EasyChair Preprint

№ 1618

HiNUMA: NUMA-aware Data Placement and Migration in Hybrid Memory Systems

Zhuohui Duan, Haikun Liu, Xiaofei Liao, Hai Jin, Wenbin Jiang and Yu Zhang

October 9, 2019

HiNUMA: NUMA-aware Data Placement and Migration in Hybrid Memory Systems

Zhuohui Duan, Haikun Liu*, Xiaofei Liao, Hai Jin, Wenbin Jiang, Yu Zhang
National Engineering Research Center for Big Data Technology and System,
Services Computing Technology and System Lab/Cluster and Grid Computing Lab
School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
Email: {zhduan, hkliu, xfliao, hjin, wenbinjiang, zhyu}@hust.edu.cn

***Abstract*—*Non-uniform memory access* (NUMA) architectures feature asymmetrical memory access latencies on different CPU nodes. Hybrid memory systems composed of *non-volatile memory* (NVM) and DRAM further diversify memory access latencies due to the relatively large performance gap between NVM and DRAM. Traditional NUMA memory management policies fail to manage hybrid memories effectively and may even hurt application performance. In this paper, we present *HiNUMA*, a new NUMA abstraction for memory allocation and migration in hybrid memory systems. *HiNUMA* advocates NUMA topology-aware hybrid memory allocation policies for the initial data placement. *HiNUMA* also proposes a new NUMA balancing mechanism called *HANB* for memory migration at runtime. *HANB* considers both data access frequency and memory bandwidth utilization to reduce the cost of memory accesses in hybrid memory systems. We evaluate the performance of *HiNUMA* with several typical workloads. Experimental results show that *HiNUMA* can effectively utilize hybrid memories, and deliver much higher application performance than conventional NUMA memory management policies and other state-of-the-art work.**

***Index Terms*—Hybrid Memory, NUMA, Data Placement, Data Migration**

I. Introduction

The advent of NVM technologies has attracted increasing research interest in hybrid memory systems. A typical use of NVM is to organize it as an extension of DRAM in a single (flat) address space [1]. For example, Intel Optane DC Persistent Memory [2] provides an *App Direct Mode* to use both NVM and DRAM as main memory [3]. Due to the relatively large performance gap between NVM and DRAM [3], [4], data placement and page migration policies [5], [6], [7], [8], [9] have been widely studied to improve the performance of hybrid memory systems in recent years. Those studies all follow a basic principle that frequently-accessed (hot) data should be placed on fast memory (i.e., DRAM) to reduce the total memory access delay. Data placement policies usually rely on offline profiling techniques to characterize applications' memory access patterns, which are used to guide the initial memory allocation at the object/block level. In contrast, page migration policies usually exploit online page access monitoring techniques to predict memory behaviors in the future, and then migrate hot data from NVM to DRAM. However, online memory monitoring at the software layer can cause significant performance overhead, while hardware-assistant approaches usually require disruptive modifications of current hardware architectures.

Modern multicore systems are usually based on *Non-Uniform Memory Access* (NUMA) architectures. A NUMA system is composed of multiple CPU nodes. Each CPU node has memory attached to it, and accesses its local memory much faster than other CPU's memory. For the *App Direct Mode* of Intel Optane DC Persistent Memory, the OS kernel manages NVM and DRAM in different NUMA nodes [3]. Because NVM is slower than DRAM, especially for write operations [3], [4], hybrid memories would further intensify the feature of non-uniform memory access latencies in NUMA systems. The traditional NUMA-aware memory allocation and load balancing strategies focus on reducing the cost of inter-node NUMA communication and improving memory access locality. They are no long effective in hybrid memory systems because the access latency of local NVM may be even higher than that of remote DRAM. The difference of access latency between DRAM and NVM becomes a more important factor than the cost of inter-node NUMA communication. The traditional NUMA policy may even slow down the execution of applications in hybrid memory environments (see more details in Section II). In the following, we summarize the challenges of utilizing hybrid memories in NUMA architectures.

First, *the diverse hybrid memory access latencies in NUMA systems make the initial data placement become a complicated problem*. The NUMA programming library *libnuma* offers some sample data placement policies, such as local allocation and page interleaving. The local allocation policy only allocates memory from the node where the application process is executing. If the local memory is used up, it cannot use memory resource on remote nodes. This policy can achieve good performance because the local memory has the low access latency without inter-node communications, but cannot utilize memory resource on other NUMA nodes even there is sufficient memory in the machine. In contrast, the page interleaving policy allocates memory pages interleaved on all NUMA nodes in a round-robin manner. Although the memory access latency on the remote NUMA nodes becomes higher than the local accesses, it can fully utilize the

* Haikun Liu is the corresponding author.

This work is supported jointly by National Key Research and Development Program of China under grant No.2017YFB1001603, and National Natural Science Foundation of China (NSFC) under grants No.61672251, 61672250, 61732010, 61825202.

memory bandwidth of multiple NUMA nodes and balance the memory load. However, due to different performance features (latency, bandwidth) of DRAM and NVM, data distribution on DRAM and NVM is usually not the best placement using the traditional NUMA policies. For example, frequently-accessed (hot) data may be placed on a remote NUMA node attached to NVM DIMMs. Thus, data placement policies should take into account both performance characteristics of hybrid memories and inter-node communication cost.

Second, *the default NUMA memory migration policy is not effective in hybrid memory system*. Applications generally perform best when the task accesses memory on the local NUMA node. The *automatic NUMA balancing* (ANB) strategy always tries to move application data to the memory node closer to the tasks (threads or processes). However, in hybrid memory systems, the access latency of NVM on a local node may be even higher than that of DRAM on remote nodes. Without considering the inherent performance gap between DRAM and NVM, ANB may falsely migrate application data to a place that is even slower than its prior residence. Moreover, ANB would migrate a remote page to local memory once it is accessed twice, and thus may lead to unnecessary page migrations and a waste of memory bandwidth. The traditional ANB does not consider the memory heterogeneity and data access frequency, and thus is not effective in hybrid memory systems and may even hurt application performance.

In this paper, we present *HiNUMA*, an extension of traditional NUMA mechanism for hybrid memory architectures. *HiNUMA* provides new memory access interfaces to distinguish NVM from DRAM in different NUMA groups. We propose two NUMA topology-ware hybrid memory allocation strategies, i.e., *Low-latency* and *Hi-bandwidth* for latency-sensitive applications and bandwidth-sensitive applications, respectively. We also propose a new NUMA balancing mechanism called *HANB* for hybrid memory systems. It takes both data hotness and inter-node bandwidth utilization into account for page migrations. The proposed mechanisms are implemented in the *Operating System* (OS) layer, without modifications of hardware and applications.

We evaluate the performance of *HiNUMA* with several representative workloads. Experimental results show that *HiNUMA* can effectively utilize hybrid memories. *HiNUMA* data placement policies can improve application performance by up to 38.2% compared to the vanilla NUMA-interleave policy, and by up to 20% compared to the state-of-the-art *Bandwidth-aware Memory Placement and Migration* (BMPM) Policy [10]. *HiNUMA* page migration policies can further improve application performance by up to 33.2% and 18.9% compared to BMPM [10] and HeteroVisor [11], respectively.

The remainder of this paper is organized as follows. Section II introduces background and motivations of this paper. Section III describes the detailed design and implementation of *HiNUMA*. Section IV presents the experimental results. We discuss the related work in Section V and conclude in Section VI.

II. Background and Motivation

We first briefly introduce the background, and then present our experimental observations that motivate this work.

A. Hybrid Memory Emulation

Generally, NVM features higher memory density, much lower static power consumption than DRAM, at the expense of higher access latency and lower memory bandwidth. Currently, Intel Optane DC Persistent Memory is the only commercially available NVM DIMMs [2]. Its read latency is about 170-320ns, about 1.5-3 times higher than that of DRAM. Its read and write bandwidths are about 2.4–7.6GB/s and 0.5–2.3GB/s, respectively [3], [4]. Particularly, the bandwidth of random write is about 30 times lower than that of DRAM. Since the Optane DC persistent memory is not available before this work, we evaluate application performance in hybrid memory systems through an emulation approach. We use HME [12], [13], a lightweight hybrid memory emulator to emulate the performance characteristics of NVM by using DRAM.

B. Motivation

At first, we analyze the problems of using traditional NUMA management mechanisms in hybrid memory systems. The most interesting thing is the difference between the inherent memory access latency and the NUMA inter-node interconnection cost. We use Intel *Memory Latency Checker* (MLC) [14] to measure the NUMA interconnection cost in Intel Xeon servers. We find that the NUMA interconnection cost typically ranges from 53ns to 89ns. According to the NVM latencies described in Section II-A, the difference of access latency between NVM and DRAM is much larger than the cost of NUMA interconnection. The memory heterogeneity makes the data placement more complicate in NUMA systems.

We conduct experiments to verify the effectiveness of vanilla NUMA data placement policies in hybrid memory environments. HME is able to emulate the performance characteristics of many NVMs, such as STT-RAM, PCM, and ReRAM. Without loss of generality, we set the NVM read and write latencies to be twice and eight times of the DRAM, respectively, and limit the NVM bandwidth to be one half of the DRAM. The NUMA system contains only two memory nodes in which one is used to emulate the NVM node. There are generally four kinds of memories in NUMA systems: local DRAM, local NVM, remote DRAM, and remote NVM.

We evaluate the traditional NUMA policies using *convolution* and *YCSB* [15], two typical and popular multi-threaded applications for machine learning and big data processing, respectively. The *convolution* algorithm shows uniform memory accesses on all data, while *YCSB* shows very different memory access patterns in terms of access frequency (hotness).

We compare the following NUMA memory management policies: (1) Page interleaving (NUMA interleaving policy), pages are interleaved evenly on DRAM and NVM, i.e., NVM-to-DRAM ratio is 1:1. (2) NVM-to-DRAM ratio (1:4), data is placed on NVM and DRAM with a ratio of 1 to 4. More data placed on DRAM implies higher data access performance.

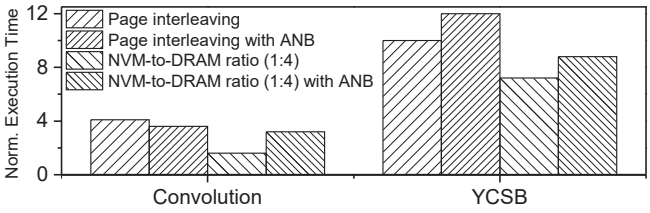

Fig. 1: The execution time of benchmarks using different placement policies, all normalized to the DRAM-only allocation policy

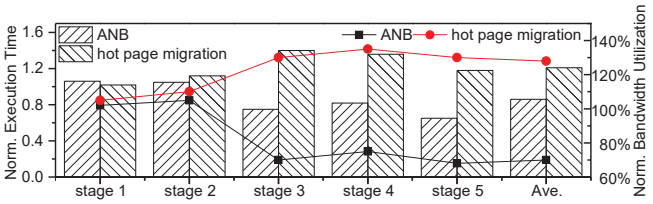

Fig. 2: The performance vs. bandwidth utilization of YCSB

Previous work [16] advocates to allocate DRAM preferentially, and migrate cold data to NVM only when DRAM is used up. We conservatively and tentatively set the NVM-to-DRAM ratio as 1:4. We also evaluate the policies (1) and (2) with automatic NUMA balancing (ANB) policy enabled. Figure 1 shows the experimental results, all normalized to a DRAM-only policy, in which all data are placed on DRAM nodes.

Observation 1: The NUMA page interleaving policy usually achieves sub-optimal application performance in hybrid memory systems. The page interleaving policy places data on NVM and DRAM evenly to balance the memory bandwidth of different nodes. However, in hybrid memory systems, this policy may hurt application performance. Generally, multi-threaded applications run much faster when more data is placed on DRAM. The policy "NVM-to-DRAM ratio (1:4)" leads to much less execution time than the page interleaving policy for both convolution and YCSB. Moreover, due to the difference of memory bandwidth between DRAM and NVM, the page interleaving policy is also sub-optimal from the perspective of memory bandwidth balancing [10].

Observation 2: The ANB policy is not effective for hybrid memory systems, and may even cause application performance degradation. ANB is a dynamic data placement strategy to improve the data access locality at runtime. The intent of ANB is to migrate application data to the memory node that the processes/threads are scheduled. However, Figure 1 shows that ANB even slows down the execution of applications. For the case of "NVM-to-DRAM ratio (1:4)", ANB significantly increases the execute time of *convolution* by a factor of 2. Because the local NVM is even slower than the remote DRAM, ANB falsely migrates a portion of recently-accessed application data from the DRAM node to the NVM node, and thus slows down the execution of *convolution*. As ANB usually changes data distribution in hybrid memory systems, it makes the initial data placement strategy useless at runtime.

Observation 3: The ANB policy can not fully utilize DRAM and NVM bandwidth. Figure 2 shows the performance and bandwidth utilization of YCSB using ANB and hot page migration schemes in five stages evenly divided by execution time. We use the "NVM-to-DRAM ratio (1:4)" as the initial data placement policy. The performance at different stages is also normalized to this static data placement policy. We find that ANB migrates a lot of data from high-bandwidth DRAM to low-bandwidth NVM (costly), and thus can not effectively utilize the high bandwidth of DRAM. The traditional ANB moves pages across different memory nodes based on data access recency. However, in hybrid memory systems, data migration between NVM and DRAM becomes more costly. The on-demand page migrations are usually unnecessary and waste memory bandwidth. A simple threshold-based hot page migration policy even achieves much higher performance than ANB. We also find that the hot page migration policy improves memory bandwidth utilization at the beginning of migration, and then the bandwidth utilization declines slowly. The reason is that the threshold-based hot page migration policy does not take memory bandwidth utilization into account. The data layout in the hybrid memory system can not maximize the bandwidth utilization of both DRAM and NVM.

Observation 4: Traditional task migration strategies for NUMA systems could not achieve the best performance in hybrid memory systems. To improve data locality in NUMA systems, a common approach is to move tasks to the data instead of moving data to the tasks. This approach is more efficient if a large amount of data should be moved. Because the NUMA intra-node communication cost is the major performance bottleneck in homogeneous memory systems, the task migration strategies can mitigate the overhead of data movement. However, in a hybrid memory environment, the difference of access latency between DRAM and NVM has a much larger impact on application performance than the inter-node communication cost of NUMA. Although task migration strategies can increase data locality, they increase a risk of accessing local yet slower NVM. Because accessing remote DRAM is still much faster than accessing local NVM, task migration strategies are no longer effective in hybrid memory architectures. The key problem of those strategies is that they do not change data distribution in hybrid memory systems. Overall, task migration strategies are not effective in hybrid memory architectures yet, and we should still rely on data migration to achieve higher memory access performance.

In summary, traditional NUMA memory allocation and ANB mechanisms are not yet effective in hybrid memory systems and may even hurt application performance. The above observations motivate us to develop new NUMA data placement mechanisms for hybrid memory systems.

III. Design and Implementation

In this Section, we present the implementation details of *HiNUMA*.

A. Physical Memory Abstraction

Figure 3 shows the hybrid memory architecture of *HiNUMA*. Linux kernel manages NUMA physical memory in several memory nodes, and each node comprises multiple

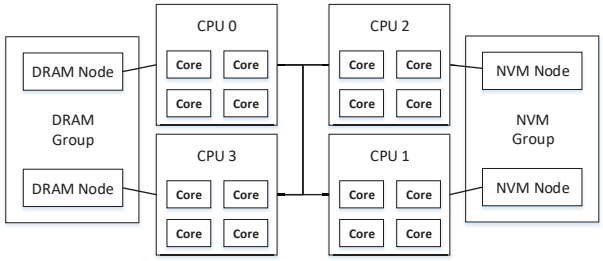

Fig. 3: Hybrid memory architecture of HiNUMA

DIMMs. To simplify hybrid memory management in Linux kernel, each NUMA node is composed of only one kind of memory media. We distinguish NVM nodes from DRAM nodes with an identifier, and cluster NVM and DRAM nodes in two NUMA memory groups separately. We use a data structure to maintain NUMA memory groups and the corresponding performance characteristics (bandwidth, latency, etc.). In this way, Linux kernel can manage different memory media at the granularity of NUMA nodes.

As the distance between different NUMA nodes also has non-trivial impact on remote memory access latencies, we measure NUMA inter-node communication cost through Intel *Memory Latency Checker* (MLC) [14]. We construct a weighted directed graph to present the communication cost between NUMA nodes. We record the NUMA topology to guide the memory allocation and migration. This NUMA topology-aware data placement can also reduce the communication cost across different NUMA nodes.

B. Memory Allocation Policies

Like the traditional NUMA, the memory allocation strategy in *HiNUMA* also aims to improve the performance of multi-threaded or multi-process applications in NUMA architectures. Because the OS kernel is usually unaware of the application semantic knowledge, it is hard to determine the initial data placement in hybrid memory systems for applications. *HiNUMA* provides two memory allocation policies as alternatives for programmers. Specifically, a *Low-latency* policy is designed to minimize the average memory access latency for latency-sensitive applications, while a *Hi-bandwidth* policy is offered to maximize both DRAM and NVM bandwidth utilization for bandwidth-sensitive applications. We provide several new interfaces in *libnuma* to perform our *HiNUMA* policies easily. For example, the system administrators can directly use "numactl – Low-latency/Hi-bandwith" to specify the memory allocation strategies for running latency-intensive and bandwidth-intensive applications, respectively. The application programmers do not need to modify their source codes at all.

At first, we present our *Low-latency* policy for memory latency-sensitive multi-threaded applications. Their execution times are mainly determined by transferring data from the memory subsystem to CPUs [7], [10]. Here, we consider a common case that a multi-threaded application's execution time is determined by the accumulated access delay on hybrid memories, and the CPU time consumed by all threads are equal. As DRAM and NVM nodes use separate memory controllers, the basic principle of *Low-latency* policy is to balance the total access delay on DRAM (t_{DRAM}) and NVM (t_{NVM}), so that the application execution time is minimized. That is

$$t_{NVM} = t_{DRAM} \tag{1}$$

Let N_{DRAM} and N_{NVM} denote the total number of accesses to DRAM and NVM, respectively, T_{DRAM} denote the DRAM read/write latency, T_{NVM_r} and T_{NVM_w} denote the asymmetric NVM read and write latencies, respectively. We assume that the portions of read and write operations on NVM are p and $1-p$, respectively. We assume the number of pages placed on NVM and DRAM nodes are n_{NVM} and n_{DRAM}, respectively. The total access delay on DRAM becomes

$$t_{DRAM} = N_{DRAM} * T_{DRAM} \tag{2}$$

The total access delay on NVM becomes

$$t_{NVM} = N_{NVM}(p * T_{NVM_r} + (1-p) * T_{NVM_w}) \tag{3}$$

Although some advanced technologies such as offline/online memory profiling can be used to analyze applications' memory access behaviors, in this paper, we assume OSes have no such priori knowledge for the initial memory allocation, and a better data placement mainly relies on dynamic page migration at runtime. As a result, we assume the memory accesses are uniformly distributed in all NVM and DRAM pages. According to Equation 1, 2, 3, we have the optimal NVM-to-DRAM ratio (R_{opt}) to place data on NVM and DRAM nodes.

$$R_{opt} = \frac{n_{NVM}}{n_{DRAM}} = \frac{T_{DRAM}}{p * T_{NVM_r} + (1-p) * T_{NVM_w}} \tag{4}$$

Second, we present the *Hi-bandwidth* policy designed for memory bandwidth-sensitive applications. Similar to the *Low-latency* policy, *Hi-bandwidth* aims to balance the total time spent in transferring data from DRAM (t_{DRAM}) and NVM (t_{NVM}) nodes to CPUs, so that the total application execution time is minimized. Equation 1 still works in this case. We assume that B_{DRAM} and B_{NVM} are bandwidths of DRAM and NVM, respectively. We can figure out the optimal NVM-to-DRAM ratio (R_{opt}) that data is placed on NVM and DRAM nodes using Equation 5.

$$R_{opt} = \frac{n_{NVM}}{n_{DRAM}} = \frac{B_{NVM}}{B_{DRAM}} \tag{5}$$

In *HiNUMA*, the memory allocation can be divided into three steps. First, we choose several NUMA nodes from two NUMA groups according to the NVM-to-DRAM ratio calculated by *HiNUMA*'s memory allocation policies. When the memory pages allocated from the DRAM group exceed a given threshold, the required memory page is allocated from the NVM group. Second, we rely on the NUMA topology to select the target NUMA nodes. We choose NUMA nodes with adequate free memory and the minimum NUMA interconnect cost as the target memory nodes. Finally, we allocate memory pages on DRAM nodes and NVM nodes according to the

NVM-to-DRAM ratio. When there is no free DRAM in the DRAM group, a NVM node is chosen for memory allocation. If all NUMA nodes have no free memory, we swap data in the local memory node to external storage to reclaim memory.

C. Page Access Counting

Hybrid memory systems often relies on page migration technologies to achieve better data access performance. To identify the hot data in NVM nodes, it is essential to monitor memory accesses at the page granularity. However, current x86 architectures do not provide hardware support for memory access counting. We develop a lightweight software approach to page access counting by using some reserved bits in *page table entries* (PTEs). To track memory references, we use BadgerTrap [17], a kernel extension to intercept TLB misses. Upon a page access, we mark (poison) its PTE by setting a reserved bit (the 51st bit), and then flush the PTE from the TLB. The consequent access to this page would trigger a TLB miss and a hardware page table walk. The poisoned PTE causes a protection fault which is intercepted by BadgerTrap. The fault handler adds one to the page's access counts, and then reset the reserved bit. In this way, we count BadgerTrap faults to estimate the number of page accesses for each page.

However, the selection of hot pages is still time-consuming. Previous work usually uses sorting [11] or threshold-based [16] schemes to classify pages as hot or cold. We shrink the scope of hot pages lookup through a sampling approach. As many applications exhibit good temporal/spatial locality of memory references, we use a multi-level queue as a buffer to record TLB misses intercepted by BadgerTrap. Each entry in the queue contains the index of a physical page and its access counts. We maintain the buffer with a pseudo *Least Recently Used* (LRU) algorithm. This buffer is able to record the access counts of most recently used pages. We periodically sort the pages in the buffer in ascending order of access counts to identify the hot pages, without sorting the whole physical memory pages. This approach can significantly reduce the cost of hot page identification by limiting the lookup scope within a LRU list.

Generally, a larger buffer can increase the accuracy of hot page identification, at the expense of more time spent in sorting. The length of queue can be determined by the maximum performance degradation specified by the users. We assume that the acceptable performance penalty due to hot page identification is $X\%$ at most, the total number of pages is N, the length of queue is L, and the average time spent in sorting and traversing per page is t_p. We deduce that L should not larger than $\frac{X}{100t_p} * N$.

Due to the impact of on-chip cache filtering, the number of TLB misses is not exactly equal to the number of page accesses. Our software-based page access counting mechanism actually utilizes the TLB misses as a proxy of LLC misses to estimate the number of page accesses [16]. Although this approach is not aware of how many accesses to a page are cache hits or misses, there exists a correlation between LLC miss rate and TLB miss rate. If the LLC miss rate is low, the number of TLB misses tracked by BadgerTrap is usually larger than the number of actual memory accesses. In contrast, a high LLC miss rate implies that the number of TLB misses tracked by BadgerTrap approximates to the number of actual memory accesses.

In order to calibrate the estimation of LLC misses and eliminate the impact on hot page detection, we adjust the sampling interval of hot page detection according to the change of LLC miss rate, as shown in Equation 6. When the LLC miss rate decreases, we increase the sampling interval to accumulate more accesses to the hot pages, and thus reduce the impact of cache filtering on hot page tracking. When the LLC miss rate becomes high, we reduce the sampling interval to detect the hot page more quickly.

$$\begin{aligned} \Delta LLCMiss &= \frac{LLCMiss_i - LLCMiss_{i-1}}{LLCMiss_{i-1}} \\ Interval_i &= Interval_{i-1} \times (1 - \Delta LLCMiss) \end{aligned} \tag{6}$$

D. Hybrid Memory-aware Automatic NUMA Balancing

In this section, we present hybrid memory-aware automatic NUMA balancing called *HANB*. Because the *HiNUMA* memory allocation strategies have no knowledge about the data hotness to direct data placement, and thus may not fully utilize the memory bandwidth of both DRAM nodes and NVM nodes at runtime. The goal of *HANB* is to further optimize data placement based on page migration during the execution of programs. We extend the traditional ANB to support topology-aware page migration within the same NUMA group, and hotness-aware page migration across different NUMA groups.

We implement *HANB* strategies by modifying the handler of NUMA hint page faults. When a page is migrated, we first check whether the page is moved within the same NUMA group or across different NUMA groups according to the page's physical address and the address mapping table of NUMA groups, and then choose the page migration policies. Upon a page migration, all read/write operations on the page are forbidden, and TLB entries corresponding to the page are invalidated. We exploit a copy-on-write method to migrate the memory page, and finally update the physical address in the page's PTE.

For page migration within the same NUMA group, the traditional ANB migrates data to the memory node where the processes/threads are currently scheduled in an on-demand manner. It is still effective for improving memory affinity. However, it does not consider the NUMA topology for intra-node data movement. We thus propose a NUMA topology-aware data migration scheme to reduce the communication overhead between different NUMA nodes. If the tasks' local memory node is under high memory pressure, we choose the closest memory node as the destination for page migration. If all NUMA nodes have no free page available, we reclaim memory in the local node to serve the upcoming memory requests. Those schemes are based on the NUMA topology stored in each NUMA group.

For page migration across NUMA groups, we further consider data hotness because the performance gap between NVM

and DRAM nodes becomes much larger than the inherent asymmetry of NUMA. As the local NVM node is even slower than the remote DRAM node, we need to migrate cold pages to NVM nodes, and migrate hot pages to DRAM nodes. Traditional page migration methods usually exchange roughly equal numbers of cold and hot pages between DRAM and NVM. They have not considered the memory bandwidth utilization. Thus, we propose asymmetrical page migration to take into account both memory latencies and bandwidth utilization. We use Intel VTune Performance Analyzer, *Performance Tuning Utility* (PTU) and *Event-Based Sampling* (EBS) to measure application bandwidth usage [18]. We use Equation 7 to evaluate the difference of memory bandwidth utilization between two sequential intervals.

$$\Delta Bandwidth = \frac{Bandwidth_i - Bandwidth_{i-1}}{Bandwidth_{i-1}} \quad (7)$$

To balance the bandwidth utilization between DRAM and NVM nodes, when we migrate a hot NVM page to DRAM nodes, N cold DRAM pages are migrated to NVM nodes at the same time. We use Equation 8 to determine the ratio (N) of DRAM pages to NVM pages. If $\Delta Bandwidth > 0$, we gradually increase the ratio N so that more cold DRAM pages are migrated to NVM nodes. Otherwise, less DRAM pages are migrated to NVM nodes. With this simple yet effective approach, the memory bandwidth utilization would be finally maximized with trivial fluctuations, and the value of N also becomes stable.

$$N_i = N_{i-1} + \lceil \Delta Bandwidth * N_{i-1} \rceil \quad (8)$$

IV. Evaluation

In this section, we evaluate *HiNUMA* page placement and migration strategies separately. We also compare *HiNUMA* with the state-of-the-art work [10] which we call BMPM, and HeteroVisor [11] which is implemented in a non-virtualized environment.

A. Experimental Setup

We use HME [13], [12] to emulate hybrid DRAM/NVM on NUMA-based servers, which are equipped with two-socket octa-core Intel Xeon E5-2650 2.0 GHz processors and 64 GB DRAM. We use 32 GB DRAM on one socket to emulate the NVM node. We set the NVM read and write latencies to be twice and eight times of the DRAM, respectively, and limit the NVM bandwidth to be one half of the DRAM [4], [3]. These settings are very like the performance characteristics of commercial Intel Optane DC DIMMs. The servers run CentOS 7 with a modified kernel 3.11.0 to support *HiNUMA*. We use Intel PMU tools to count memory accesses and bandwidth utilization.

We evaluate twelve multi-threaded workloads selected from PARSEC [19], SPLASH [20], BigDataBench [21], and YCSB [15]. Canneal, FFT, kmeans, ocean_cp, ocean_ncp, streamcluster and wordcount are memory bandwidth-sensitive workloads. Blackscholes, facesim, freqmine, raytrace, and swaptions are memory latency-sensitive workloads. We also evaluate the performance of *HiNUMA* using two key-value stores cassandra and redis. We use YCSB [22], which is a key-value store-based benchmark that mimics data access in web applications. Table I shows the configurations of six YCSB workloads in our experiments. Each workload performs 10K K-V operations on 50M items of 1 KB size.

TABLE I: YCSB Workload Configuration

Pattern	Read	Update	Scan	Insert	R&U
a	50%	50%	-	-	-
b	95%	5%	-	-	-
c	100%	-	-	-	-
d	95%	-	-	5%	-
e	-	-	95%	5%	-
f	50%	-	-	-	50%

We use the traditional NUMA placement and balancing policies as a baseline. We also compare HiNUMA with the state-of-the-art data placement and migration schemes BMPM [10] and HeteroVisor [11].

B. Experimental Results

Figure 4 shows the performance of workloads using different data placement policies in the hybrid memory system, all normalized to the baseline NUMA-local. The goal of NUMA-interleave is to balance memory bandwidth utilization among multiple memory nodes. However, it does not consider the difference of bandwidth between DRAM and NVM nodes, and thus can not fully utilize the DRAM bandwidth. BMBP [10] uses a memory bandwidth-aware page placement scheme, and achieves 22.9% performance improvement compared to the traditional NUMA policies for memory bandwidth-sensitive applications (from canneal to wordcount). Our *Hi-bandwidth* policy considers both NUMA topology and the asymmetrical performance of hybrid memories for data placement, and thus improves application performance by up to 36% and 19.4% compared to NUMA-interleave and BMBP, respectively. BMBP has very little impact on memory latency-sensitive applications (from Blackscholes to swaptions). In contrast, the *Low-latency* policy is specifically designed for latency-sensitive applications, and it is especially effective for those workloads. It reduces the execution time of latency-sensitive applications by 24.2% compared to BMPM. Overall, our *HiNUMA* policies can improve application performance by up to 38.2% and 20% compared to NUMA-interleave and BMPM, respectively.

We also evaluate the performance of HiNUMA with a large-scale benchmark YCSB. It uses two K-V stores, i.e., cassandra and redis. These workloads have very large memory footprints. Figure 5 shows the normalized response time of K-V operations, all normalized to the NUMA-interleave policy. We can find that BMPM and *Hi-bandwidth* are not effective for these memory latency-sensitive workloads. Because BMPM and *Hi-bandwidth* are designed particularly for memory bandwidth-sensitive applications. Compared to the NUMA-interleave strategy, BMPM only achieves a performance improvement of 10.1% on average, while *Low-latency* improves application performance by 35%. These results demonstrate the applicability of HiNUMA for different kinds of applications. It is

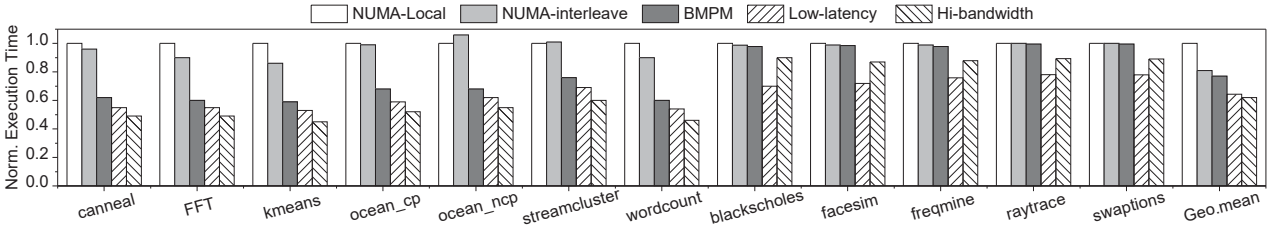

Fig. 4: Application performance for different data placement policies, all normalized to the NUMA-local policy

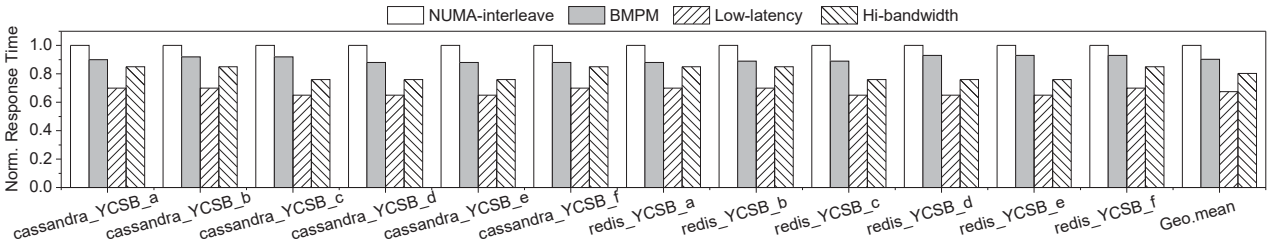

Fig. 5: Performance of YCSB benchmarks using different data placement policies, all normalized to the NUMA-interleave policy

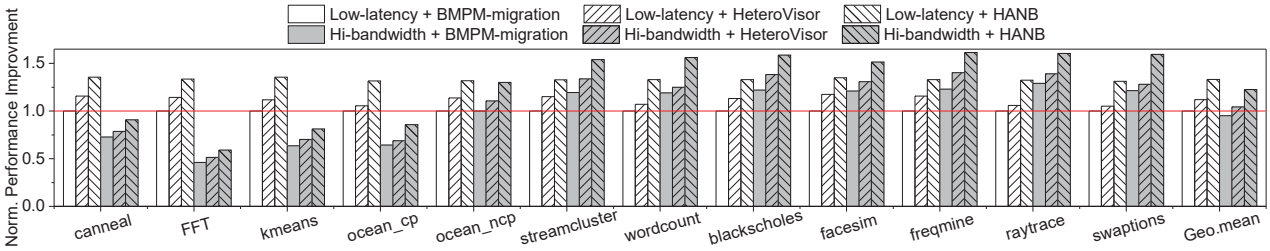

Fig. 6: Performance improvement of different page migration policies, all relative to the initial data placement policies

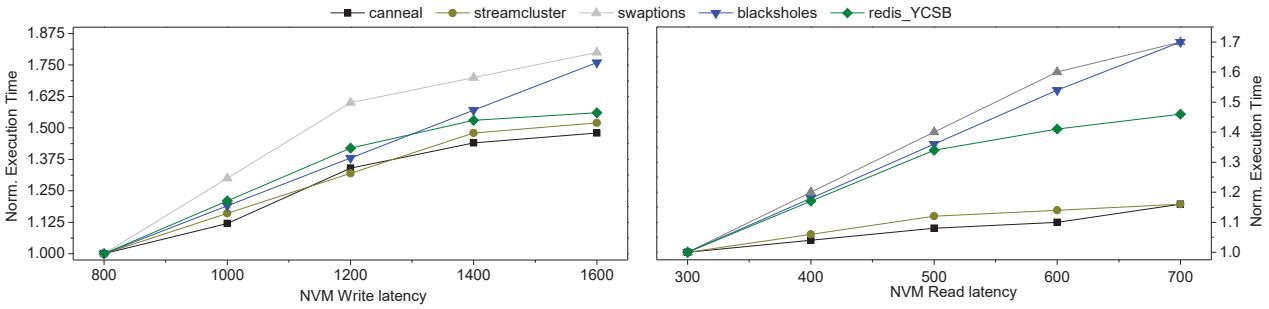

Fig. 7: Application execution time sensitive to NVM read/write latency

particularly applicable for memory latency-sensitive workloads that previous studies such as BMPM have not considered.

Figure 6 shows the performance gains of different page migration policies, all relative to the policy "*Low-latency* + BMPM-migration". We note that the traditional ANB even hurts application performance under a given data placement policy, so we only compare HANB with the state-of-the-art BMPM [10] and HeteroVisor [11]. Since "BMPM-migration" policy aims to maintain the initial data distribution on hybrid memories and does not consider the hotness of pages, it achieves only 2% performance improvement. The page migration policy in HeteroVisor is based on the page hotness. However, its simple page access counting and hot page detection strategies cause excessive performance overhead, and thus offset the performance gain from hot page migrations. In contrast, our HANB significantly reduces the overhead of hot page identification, and also maximizes the memory bandwidth utilization of applications. As a result, HANB can achieve up to 33.2% and 18.9% higher performance improvement compared to BMPM and HeteroVisor, respectively.

We have explored how different NVM access latencies can effect the application performance in *HiNUMA*. As shown in Figure 7, the application execution time is roughly linear to the increase of NVM read/write latencies. However, the execution time of bandwidth-sensitive applications such as canneal and streamcluster increases much slower than that of latency-sensitive applications such as swaptions, blacksholes, and YCSB. From another point of view, this experiment validates that the performance of latency-sensitive applications is more sensitive to NVM read/write latencies.

To evaluate how the application performance is affected by the scope of hot page lookup, we increase the size of the

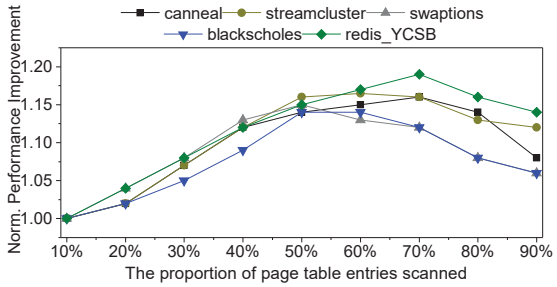

Fig. 8: Performance improvement sensitive to PTEs scanned

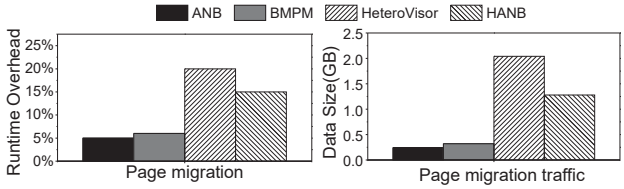

Fig. 9: YCSB performance overhead due to page migrations

buffer which is used to store the PTEs flushed from TLB, so that the scope of hot page lookup is extended gradually. Figure 8 shows the applications' performance varies with the buffer sizes. When the buffer size increases at the beginning, the extended searching space provides higher opportunities to find out more hot pages for migration, and thus improve the applications' performance. However, when the proportion of the scanned pages become larger than 50%, the application performance tends to slow down because the benefit of hot page migrations is offset by the increasing overhead of hot page lookup. As a result, we empirically set the buffer size to cover only one half of the applications' footprints.

Figure 9 shows the performance overhead due to hot page detection and migration for different policies running YCSB. ANB and BMPM all use an on-demand page migration scheme. They only lead to 0.24 GB and 0.32 GB data traffic due to page migrations, and thus the runtime overhead is only 5.0% and 5.6% of total execution time for ANB and BMPM, respectively. However, HeteroVisor and HANB all exchange the hot pages and cold pages between DRAM nodes and NVM nodes, and thus significantly increase page migration traffic to 2.04 GB and 1.28 GB, respectively. The runtime overhead due to page access counting and migration also increases to 20% and 15% for HeteroVisor and HANB, respectively. However, the runtime overhead can be offset by the performance gain from accessing hot data on the fast DRAM.

V. Related Work

There have been many studies on data placement mechanisms in NUMA systems with homogeneous memory [23], [24], [25]. Majo *et. al* [25] studied the impact of data placement on the performance of multi-threaded applications in the NUMA architectures. Carrefour [23] is a hardware-based memory placement mechanism for reducing NUMA traffic congestion. Gaud *et. al* [24] argue that superpages may be harmful in NUMA systems, and propose a new scheme to place superpages in NUMA nodes. Those previous work all assume that the memory nodes in NUMA systems are homogeneous. However, in a NUMA system with hybrid memories supported, the asymmetric memory access latencies of NUMA can be further amplified by the hybrid memories. They allow data placement on NUMA systems become more complicated. HiNUMA is specifically designed for data placement and migration in NUMA systems. We propose a NUMA topology and hybrid memory-aware data placement policy, and a page hotness-aware NUMA balancing mechanism to maximize the memory bandwidth utilization.

In recent years, there are also some studies on data placement and migration in hybrid memory systems [5], [6], [7], [8], [9], [26], [27], [28]. A lot of work propose hardware-assisted page migration policies, with TLB or memory controller monitoring page access patterns and migrating pages between hybrid memories [9], [29], [30]. Some other work propose OS-managed page migration in hybrid memory systems [31], [32]. However, the hot page detection usually introduces significant performance overhead that may offset the benefit of hot page migrations. Thermostat [16] exploits TLB miss rate as a proxy of cache miss rate to find out cold pages in DRAM, and migrate them to NVM. HeteroVisor [11] exploits hardware-assisted virtualization techniques to monitor hot pages in guest OSes, and migrate them to fast memory. HeteroOS [33] introduces a more efficient approach to hot page detection in virtualization environments to reduce the cost of page access counting. However, those hybrid memory management mechanisms are not particularly designed for NUMA systems, and only achieve sub-optimal performance improvement.

BMPM [10] perhaps is the most relevant work to HiNUMA. BMPM and HiNUMA both provide bandwidth-aware hybrid memory allocation and migration policies for NUMA systems. The major difference between HiNUMA and BMPM is that HiNUMA also provides a data placement policy specifically for latency-sensitive applications. Moreover, HiNUMA supports access-frequency based hot page detection and migration, while the page migration in BMPM is still based on traditional ANB. These optimizations in HiNUMA further improve application performance in NUMA-based hybrid memory systems.

VI. Conclusion

Traditional NUMA memory management policies fail to be effective in hybrid memory systems and may even cause application performance degradation. In this paper, we present NUMA-aware data placement mechanisms called *HiNUMA* for hybrid memory systems. *HiNUMA* exploits NUMA-aware memory allocation and asymmetric page migration mechanisms to improve the efficiency of using hybrid memories. Experimental results show that *HiNUMA* can improve application performance by up to 38.2% and 20% compared to the NUMA-interleave policy and the state-of-the-art BMPM [10], respectively. *HiNUMA* can also improve application performance by up to 33.2% and 18.9% compared to the ANB policy and the state-of-the-art HeteroVisor [11], respectively.

REFERENCES

[1] Sudarsun Kannan, Ada Gavrilovska, and Karsten Schwan. pVM: Persistent Virtual Memory for Efficient Capacity Scaling and Object Storage. In *Proceedings of the 11th European Conference on Computer Systems (EuroSys' 16)*, pages 13:1–13:16, 2016.

[2] Intel Optane DIMM. *https://www.tomshardware.com/reviews/intel-cascade-lake-xeon-optane,6061-3.html*.

[3] 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. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR*, abs/1903.05714, 2019.

[4] Intel Optane DIMM Pricing and Performance Details. *https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html*.

[5] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable High Performance Main Memory System Using Phase-change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA' 09)*, pages 24–33, 2009.

[6] HanBin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael A. Harding, and Onur Mutlu. Row Buffer Locality Aware Caching Policies for Hybrid Memories. In *Proceedings of the 30th International Conference on Computer Design (ICCD' 12)*, pages 337–344, 2012.

[7] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS' 15)*, pages 607–618, 2015.

[8] Hao Wang, Jie Zhang, Sharmila Shridhar, Gieseo Park, Myoungsoo Jung, and Nam Sung Kim. DUANG: Fast and lightweight page migration in asymmetric memory systems. In *Proceedings of the 2016 IEEE International Symposium on High Performance Computer Architecture (HPCA' 16)*, pages 481–493, 2016.

[9] Haikun Liu, Yujie Chen, Xiaofei Liao, Hai Jin, Bingsheng He, Long Zheng, and Rentong Guo. Hardware/Software Cooperative Caching for Hybrid DRAM/NVM Memory Architectures. In *Proceedings of the 2017 International Conference on Supercomputing (ICS' 17)*, pages 26:1–26:10, 2017.

[10] Seongdae Yu, Seongbeom Park, and Woongki Baek. Design and Implementation of Bandwidth-aware Memory Placement and Migration Policies for Heterogeneous Memory Systems. In *Proceedings of the 2017 International Conference on Supercomputing (ICS' 17)*, pages 18:1–18:10, 2017.

[11] Vishal Gupta, Min Lee, and Karsten Schwan. HeteroVisor: Exploiting Resource Heterogeneity to Enhance the Elasticity of Cloud Platforms. In *Proceedings of the 11th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE' 15)*, pages 79–92, 2015.

[12] HME. *https://github.com/CGCL-codes/HME*.

[13] Zhuohui Duan, Haikun Liu, Xiaofei Liao, and Hai Jin. HME: A lightweight emulator for hybrid memory. In *Proceedings of the 2018 Design, Automation Test in Europe Conference Exhibition (DATE' 18)*, pages 1375–1380, 2018.

[14] Intel Memory Latency Checker. *https://software.intel.com/enus/articles/intelr-memory-latency-checker*.

[15] PerfKit. *https://github.com/GoogleCloudPlatform/PerfKitBenchmarker*.

[16] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent Page Management for Two-tiered Main Memory. In *Proceedings of the 22Nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS' 17)*, pages 631–644, 2017.

[17] Jayneel Gandhi, Arkaprava Basu, Mark D. Hill, and Michael M. Swift. BadgerTrap: A Tool to Instrument x86-64 TLB Misses. *SIGARCH Comput. Archit. News*, pages 20–23, 2014.

[18] Detecting Memory Bandwidth Saturation in Threaded Applications. *https://software.intel.com/en-us/articles/detecting-memory-bandwidth-saturation-in-threaded-applications*.

[19] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT' 08)*, pages 72–81, 2008.

[20] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *Proceedings of the 22Nd Annual International Symposium on Computer Architecture (ISCA' 95)*, pages 24–36, 1995.

[21] Lei Wang, Jianfeng Zhan, Chunjie Luo, Yuqing Zhu, Qiang Yang, Yongqiang He, Wanling Gao, Zhen Jia, Yingjie Shi, Shujie Zhang, Hen Zheng, Chen Zheng, Gang Lu, Kent Zhan, Xiaona Li, and Bizhu Qiu. BigDataBench: A Big Data Benchmark Suite from Internet Services. In *Proceedings of the 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA' 14)*, pages 488–499, 2014.

[22] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC' 10)*, pages 143–154, 2010.

[23] Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS' 13)*, pages 381–394, 2013.

[24] Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quema. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC' 14)*, pages 231–242, 2014.

[25] Zoltan Majo and Thomas R Gross. (Mis) Understanding the NUMA Memory System Performance of Multithreaded Workloads. In *Proceedings of the 2013 IEEE International Symposium on Workload Characterization (IISWC' 13)*, pages 11–22, 2013.

[26] Chiachen Chou, Aamer Jaleel, and Moinuddin Qureshi. BATMAN: Techniques for Maximizing System Bandwidth of Memory Systems with Stacked-DRAM. In *Proceedings of the International Symposium on Memory Systems (MEMSYS'17)*, pages 268–280, 2017.

[27] Neha Agarwal, David Nellans, Mark Stephenson, Mike O'Connor, and Stephen W. Keckler. Page Placement Strategies for GPUs Within Heterogeneous Memory Systems. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, pages 607–618, 2015.

[28] Thaleia Dimitra Doudali, Sergey Blagodurov, Abhinav Vishnu, Sudhanva Gurumurthi, and Ada Gavrilovska. Kleio: A Hybrid Memory Page Scheduler with Machine Intelligence. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 37–48, 2019.

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

[30] Xiangyu Dong, Yuan Xie, Naveen Muralimanohar, and Norman P. Jouppi. Simple but effective heterogeneous main memory with on-chip memory controller support. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC' 10)*, pages 1–11, 2010.

[31] Wangyuan Zhang and Tao Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques (PACT' 09)*, pages 101–112, 2009.

[32] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data tiering in heterogeneous memory systems. In *Proceedings of the 11th ACM European Conference on Computer Systems (EuroSys' 16)*, 2016.

[33] Sudarsun Kannan, Ada Gavrilovska, Vishal Gupta, and Karsten Schwan. HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA' 17)*, pages 521–534, 2017.