate numerous read/write operations to their memory pages in the long term. Thus, we project that the memory read/write performance of each process is associated with all its page references. Moreover, because the byte-addressable heterogeneous main memory devices will behave with different page reference costs when CPU cache loads and flushes data on them, the overall performance is associated with the sum of page reference costs in the fast DRAM and SMM like Optane memory. So, we design a runtime page management mechanism in FlexHM, which can flexibly decide the usage percentages of fast and slow memory and follow customized aggressive or conservative HMM management policies.

In FlexHM, we design a user-level manager named Memory-optimizer to perform the runtime fine-grained page management for each native process, container, and virtual machine. The manager performs the page management in a periodical procedure in three main stages: **Collection, Analysis, and Action**. First, in **Collection** stage, the user-level manager calculates page reference frequency by checking page reference hotness. Then, in **Analysis** stage, the manager does page hotness analysis to project the runtime memory access patterns and the data reference distribution of optimization targets. Finally, the manager conducts memory migration between HMM according to the resource utilization ratios and the customized management policies to provide optimization for different workload scenarios in **Action** stage.

The primary objective of the FlexHM HMM page management is that it can identify the most frequently referenced SMM pages of the target memory workload as the **identified hot SMM pages** and those most idle-referenced DRAM pages as the **identified cold DRAM pages** according to the collection of all the reference information as correctly as possible. To be mentioned, the **identified hot SMM pages** and **identified cold DRAM pages** are measured values and may have some difference with the real hot/cold data of each workload. Hence, the FlexHM user-level manager is designed to flexibly take different management policies to increase the accuracy of hot/cold page monitoring, analysis, and identification during the runtime memory management and performance optimization.

We use Figure 5 to demonstrate the periodical page management procedures. The user-level manager periodically repeats the procedure within a short epoch e, including the collection, analysis, and action stages. Specifically, within each epoch e, the manager first conducts the hotness collection with a specified time interval i until a maximum reference count r to monitor and calculate how many times all the pages are referenced in one epoch e. Later, the manager analyzes the page hotness information of the pages in the $r * i$ time (within the entire epoch e) and projects the data distribution of the target workload and **identical hot/cold pages**. Finally, the manager selects a particular memory size of the identical hot SMM pages (along with identical cold DRAM pages) and conducts page promotion and demotion to optimize the target memory usage to reach its own management goal. At last, the manager sleeps and waits until the next epoch to reduce frequent page management overhead. We introduce more details in the following three paragraphs.

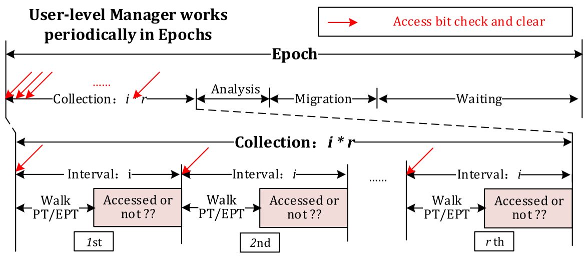


Fig. 5. How the user-level manager periodically works.

First, we introduce the collection stages, which are essential and fundamental for monitoring page reference information. We design a module named "*kvm-ept-idle*" in FlexHM kernel and update the original management to the **Page Table (PT)** and **Extend Page Table (EPT)**. Specifically, it adds two interfaces to check and clear the Access bits of PFNs of native process pages (including the native applications and regular containers) or the anonymous pages of virtual machines (such as QEMU). The interfaces can be called from the kernel space and the user space. So, the module can check if a page has been referenced whenever calling the interface. After that, the kernel module clears the Access bit to 0 if the bit is 1. Whenever rechecking this page Access bit, the changes of Access bits can tell whether the pages of the target process are referenced between two checks. In the FlexHM design, the user-level manager calls the interface of the *kvm-ept-idle* module for r times with an interval i. We use a hash map for each target in the user-level manager to store the HVA addresses and the peer node information of the target's memory pages, which also records the types of page references of these memory pages (e.g., read, write, exec) during the Collection stage. Hence, the manager can correctly check the memory pages of different target processes according to their *ProcMaps* structures and monitor the page reference information of the target process.

Second, FlexHM analyzes the measured page reference information of the target process and projects its page reference data distribution. Previous works have proved that memory reference distributions in systems, especially in cloud computing, follows Zipfian [15, 47], Gaussian [36], or Pareto [2]. If the page reference follows a specific data distribution, then it is friendly for FlexHM to improve the performance of random memory read or write based on this data distribution. Moreover, FlexHM can flexibly adopt customized page management policies for various cloud workloads with different memory reference distribution. In FlexHM, the user-level manager maintains a data structure for each target to store the reference counts of all the target workload memory pages. The page reference information is stored in the user-level memory areas so it will not incur heavy memory usage to the limited kernel memory resources. After the Collection stage, the manager can calculate and sort the page reference counts from r to 0 of all the pages in different **virtual memory areas (VMA)**. Then, FlexHM can analyze the virtual memory area containing the most frequently referenced pages and project that the workload follows a certain data distribution. The manager then examines whether these pages are located in DRAM or SMM peer nodes and analyzes the total size of the identified hot SMM (and cold DRAM) pages that should be moved.

Finally, During the Action stage, the manager can promote identical hot SMM pages into DRAM nodes and correspondingly demote the identical cold DRAM pages with the same sizes into SMM nodes by calling the *move_pages()* syscall between DRAM or SMM peer nodes. The page migration between the DRAM and SMM NUMA peer nodes will occupy some of the memory bandwidth of the DRAM and Optane memory hardware, so the manager sleeps and waits for a time until the next periodical management epoch to avoid a high management overhead.

5.3 Flexible Management Policies

In FlexHM runtime page management, the user-level manager uses the page reference distribution projection in the $r * i$ time as the representative of hotness distribution in an epoch e and finally projects the data reference distribution of the target workload. If the workload has a relatively stable and regular memory access pattern during its lifetime, then the projection in each *epoch* will be very accurate. While if the workload is generated by multiple processes and its memory pattern is not stable in a short period, then the manager may indicate different identified hot/cold pages in different epochs if the workload shows very irregular memory access patterns. So, using each *epoch* data distribution projection is not accurate and will cause some unnecessary hot/cold page movement, which will not optimize the performance efficiently and brings high management overhead.

FlexHM provides a flexible and comprehensive tuning mechanism of the i, r, e parameters in the user-level manager for designing and customizing different page management strategies. For interval, using a smaller interval i can increase the frequency of page hotness collection and help increase the accuracy of measured hotness information. However, this may incur extra overhead to the page table or EPT reference and reduce system memory performance. Similarly, using a smaller epoch e increases the page movement frequency for better HMM memory resource usage and better performance. However, this may incur extra overhead to the system memory, because the page movement between DRAM and SMM peer nodes will occupy a large memory bandwidth. Besides, a larger r parameter helps increase the accuracy of analyzing and describing the page reference distribution with noticeable features, but using a large but unsuitable r reduces the sensitivity of page movement and hurts the management efficiency.

Specifically, we introduce two main directions for designing different HMM management and optimization policies. One is the aggressive policy, which tries to promote all the identified hot pages from SMM to DRAM nodes and demotes identified cold pages in each epoch; the other is a conservative policy, which only promotes a certain percentage of identified hot pages and demotes identified cold pages with the same percentage in each epoch. For some memory workloads with stable memory access patterns, the aggressive policy can immediately reach an optimal resource utilization of heterogeneous memory and reduce long-term runtime page management overhead. For some intensive workloads whose memory hotness is irregular, choosing a conservative migration policy is better than the aggressive policy. The main reasons are as follows: (1) Frequent page migration of large numbers of pages can optimize page reference latency of target workloads but incur serious migration latency overhead. (2) The page hotness collected in each epoch may be very different, and we need to use it to project the future memory reference distribution. The conservative policy helps to improve projection accuracy and optimize management efficiency. (3) Page migration processes occupy memory bandwidth, and the conservative page migration can reduce the bandwidth interference between the migration process and target workloads. So, the Memory-optimizer manager of FlexHM also provide a parameter "*max_migration_size*" to decide the percentage of the identified hot/cold pages that need to be moved between DRAM and SMM after each epoch, to cooperate with the i, r, e parameters. We introduce the parameter choices in Section 6, and we use experiments to evaluate where we should choose aggressive or conservative policies with the percentage of identical hot/cold pages in Section 7.5.

5.4 Memory-optimizer *task-refs* and *sys-refs* Tools

When the optimization target workload is a single application on the host OS or inside a guest machine, we develop a *task-refs* tool of the user-level Memory-optimizer manager for memory performance and usage optimization. To deal with the workloads of all host processes (including the qemu-kvm processes), we developed a *sys-refs* tool to provide optimzation for multiple processes, and *sys-refs* can also manage all processes of the FlexHM system from the global perspective.

The *task-refs* tool uses the pid of the target process to provide customized performance and resource utilization optimization for one single native workload or virtual machine workload. Before starting the *task-refs* tool, the cloud administrators can reach an agreement with the cloud service tenants on the optimization target; for example, *task-refs* can optimize the performance of a cloud workload using a fixed ratio of HMM by providing runtime performance monitoring on HMM pages and trigger page migration between DRAM and SMM peer nodes or can move cold data from DRAM nodes into SMM nodes to optimize the HMM resources utilization and reach more affordable prices for each cloud workload. *task-refs* can flexibly accept the interval parameter i, maximum reference parameter r, epoch parameter e, and the "*max_migration_size*" to customize different management policy.

If a system faces complicated memory workloads spawn by multiple applications or virtual machines, which results memory interference and shortage in the system, then we design a more functional *sys-refs* tool to simultaneously optimize memory for different processes to reach a global optimization results when allocating heterogeneous memory resources for multiple workload. Algorithm 1 describes the basic procedures about how *sys-refs* works. *sys-refs* can accept a description of multiple target workload, for example "qemu-system-x86_64," to provide management for all the virtual machines running on the FlexHM system. If the system administrators do not appoint any target, then *sys-refs* can provide optimization for all the system processes according to their *pids*.

Similar to the *task-refs* tool, the *sys-refs* can accept parameters to customize different HMM management policy for different optimization targets. Moreover, the *sys-refs* tool can dynamically update the hotness collection interval parameter i in different times of page table or EPT walking according to the actual measured walking time and page hotness changes in each round. If the measured walking time is much shorter than collection interval and each page scan show small difference, then *sys-refs* can enlarge the scan interval, since the entire system is probably facing a low memory workload. Or if one-round measured walking time is very long and each page scan show very different hot page distribution on virtual addresses, then *sys-refs* can shrink the scan interval parameters to get more accurate reference hotness. The *sys-refs* tool can dynamically enlarge the epoch parameter e to reduce the runtime HMM management overhead when the memory-intensive workloads become idle and there are no changes in their memory reference distributions.

6 FLEXHM IMPLEMENTATION

The implementation of FlexHM is practical on the Linux system with the open source from 2018. Specifically, FlexHM kernel is modified on the Linux kernel, open-sourced at https://git.kernel.org/pub/scm/linux/kernel/git/wfg/linux.git. Besides, the prototype codes of FlexHM user-level manager are at https://github.com/intel/memory-optimizer.

The kernel implementation is practical for Optane memory and open-sourced with 1,000+ lines of code based on Linux, bringing no modification to hardware BIOS and user-space applications or virtual machines, thus, successfully supports full virtualization. Specifically, we now add Optane memory as a slow main memory type into the *e820* table and modify the *acpi/numa* module so we can memorize SMM devices into NUMA peer node and extend *SRAT* table [48]. Similar implementations have been later merged into Linux 5.4 in 2020. The two-level NUMA design is based on this transparent memory implementation. Also, we extend kernel *mm/pgtable* memory management to force *__get_free_pgtable_pages* to get pages from DRAM peer nodes when allocating new memory for new page table structures.

ALGORITHM 1: *sys-refs* Performance and Utilization Optimization

```
Input: The checking interval parameter $i$; maximum reference parameter $r$; Epoch $e$;
while true do
    for each p in processes do
        monitor addresses of $p$ into ranges;
    while $r > 0$ do
        create threads for hotness collection of ranges;
        record hotness collection of ranges;
        sleep until $i$;
    Analyze Page hotness of of ranges;
    Update $i, e$;
    for each p in processes do
        Adapt optimization policy;
        create threads for page replacement for $p$;
    Sleep until $e$;
```

We implement the collection of the memory access information in a new *kvm-ept-idle* kernel module. For native applications, we implement an interface named *mm_idle_walk_range*. The interface can scan the page table and checking the Access bit of all the pages according to the virtual address area of a single application. If the OS uses no **transparent huge page (THP)** and the application uses no huge pages, then all data r/w operations work on 4K pages, and the Access bits of the PTEs can represent the reference information. Otherwise, the Access bits of the 1 G or 2M PT entries can represent reference of data r/w operations. Since Linux kernel has implemented a *walk_page_range* function that can recursively walk the five-level PT of a process within its virtual address range, we export the symbol of *walk_page_range* to user space and reuse the function in *mm_idle_walk_range* so the interface can scan the PT and check the PFN Access bits of 4K page until it touches the **page table entry (PTE)** (or 1 G page until the **page upper directory (PUD)**, 2M page until the **page middle directory (PMD)**).

For guest machines, we first convert the HVA of anonymous pages to GFN and then convert the GFN to GPA with the translation information in EPT. Then, we implement an *ept_idle_walk_hva_range* interface to receive the anonymous page HVA as an input and check the reference information of guest pages in *kvm-ept-idle*. We implement a *ept_page_range*, which recursively walks the five-level EPT of a guest within its virtual address range until it checking the Access bit modification. Furthermore, we invalidate TLB when the module finishes each round of EPT walking to ensure that the hardware correctly clears and sets Access bits for super-hot pages. To reduce the time overhead of PT/EPT walking, we skip the whole 512 PTEs if we find that a high-level PMD is idle.

For flexible and efficient HMM management and performance optimization, we implemented a full functional user-level manager named Memory-optimizer with 3,000+ c++ codes. The project is open-sourced, and it provides runtime page management functions for optimizing one single workload (for example, applications or guest machines) or optimizing all processes on the FlexHM system. We provide several parameter choices for different page management policies. When following the aggressive policy, we choose 10 as the reference parameter r, 0.1 second as the interval i, and 10 seconds as an epoch e for the system only uses 4K pages without huge pages (also without THP configuration); for a system using 2M/1 G huge pages, we use 6 as the reference parameter r to avoid the write amplification issues where most pages show similar page hotness. Also, since the manager will promote all the identical hot SMM pages and demote all the identical cold DRAM pages analyzed in each epoch for aggressive page management, the percentage of need-movement pages is 100% equal to the identical hot/cold pages. When designing the conservative policy, we choose 20 references as *r* and 0.1 second as interval *i*, 20 seconds as an epoch *e* for the system only using 4K pages; For a system using 2M/1 G huge pages, we use 10 seconds as an epoch *e*. Also, when following conservative management policy, the manager only chooses at most 1/2 of the identical hot/cold pages to do page movement in each epoch.

7 EVALUATION

In the evaluation section, we make comprehensive experiments to evaluate how FlexHM can flexibly and efficiently manage the HMM resources for cloud workloads.

The evaluation contents help answer the following questions:

- What is the memory performance that the cloud workloads behave with when using HMM resources on FlexHM compared with all using DRAM or Optane memory? (Section 7.2)
- How efficient is the HMM utilization management and performance optimization of FlexHM? (Section 7.4)
- How does FlexHM provide performance optimization compared with previous HMM systems and the MM mode of Optane? (Section 7.3)
- Why should we support flexible management and optimization policy adaption in FlexHM (discussion between aggressive and conservative policy)? (Section 7.5)

7.1 Experiment Environment

In our experiments, we set up the hardware environment with configurations in Table 2. The SMM devices in all experiments are real Intel Optane memory. We compile the *kvm-ept-idle* kernel module on the host with Ubuntu 18.04 system using the Linux 4.20 kernel. To test the performance of native cloud processes, we directly run the memory-intensive benchmarks on the host server. To test the performane of cloud virtual machines, we run the benchmarks inside a *qemu-kvm* guest machine, which has an Ubuntu 18.04 system with the generic kernel, because FlexHM supports full virtualization and the guests need no kernel modification.

We introduce the micro benchmarks and application benchmarks in our evaluation experiments. First, we choose Sysbench [27] as our micro benchmark, which can generate very intensive high-parallel random read or write operations to occupy the peak bandwidth of heterogeneous memory hardware. The Sysbench can generate massive random read/write operations to the in-memory data, and the parameters of Sysbench memory tests are very friendly for us to evaluate the HMM management efficiency of FlexHM for differential cloud workloads. To be mentioned, Sysbench has a “memory-block-size” parameter, representing that Sysbench creates random memory access on a data array whose size is the “memory-block-size”; it also has a “memory-total-size” parameter, which can determine that the Sysbench generate memory-total-size/memory-block-size times of random read or write operations during the entire benchmark. The random memory references of our Sysbench benchmark follow an approximately generated Gaussian distribution, which ensures that 80% of random references lie in a 20% symmetric interval across the expectation of the data distribution.[1] We run groups of 1-thread and 32-thread Sysbench benchmarks in 32-core 68 G guest machines to represent the management for guest machine workloads, and we perform several groups of 1-thread and 32-thread native Sysbench processes on the server to demonstrate

[1]Sysbench uses a **central limit theorem (CLT)** [29] implementation to generate Gaussian distributions, and it can set a parameter in its implementation with a mathematical meaning similar to the standard deviation to determine the data distribution.

Table 2. The Hardware Configurations in the Evaluations

CPU	2 * Intel(R) Xeon(R) Platinum 8260L @ 2.40 GHz, 24 Cores * 2 Threads
DRAM	8 * 64 GB (total 512 GB)
SMM (NVM)	4 * Intel Optane DC Persistent Memory DIMMs 128 GB (total 512 GB)
Network	Intel Ethernet Converged Network Adapter XL710-QDA2 (40 Gbps, dual-port)

the management of native services or containers. The Sysbench are the random read/write experiments with a "global" random memory reference with a 64 G "memory-block-size" and a 2 T "memory-total-size."

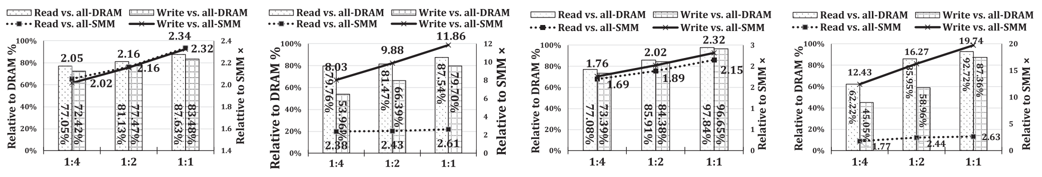
We choose Memcached [8] (in-memory database) (using YCSB framework [3] to generate Zipfian distribution workloads on Memcached) and Ligra [45] (graph computing) as representatives of the application benchmark.

Memcached is a high-performance, distributed in-memory caching system intended to speed up dynamic web applications to improve database performance. We set up the Memcached server in a 64 G guest as a cloud server benchmark, and the guest memory is allocated from NUMA peer nodes on server Socket 1. We bind YCSB multi-thread generators on Socket 0 and generate operations to the Memcached server through the 40 Gb dual-port network card. Specifically, each operation that YCSB clients generate works on a 100K K-V entry, and we store 500K entries in Memcached so all data occupied 52 G memory (containing metadata) in the 64 G guest. The random key selection of K-V operations to data follows a Zipfian distribution from 0 to 500K, which we manually set that 80% references lie in the top 20% data, with data hotness similar to the Sysbench settings.

Another application benchmark is a graph-computing framework Ligra [45]. Graph computing is a typical memory-intensive workload in cloud computing when a graph-computing framework loads a very big graph into memory and then runs computing algorithms. Data in a big graph may be fetched with an imbalanced data distribution when it is computed in different iterations. We, respectively, run **BFS (Breadth-First Search)**, **MIS (Maximal Independent Set)** [22], and **PR (PageRank)** [39] in Ligra, a Lightweight graph processing framework for shared memory. We run algorithms on random-generated graphs with 160M vertex in Ligra, and the graph consumes 62 G memory for in-memory computing in a 64 G guest.

7.2 Overall Performance with Different Ratios of HMM

7.2.1 Micro Benchmark – Sysbench. We configure the Sysbench micro benchmark with the random read/write experiments with a "global" random memory reference with a 64 G "memory-block-size" and a 2 TB "memory-total-size." We run several groups of 1-thread and 32-thread Sysbench experiments in a 32-core 68 G guest machine (providing some memory for the VM system processes) to represent the management for guest machine workloads and several groups of 1-thread and 32-thread native Sysbench processes on the server to demonstrate the management for native services or containers. We accurately fix different percentages of DRAM/SMM usage of Sysbench experiments in different groups, including the all-DRAM, DRAM: SMM = 1:1, 1:2, 1:4, and all-SMM (all Optane) configurations. To be mentioned, since we want to compare the performance of FlexHM with the baselines, we use *numactl* and *taskset* commands to bind the CPUs and memory of the guest machine or the native Sysbench process inside Socket 1 of the server. (Although the Memory-optimizer on FlexHM can easily set up HMM resource percentage



(a) 1-thread Sysbench in guest VMs. (b) 32-thread Sysbench in guest VMs. (c) 1-thread native Sysbench on host OS. (d) 32-thread native Sysbench on host OS.

Fig. 6. The overall Sysbench performance. Sysbench uses different percentages of DRAM and SMM (from 1:4 to 1:1), compared with two baselines where the workloads run in an all-DRAM or all-SMM system. (The performance criteria is the memory throughput, where higher is better).

of DRAM and Optane, the native Ubuntu system baseline needs to set up some fake NUMA nodes [26] and later use *numactl* to generate 1:1, 1:2, 1:4 ratio of HMM.)

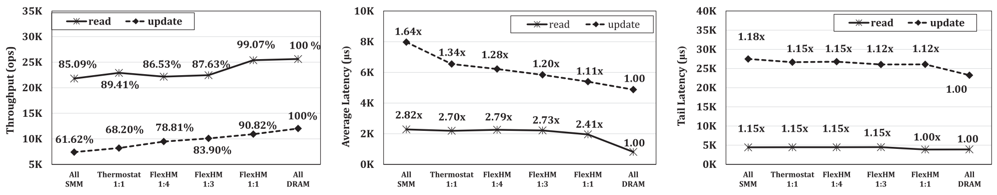
The results of guest Sysbench experiments are shown in Figure 6(b), and we choose the performance where Sysbench uses all DRAM resources as the baseline. From these results, we can find that the performance of FlexHM is promising with the help of efficient HMM management, although the throughput performance of the random read/write operations relatively decreases compared with the baseline when the Sysbench uses a higher percentage of Optane as memory resources, especially in the more-intensive 32-thread workload than the 1-thread Sysbench. Typically, a 32-thread Sysbench in the guest using a 1:1 percentage of DRAM and Optane can provide over 88% read throughput performance of the baseline and over 80% read throughput when it uses 1:4 percentage of DRAM and Optane. The write performance decreases more apparently when a heavy workload uses a higher percentage of slow memory Optane, as the 32-thread Sysbench can provide over than 80% write performance of the baseline with the 1:1 ratio and about 54% write performance with the 1:4 ratio.

Also, the performance results of the native Sysbench experiments on FlexHM are demonstrated in Figures 6(c) and 6(d). The native Sysbench performance also shows that native applications on FlexHM systems can achieve promising performance with different HMM ratios, for example, achieve over 93% read performance of the all-DRAM baseline in a 32-thread Sysbench with a 1:1 ratio configuration. These experiments show that memory workloads on FlexHM can get promising performance without serious performance degradation by choosing from different ratios of HMM if they want to save the expensive DRAM costs.

In summary, the Sysbench benchmarks demonstrate that the FlexHM system can provide efficient HMM management for guest or native workloads so these workloads can perform with promising performance.

7.2.2 Application Benchmark – Memcached. First, we set up the Memcached service and use YCSB workload generator to generate random memory access operations with the hardware and software configurations introduced in Section 7.1. We use the read-only and update-only workloads of YCSB to show the performance of FlexHM. The throughput results are shown in Figure 7(a), and the average latency results and tail latency results are in Figures 7(b) and 7(c).

We first give the throughout performance results of the Memcached using different percentages of HMM on FlexHM, compared with all-DRAM and all-SMM results. The all-DRAM baseline random read throughput performance reaches the upper bound of the network bandwidth (20 Gb/s) of the one port of our NIC. When Memcached uses HMM resources on FlexHM, it can also reach the upper bound of the network bandwidth when we increase the ratio of the test cases to DRAM:SMM = 1:1. When the Memcached uses a ratio less than 1:2, the random read



(a) Memcached Througput (OPS) (b) Average Latency (Lower is better) (c) 99th Latency (Lower is better)

Fig. 7. Memcached benchmark results (with different percentages of DRAM and SMM (from 1:4 to 1:1).

performance is around 85.9% to 87.64% of the all-DRAM baseline. The random update performance of Memcached is lower than the random read operations, because the update operations incur large numbers of random write to the memory, and the Optane memory has poor random write performance compared with DRAM. Specifically, when we let the Memcached use a 1:1 ratio of DRAM and Optane SMM, it can achieve over 90% of the all-DRAM baseline.

Besides the throughput performance, we concentrate on the average latency performance results. Compared with the all-DRAM baseline, the Memcached using HMM resources on FlexHM have over 2× average latency in the random read experiments. However, the average latency of random read operations stays in a stable status without latency performance degradation when the Memcached uses a higher SMM percentage than 1:1. In the random update experiments, the Memcached on FlexHM using 1:1 HMM resources can provide only 11% additional average latency overhead than the all-DRAM baseline. Moreover, the results prove that the FlexHM can significantly optimize the average latency performance over the all-SMM test case.

Last but not least, we discuss the tail latency performance results of the Memcached. Compared with the average latency, the tail latency of the random read is around 2× of its average latency, and the tail latency of the random update is over 5× of its average latency. However, the Memcached using different ratios of HMM resources do not have serious tail latency compared with the all-DRAM baselines; For example, the Memcached on FlexHM using 1:1 HMM resources can provide nearly no tail latency overhead over the all-DRAM baseline in the random read experiment and only 12% overhead in the random update experiment. This is because the tail latency of Optane SMM is not seriously worse than DRAM, so the tail latency performance of FlexHM memory subsystem will not become a system bottleneck.

In general, the in-memory database cloud applications represented by Memcached can get promising memory read or write performance because of the efficient HMM management of FlexHM. According to Reference [6], we assume that the Optane SMM and fast DRAM has a 1: 2.25 price ratio in public clouds. The in-memory database users can use 50% Optane SMM to replace DRAM for saving 28% memory prices with only 1% random read throughput performance overhead.

7.2.3 Application Benchmark – Ligra Graph Computing. Graph computing applications are popular in cloud services. The graph-computing frameworks load large-scaled data into the main memory and compute these data iteratively. Specifically, the computing algorithms fetch and use the graph data with imbalanced reference distribution, which can benefit from the FlexHM system when using HMM resources for better resource utilization and save the memory purchase prices.

We set up the Ligra graph-computing framework in a 64 GB guest machine, load a large graph file into memory, and run graph-computing algorithms on the in-memory graph data. When we load large graph files into memory, we ensure that these algorithms on these graph files can occupy over 62 GB of the entire 64 GB guest memory during the algorithm computing time. We separately configure the guest memory with all fast DRAM, DRAM: SMM = 1:1, 1:2,1:3, and all

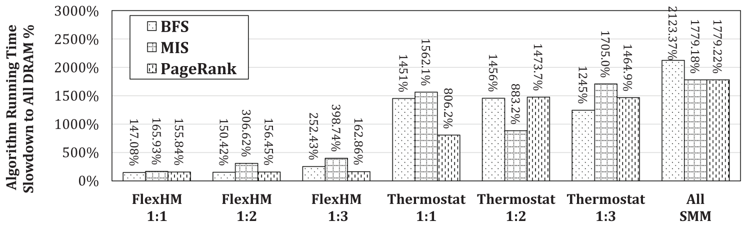
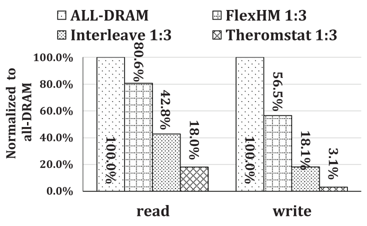
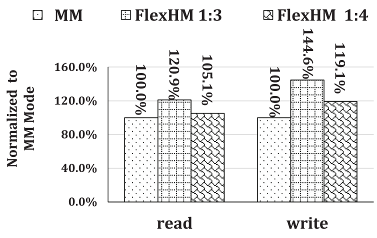


Fig. 8. The graph-computing benchmark on FlexHM using different HMM ratios.



(a) vs. interleave on Linux & Thermostat.



(b) vs. Memory Mode (MM).

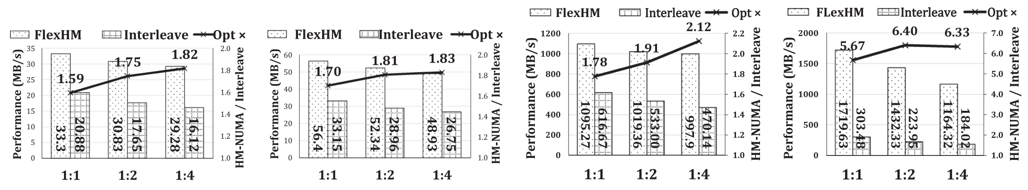
Fig. 9. Comparing FlexHM system with the mainstream HMM systems and solutions, where FlexHM can provide better performance optimization than Thermostat and MM mode. (The performance criteria is the memory throughput, where higher is better.)

Optane SMM case, and we choose the computing time of each algorithm as the performance criteria and use the relative performance slowdown to the all-DRAM configuration for better performance comparison.

We demonstrate the performance results of Ligra graph-computing algorithms with different HMM ratios in Figure 8. From the results, we can find that the all-SMM test cases show much more unsatisfactory relative performance to the all-DRAM baseline in Ligra than in the former Sysbench or Memcached experiments, for example, more than 17× performance degradation in all these three algorithms. Fortunately, when using 1:1, 1:2, and 1:3 HMM resources on FlexHM, the performance degradation to all-DRAM baselines can be significantly optimized, and using the 1:1 configuration can achieve a near-DRAM performance (up to 47% performance overhead). So, these results prove that FlexHM can provide appropriate HMM configuration choices for graph-computing applications with large memory usage to achieve acceptable performance and save the expensive memory costs. Among the three algorithms, the PageRank algorithm can get better performance on FlexHM, because the data reference of the PageRank reads and writes intensively on the hot data, and the BFS and MIS algorithms have the data reference distribution that is more similar to the uniform distributions.

7.3 Compared with Previous HMM Systems and MM Mode

We choose the Thermostat as the representative of the state-of-the-art transparent HMM systems, and we compare the performance optimization effect of FlexHM with Thermostat and the Memory Mode of Optane, with the results in Figure 9. We run the 32-thread Sysbench benchmark in the 32-VCPU and 34 GB memory guest machines and we make all the test VMs to use heterogeneous memory with DRAM: SMM = 1:3 configured by the Memory-optimizer manager, and Thermostat uses virtual NUMA of QEMU to set the HMM ratio. All the Sysbench experiments work on a 32 G “memory-block-size” and with a 2 TB “memory-total-size” so this workload will occupy over 32 G memory of the test VMs.



(a) Single-thread Sysbench Read Optimization. (b) Single-thread Sysbench Write Optimization. (c) 32-thread Sysbench Read Optimization. (d) 32-thread Sysbench Write Optimization.

Fig. 10. The throughput performance results and the optimization effects of FlexHM runtime management on different percentages of DRAM and SMM (comparing FlexHM with using NUMA interleave policy between DRAM and SMM peer nodes on the two-layer NUMA).

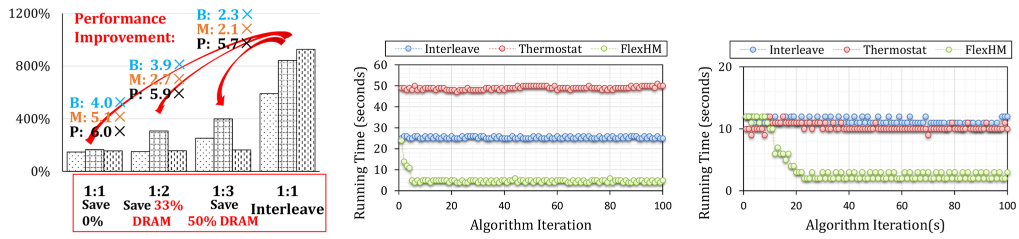
The comparison results with Thermostat are demonstrated in Figure 9(a), and we give the original read/write throughput normalized to an all-DRAM Sysbench baseline. FlexHM can provide 1.9× read performance optimization and 3.1× write performance optimization against the NUMA interleave policy. Also, we can see that the Thermostat does not perform well on the practical slow main memory resources. FlexHM can provide 2.4× read performance optimization and 5.8× write performance optimization against Thermostat with virtual NUMA support. The main reason is that Thermostat lets the kernel to do all the heterogeneous memory and it brings serious overhead to the entire system when a 32-thread Sysbench intensively uses the CPU and memory resources of the test guest machine. It cannot achieve even positive optimization for these intensive workloads. Also, we run the memcached and Ligra benchmarks on a Thermostat system using the same configuration as the FlexHM system used in Figures 7 and 8, and we analyze the performance data of the application benchmarks of Thermostat from these two figures. We can find that Thermostat cannot provide efficient HMM management for the application workloads; for example, it will incur at least 8.02× running time overhead to the Ligra graph-computing algorithms. These experiments show that FlexHM system can provide auspicious performance when using the practical Optane slow memory devices.

We also set up an HMM environment by configuring the Memory Mode of Optane. We change the DRAM DIMMs from 64 GB per blade to 16 GB so we have 128 GB DRAM and 512 GB Optane in the 2LM environment, so it is a 1:4 HMM environment. We run the 32-thread Sysbench benchmark in the 32-VCPU and 256 GB memory guest machines and bind the VM to one hardware socket, which has 64 GB DRAM and 256 GB Optane in total. We make all the experiments using a heterogeneous memory with DRAM: Optane = 1:3 or 1:4 on FlexHM to compare with the 2LM results. The results in Figure 9(b) show that FlexHM can achieve more efficient HMM management than two-level Memory Mode when using the same percentage of HMM memory (19.1% improvement in the 1:4 case) or using the same capacity of DRAM (44.6% improvement in the 1:3 case where FlexHM and 2LM both use 64 GB DRAM).

7.4 FlexHM Optimization Effect

We further design some experiments to demonstrate the efficiency of the FlexHM HMM management and explain why workloads on FlexHM can achieve the promising random read/write performance.

First, we demonstrate the HMM management effects for the Sysbench benchmarks in Figure 10. We give the original read/write throughput and give the relative performance between FlexHM and the interleave policy (where all the data locate evenly in the DRAM and SMM NUMA peer nodes according to the HMM resource ratios, and the system does not move hot/cold data between DRAM and SMM). In all the single-thread and 32-thread Sysbench experiments, our FlexHM system can



(a) Optimization effects over the NUMA interleave policy.

(b) How Thermostat and FlexHM optimize performance (90% memory usage).

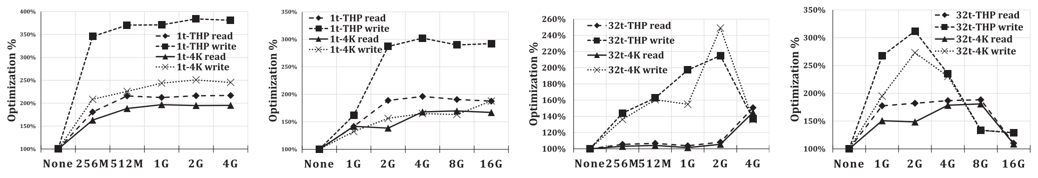
(c) How Thermostat and FlexHM optimize performance (50% memory usage).

Fig. 11. Overall performance optimization effect of FlexHM. (The performance criteria is the running time of the graph-computing algorithm, where lower is better.)

optimize the random read or write performance against the baseline without page management. Since the SMM (Optane memory) has less promising random write performance than the random read, the random write benchmarks meet a more severe performance bottleneck. Meanwhile, the runtime page management can achieve a better performance optimization effect for the random write benchmarks. Specifically, in single-thread Sysbench benchmarks, FlexHM HMM management can give an 82% performance optimization when we fix the ratio as 1:4 and also give over 50% performance optimization in other cases. Meanwhile, the runtime HMM management works better in 32-thread Sysbench experiments; for example, 5.67× write improvement in 1:1 case, 6.40× in 1:2 case, and 6.33× write improvement in 1:4 case. So, our runtime page management provides efficient management for workloads using fixed percentages of HMM on the FlexHM system.

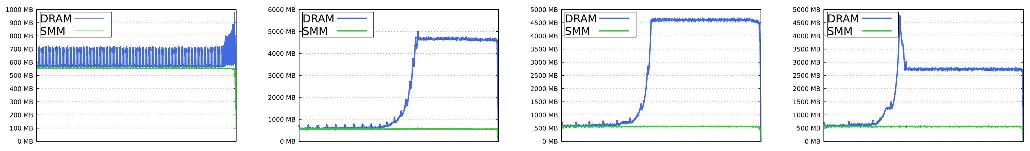
Finally, we also demonstrate the HMM management effects for the Ligra graph-computing benchmarks in Figure 11. In Figure 11(a), we compare the results of the benchmark using the 1:1, 1:2, and 1:3 HMM resources on FlexHM with using a 1:1 ratio under the interleave policy (where all the data locate evenly in the DRAM and SMM NUMA peer nodes according to the HMM resource ratios). From the results, we can find that our FlexHM gives up to 5.7× performance optimization for 1:1 baseline and saves 50% DRAM capacity with a 1:3 ratio between DRAM and SMM when we run PageRank algorithms. Also, BFS and MIS efficiently benefit from FlexHM runtime management. The significant performance improvement over the 1:1 interleave policy baseline proves that FlexHM can provide flexible HMM ratio choices for users to save memory prices with acceptable performance degradation because of its efficient HMM management mechanism.

We also demonstrate the optimization effect of FlexHM over the Thermostat by providing more experimental details in Figures 11(b) and 11(c). In these figures, we run on the Ligra benchmarks with the same PageRank algorithm on FlexHM, Thermostat, and a system using NUMA interleave, where all the three systems use 1:1 DRAM and SMM resources. FlexHM can efficiently select the hot/cold data and conduct the runtime HMM management to gradually reduce the running time of the PageRank algorithms during different algorithm iterations. However, similar to the results in Figure 11(a), Figure 11(b) proves that Thermostat has poor performance when the PageRank occupies over 93% of the total HMM resources, and its page replacement incurs more overhead than its performance optimization effect. When we use a small graph (Figure 11(c)), which the PageRank algorithm will only occupy 46% of the total HMM resources, Thermostat can gradually optimize the performance by replacing the hold/cold pages between DRAM and SMM. However, it cannot achieve the same optimization as FlexHM even when the system has adequate DRAM resources. FlexHM can achieve up to 11.9× optimization effect over Thermostat in Ligra graph-computing benchmarks.



(a) 1-thread 16G Sysbench. (b) 1-thread 64G Sysbench. (c) 32-thread 16G Sysbench. (d) 32-thread 64G Sysbench.

Fig. 12. FlexHM flexibly adopts different policies to provide optimal performance for different workloads. (The performance criteria is the memory throughput, where higher is better.)



(a) Baseline: No migration. (b) Max migration size: 1G. (c) Max migration size: 2G. (d) Max migration size: 4G.

Fig. 13. Emon online monitoring of Sysbench write experiments with different policy.

7.5 Significance of Flexible Management Polices

We do further experiments to prove the significance of flexibly supporting customized management policies in runtime page management of FlexHM. We individually run 1-thread and 32-thread Sysbench in 32-core 17 G guest machines with 16 G "memory-block-size," and in 32-core 68 G guest machines with 64 G "memory-block-size," and we fix the percentage as 1:3 and separately turn on/off THP on the host server. The Sysbench uses 2T as the "memory-total-size" to provide very stable long-time experiment results. In different groups, we change the sizes of maximum migration pages of one epoch from the 1/16 of the total DRAM sizes to 1/1 of that. The performance results are in Figure 12.

When the workloads are not intensive enough, we can choose a more aggressive policy for them, for example, the one-thread Sysbench experiments. Also, for read-intensive workloads, aggressive policy can be a better choice. However, the bottleneck of SMM (Optane) is write performance, so if aggressive migrating large capacity of pages, then the performance will now be optimized but be degraded because of the memory bandwidth bottleneck of SMM and the inaccuracy of hot page analysis and projection, for example, the 32-thread write experiments. So, choosing a conservative migration policy for write-intensive workloads will be better. The FlexHM can accurately collect and analyze the memory access patterns, and it can flexibly adopt different policies for different workloads because of our flexible memory management design.

Specifically, the Sysbench experiments are accompanied by an Emon [18] tool while monitoring and calculating the online memory throughput on all DRAM and Optane NVDIMM NUMA nodes. Among the different performance-tuning experiments, we demonstrate the memory access details of four experiments (No Migration, and setting the "max_migration_size" as 1 G, 2 G, 4 G) of the 32-thread 64 G Sysbench write benchmarks with the THP environment to help demonstrate the page management processes in Figure 13.

The fluctuations in Figures 13(b), 13(c), 13(d) are resulted by conducting page migration progresses. In Figures 13(b), 13(c), the hot pages are conservatively migrated by runtime HMM management with appropriate frequency and no overall performance seriously degrades by page migration. When most hot pages are in DRAM nodes, the Sysbench reaches the best performance on HMM systems. The migration progress also proves that the 2 G migration size is more appropriate for better optimization effect in this test cases. Moreover, there is a serious performance degradation in Figure 13(d), which proves that aggressive page migration is not suitable for write-intensive workloads with large memory capacity.

8 DISCUSSION

Hotness Collection Overhead: The overhead of page hotness collection is incurred by frequent PT/EPT walking in FlexHM kernel. A previous work [11] aims to give fine-grained reference information of all physical pages of a process, but the scan time becomes unacceptable when the workload is heavily memory-intensive. Compared with Reference [11], we collect coarse-grained page hotness by checking the page entries in PT/EPT, and we skip collection when we find an idle high-level page entry. We conduct experiments to test the page hotness collection time; the result is 2 s when walking 1 TB memory of 4K pages, compared with 15 s when using *idle_bitmap* in Reference [11], achieving about 7× speedup.

Page Migration Overhead: Page migration is designed to be triggered by the user-level manager for better flexibility and efficiency to adapt customized management policies. However, it may incur syscall overhead compared with kernel-level HMM management. Fortunately, we calculate the user-space execution time of Memory-optimizer is 4.18 seconds, and the kernel execution time is 68.41 seconds during the entire 32-thread Sysbench benchmark, which is only 6.12% time overhead incurred by our design.

Pros & Cons with Hardware Cache: The inclusive hardware cache implementation (such as MM mode of Optane) waste the capacity of DRAM, and the cache replacement policy may be suboptimal to the diverse cloud service workloads, while our FlexHM solution has the flexibility for different usage scenarios and even different policies for the same workload running at different times. Moreover, the hardware cache pays for additional expensive circuit cost, power, and complexity, while software approaches such as FlexHM can share the resource with the main system. Particularly, with the super-large capacity of SMM, the implementation complexity of hardware cache solution increases dramatically, making the solution impractical to implement and less efficient to improve performance. Also, the inclusive hardware cache solution must unconditionally load the new data from SMM to DRAM cache (and find a cache line to flush old data) to complete a CPU access to SMM, while FlexHM can leave the data in SMM if it is not very frequently referenced in future. So, FlexHM can save CPU cycles of cache flush and reload without stalling the host CPU execution.

9 CONCLUSION

In this article, we propose FlexHM, a practical HMM system where all guests, containers, and native applications can transparently and efficiently use heterogeneous memory with flexible and efficient HMM management and optimization. We extend the Linux for transparent HMM management, introduce the flexible and efficient runtime management of FlexHM with detailed design and implementation details, and many ideas have been merged into Linux kernel. Besides, the evaluations demonstrate FlexHM can efficiently improve performance or save DRAM capacity for different memory-intensive cloud workloads.

FlexHM is very user-friendly for researchers and developers to adapt more efficient page hotness analysis and management policies to meet more complicated workloads in the future. We also continuously develop a more efficient and more complicated purpose-oriented page management mechanism.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their constructive comments and feedback that greatly helped us improve the final article.

REFERENCES

[1] Neha Agarwal and Thomas F. Wenisch. 2017. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages & Operating Systems*.

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

[3] Joy Arulraj, Justin Levandoski, Umar Farooq Minhas, and Per-Ake Larson. 2018. BzTree: A high-performance latch-free range index for non-volatile memory. *Proc. VLDB Endow.* 11, 5 (2018), 553–565.

[4] Joy Arulraj and Andrew Pavlo. 2017. How to build a non-volatile memory database management system. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'17)*. ACM, New York, NY, 1753–1758.

[5] Adrian M. Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. 2010. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 385–395.

[6] Global Memory Procurement Corp. 2019. Server Memory Prices | Server RAM Module Price List - Memory. NET. Retrieved from https://memory.net/memory-prices/.

[7] Peter J. Denning. 1970. Virtual memory. *ACM Comput. Surv.* 2, 3 (1970), 153–189.

[8] Dormando. 2018. Memcached. Retrieved from http://memcached.org/.

[9] Thaleia Dimitra Doudali and Ada Gavrilovska. 2017. CoMerge: Toward efficient data placement in shared heterogeneous memory systems. In *Proceedings of the International Symposium on Memory Systems*. 251–261.

[10] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. 2016. Data tiering in heterogeneous memory systems. In *Proceedings of the 11th European Conference on Computer Systems*. ACM, 15.

[11] Brendan D. Gregg. 2018. Working Set Size Estimation. Retrieved from http://www.brendangregg.com/wss.html.

[12] J. Guo, Z. Chang, S. Wang, H. Ding, Y. Feng, L. Mao, and Y. Bao. 2019. Who limits the resource efficiency of my datacenter: An analysis of Alibaba datacenter traces. In *Proceedings of the IEEE/ACM 27th International Symposium on Quality of Service (IWQoS'19)*. 1–10.

[13] Manish Gupta, Vilas Sridharan, David Roberts, Andreas Prodromou, Ashish Venkat, Dean Tullsen, and Rajesh Gupta. 2018. Reliability-aware data placement for heterogeneous memory architecture. In *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA'18)*. IEEE, 583–595.

[14] Vishal Gupta, Min Lee, and Karsten Schwan. 2015. Heterovisor: Exploiting resource heterogeneity to enhance the elasticity of cloud platforms. *ACM SIGPLAN Not.* 50, 7 (2015), 79–92.

[15] S. Haitun. 1982. Stationary scientometric distributions: Part III. The role of the Zipf distribution. *Scientometrics* 4, 3 (1982), 181–194.

[16] Tom's Hardware. 2019. Intel Optane DIMM Pricing: $695 for 128GB, $2595 for 256GB, $7816 for 512GB (Update). Retrieved from https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html.

[17] A. Hassan, H. Vandierendonck, and D. S. Nikolopoulos. 2015. Energy-efficient hybrid DRAM/NVM main memory. In *Proceedings of the International Conference on Parallel Architecture and Compilation (PACT'15)*. 492–493.

[18] Intel. 2019. EMON User Guide. Retrieved from https://software.intel.com/sites/default/files/emon_user_guide_2019u3.pdf.

[19] Intel. 2019. Intel Optane DC Persistent Memory. Retrieved from https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[20] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic performance measurements of the Intel Optane DC persistent memory module. *CoRR* abs/1903.05714 (2019).

[21] Aamer Jaleel, Eric Borch, Malini Bhandaru, Simon C. Steely Jr, and Joel S. Emer. 2010. Achieving non-inclusive cache performance with inclusive caches: Temporal locality aware (TLA) cache management policies. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*.

[22] David S. Johnson, Mihalis Yannakakis, and Christos H. Papadimitriou. 1988. On generating all maximal independent sets. *Inform. Process. Lett.* 27, 3 (1988), 119–123.

[23] 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.

[24] Sudarsun Kannan, Yujie Ren, and Abhishek Bhattacharjee. 2021. KLOCs: Kernel-level object contexts for heterogeneous memory systems. In *Proceedings of the Architectural Support for Programming Languages and Operating Systems Conference (ASPLOS'21)*. Association for Computing Machinery, New York, NY, 65–78.

[25] A. Kemper and T. Neumann. 2011. HyPer: A hybrid OLTP OLAP main memory database system based on virtual memory snapshots. In *Proceedings of the IEEE 27th International Conference on Data Engineering*. 195–206.

[26] kernel.org. 2020. Fake NUMA for CPUSets. Retrieved from https://www.kernel.org/doc/html/latest/x86/x86_64/fake-numa-for-cpusets.html.

[27] Alexey Kopytov. 2004. SysBench: A system performance benchmark. Retrieved from http://sysbench.sourceforge.net/.

[28] Christoph Lameter et al. 2013. NUMA (non-uniform memory access): An overview. *ACM Queue* 11, 7 (2013), 40.

[29] Lucien Le Cam. 1986. The central limit theorem around 1935. *Statist. Sci.* (1986), 78–91.

[30] Soyoon Lee, Hyokyung Bahn, and Sam H. Noh. 2013. CLOCK-DWF: A write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures. *IEEE Trans. Comput.* 63, 9 (2013), 2187–2200.

[31] Yang Li, Saugata Ghose, Jongmoo Choi, Jin Sun, Hui Wang, and Onur Mutlu. 2017. Utility-based hybrid memory management. In *Proceedings of the IEEE International Conference on Cluster Computing (CLUSTER)*. 152–165. DOI: https://doi.org/10.1109/CLUSTER.2017.130

[32] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. 2017. Hardware/software cooperative caching for hybrid DRAM/NVM memory architectures. In *Proceedings of the International Conference on Supercomputing*. ACM.

[33] L. Liu, H. Yang, Y. Li, M. Xie, L. Li, and C. Wu. 2016. Memos: A full hierarchy hybrid memory management framework. In *Proceedings of the IEEE 34th International Conference on Computer Design (ICCD'16)*. 368–371. DOI: https://doi.org/10.1109/ICCD.2016.7753305

[34] Teng Ma, Mingxing Zhang, Kang Chen, Zhuo Song, Yongwei Wu, and Xuehai Qian. 2020. AsymNVM: An efficient framework for implementing persistent data structures on asymmetric NVM architecture. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*. 757–773.

[35] Frank McSherry, Michael Isard, and Derek G. Murray. 2015. Scalability! But at what COST. In *Proceedings of the 15th Workshop on Hot Topics in Operating Systems (HotOS XV)*.

[36] Carl Rod Nave. 1998. Gaussian Distribution Function. Retrieved from http://hyperphysics.phy-astr.gsu.edu/hbase/Math/gaufcn.html.

[37] Neha Agarwal. 2017. Thermostat Code for ASPLOS 2017 Paper. Retrieved from https://github.com/nehaag/thermostat_asplos_2017.git.

[38] M. Ben Olson, Tong Zhou, Michael R. Jantz, Kshitij A. Doshi, M. Graham Lopez, and Oscar Hernandez. 2018. MemBrain: Automated application guidance for hybrid memory systems. In *Proceedings of the IEEE International Conference on Networking, Architecture and Storage (NAS'18)*. IEEE, 1–10.

[39] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. 1999. *The PageRank Citation Ranking: Bringing Order to the Web.* Technical Report. Stanford InfoLab.

[40] H. Park, S. Yoo, and S. Lee. 2011. Power management of hybrid DRAM/PRAM-based main memory. In *Proceedings of the 48th ACM/EDAC/IEEE Design Automation Conference (DAC'11)*. 59–64.

[41] pmdk.io. 2020. Persistent Memory Development Kit. Retrieved from https://pmem.io/pmdk/.

[42] M. K. Qureshi and G. H. Loh. 2012. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-Tags with a simple and practical design. In *Proceedings of the 45th Annual IEEE/ACM International Symposium on Microarchitecture*. 235–246.

[43] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. 2011. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing (ICS'11)*. ACM, New York, NY, 85–95.

[44] 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 (SOSP'21)*. Association for Computing Machinery, New York, NY, 392–407.

[45] Julian Shun and Guy E. Blelloch. 2013. Ligra: A lightweight graph processing framework for shared memory. In *Proceedings of ACM SIGPLAN Notices*, Vol. 48. ACM, 135–146.

[46] J. Sim, A. R. Alameldeen, Z. Chishti, C. Wilkerson, and H. Kim. 2014. Transparent hardware management of stacked DRAM as part of memory. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. 13–24.

[47] Dinkar Sitaram and Geetha Manjunath. 2011. *Moving to the Cloud: Developing Apps in the New World of Cloud Computing*. Elsevier.

[48] ACPI Specification. 2017. IAdvanced Configuration and Power Interface Specification. Retrieved from https://uefi.org/sites/default/files/resources/ACPI_6_2.pdf.

[49] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. 2020. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'20)*. USENIX Association, 33–48.

[50] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing non-volatile memory in database systems. In *Proceedings of the International Conference on Management of Data*. ACM, 1541–1555.

[51] Vish Viswanathan. 2019. Intel Memory Latency Checker v3.7. Retrieved from https://software.intel.com/en-us/articles/intelr-memory-latency-checker.

[52] Stavros Volos, Javier Picorel, Babak Falsafi, and Boris Grot. 2014. BuMP: Bulk memory access prediction and streaming. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE, 545–557.

[53] Johannes Weiner, Niket Agarwal, Dan Schatzberg, Leon Yang, Hao Wang, Blaise Sanouillet, Bikash Sharma, Tejun Heo, Mayank Jain, Chunqiang Tang, and Dimitrios Skarlatos. 2022. TMO: Transparent memory offloading in datacenters. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'22)*. Association for Computing Machinery, New York, NY, 609–621.

[54] Kai Wu, Yingchao Huang, and Dong Li. 2017. Unimem: Runtime data management on non-volatile memory-based heterogeneous main memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'17)*. Association for Computing Machinery, New York, NY.

[55] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. 2019. Nimble page management for tiered memory systems. In *Proceedings of the 24th International Conference on Architectural Support for Programming Languages and Operating Systems*. 331–345.

[56] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST'20)*. 169–182.

[57] Ting Yao, Yiwen Zhang, Jiguang Wan, Qiu Cui, Liu Tang, Hong Jiang, Changsheng Xie, and Xubin He. 2020. MatrixKV: Reducing write stalls and write amplification in LSM-tree based KV stores with matrix container in NVM. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC'20)*. USENIX Association, 17–31.

[58] Shuai Zhang, Shufen Zhang, Xuebin Chen, and Xiuzhen Huo. 2010. Cloud computing research and development trend. In *Proceedings of the 2nd International Conference on Future Networks*. IEEE, 93–97.

Received 7 May 2022; revised 15 August 2022; accepted 20 September 2022