Hierarchical Hybrid Memory Management in OS for Tiered Memory Systems

Lei Liu, Shengjie Yang, Lu Peng, and Xinyu Li

Abstract—The emerging hybrid DRAM-NVM architecture is challenging the existing memory management mechanism at the level of the architecture and operating system. In this paper, we introduce Memos, a memory management framework which can hierarchically schedule memory resources over the entire memory hierarchy including cache, channels, and main memory comprising DRAM and NVM simultaneously. Powered by our newly designed kernel-level monitoring module that samples the memory patterns by combining TLB monitoring with page walks, and page migration engine, Memos can dynamically optimize the data placement in the memory hierarchy in response to the memory access pattern, current resource utilization, and memory medium features. Our experimental results show that Memos can achieve high memory utilization, improving system throughput by around 20.0%; reduce the memory energy consumption by up to 82.5%; and improve the NVM lifetime by up to 34X.

Index Terms— Memory, DRAM, NVM, Operating System, Scheduling

1 Introduction

In the era of big data and cloud computing, applications have rapidly increasing memory footprints, energy consumption, and demand for throughput. To satisfy these requirements, it is critical to increase the memory capacity, reduce the memory access latency, and improve memory energy efficiency. Emerging Non-Volatile Memory (NVM) technologies (e.g., Intel/Micron's 3D XPoint promises 6TB of storage in a dual-socket server [22,23]) provide higher density and lower energy costs but suffer from relatively long write latency compared to DRAM. Thus, future systems will likely use hybrid DRAM-NVM systems [25,32,34,46], i.e., tiered memory systems, to take advantage of both the fast access speed of DRAM and the ultra-low idle-power, high density, as well as the non-volatility offered by NVM.

Conventionally, there are two different ways of organizing hybrid DRAM-NVM (Fast-Slow) main memory systems. The first option is to place different memories "vertically", i.e., using the faster DRAM as a cache (buffer) of the NVM. In this scheme, data movement between NVM and DRAM is controlled by dedicated hardware logic, which is transparent to Operating System (OS) and user applications [34,63,65]. Alternatively, DRAM and NVM can reside "horizontally" at the same level in the memory hierarchy [34,63,71], where software manages data placement and page migration. Compared to the first approach, the horizontal architecture presents more opportunities and challenges to OS designers [57,62,81,83].

The challenges in designing OS for a hybrid memory system lie in identifying performance-critical data that should be placed in the fast memory, and maximizing the utilization of the fast memory (DRAM), which is with limited capacity [44] and low utilization (as low as 31% in Google Data Centers [45]). Many studies have discussed this topic. In contrast, our work is based on the following key insights:

(1) While the previous approaches [53,87] can identify the memory access patterns, e.g., hot/cold memory pages/regions, for desktop-level applications by PTE (Page Table Entry) sampling, they are ineffective in the environments with the workloads such as Redis and Memcached, which have large memory footprints and diverse patterns of memory access (e.g., the hot regions with diverse write/read patterns may be randomly distributed in the large address space). Scanning the entire address space periodically for them to figure out the memory patterns can incur significant overheads. And, doing in this way is often insensitive to the memory pattern changes. Thus, it is necessary to combine event sampling and page-table walks to achieve both high accuracy and low overhead for monitoring the workloads with large and diverse memory footprints.

(2) Although NVM has a larger capacity compared to DRAM, it is hard to be fully exploited when added into the existing memory hierarchy. In order to maximize NVM/DRAM utilization, it is essential to consider cache activities at multiple hierarchical levels when placing memory pages; existing memory-hierarchy-blind approaches are sub-optimal [51,52].

(3) Page migration impacts the overall system performance, especially for the cloud computing systems with hybrid memories, where a large amount of data are frequently moved across memories due to memory pattern changes. We find the core reasons for the high overhead

Lei Liu (corresponding author), Shengjie Yang and Xinyu Li are within the Sys-Inventor Lab, SKLCA, ICT, CAS. Address: 0612J, No.6 Kexueyuan South Road Zhongguancun, Haidian District Beijing, China.
Email: lei.liu@zoho.com; {liulei2010, yangshengjie}@ict.ac.cn

Lu Peng is within the Division of Electrical & Computer Engineering Louisiana State University Baton Rouge, LA 70803 USA.
Email: lpeng@lsu.edu

are multi-fold: a) to ensure the consistency for migrated pages, the OS locks them during the migration; as a result, these pages cannot be accessed during this period, hurting performance; b) updating PTEs/mappings for the migrated pages can result in expensive context switches and TLB shootdown [28]; and c) current methods often use improper migration mechanisms for pages with different memory patterns. For example, an expensive CPU memory copy is inappropriate for cold data evicted from fast memory (DRAM), which is unlikely to be reused in the near future. Hence, we believe the data migration mechanism between DRAM and NVM should combine DMA and CPU-based approaches for the best efficiency depending on the memory access pattern, and this mechanism should also avoid the unnecessary lock usage and unnecessary migrations.

With these considerations, we introduce **Memos**, a memory management framework in the OS for horizontally integrated DRAM and NVM (i.e., Multi-Channel Horizontal Architecture (**MCHA**), where memory channels connect different types of memory). The critical design ideas and contributions of this paper are listed as follows:

(1) We design the first **full hierarchy memory management framework** in the OS, to hierarchically schedule cache, channels, and DRAM/NVM banks simultaneously. Our framework not only efficiently discerns and migrates hot pages to fast memory (DRAM) but also schedules the hotness according to the cache and bank associated mapping scheme, leaving the NVM cool while maximizing the utilization of the DRAM as well as the entire memory hierarchy. (Sec.4)

(2) We propose a **Hybrid Memory Monitoring mechanism (HyMM)**, an OS kernel-level online memory-profiling module. HyMM has three new features: 1) by extensively studying several memory traces, we find the most effective history window size for predicting the future page-level memory access patterns. Leveraging this knowledge, HyMM can effectively predict the future memory patterns. 2) We discover that, within a specific address range (i.e., a sub-memory region in the address space), memory pages exhibit similar write/read memory patterns. Thus, HyMM can use only one page as a sample representing the corresponding sub-region, and thus substantially reduce the sampling overhead. 3) HyMM is the first approach to combine TLB miss rate sampling along with *access/dirty_bit* sampling while adapting to different types of memory. In practice, HyMM obtains the page hotness, write/read patterns and the stream-like uses for applications with large memory footprint with lower overhead. (Sec.3)

(3) We devise a cost-effective **hybrid data migration engine**, which combines DMA and CPU-based page migration approaches, and allows for dynamically switching to the most appropriate mode. Specifically, we optimize the DMA with lock-less migration, more efficient memory page migration, and by saving CPU time for migration. (Sec.5)

(4) We design a **two-tiered Buddy System** in the OS kernel to support Memos allocating a specific page that corresponds to any cache slab, channel, DRAM/NVM banks in constant time. By modifying the Buddy System, DMA engine, and performance-monitoring module, we implement Memos in the Linux kernel. Moreover, we design an emulation platform for hybrid DRAM-NVM on a real machine with all DRAM DIMMs by using the channel-partitioning approach. (Sec.5)

We test Memos by employing Memcached [4], Redis [9], Aerospike [14], MySQLslap [16], and the benchmarks in SPECCPU 2006 [6]. The experimental results show that, on average, Memos on MCHA can improve memory utilization by 27.4~69.9%, and improve throughput by around 20.0% on average compared to previous approaches. Moreover, Memos can reduce memory energy consumption by up to 82.5%, and greatly improve the NVM lifetime.

2 BACKGROUND AND CHALLENGES

2.1 NVM is coming!

Driven by the growing demands for closing the gap between CPU and memory/storage, several NVM technologies emerge as DRAM alternatives (e.g., Phase Change based RAM [69,72,65]). These technologies offer the potential of building a low-cost hybrid main memory system that has a larger capacity, lower power consumption and operates at near-DRAM speed. Although these NVM technologies provide unprecedented options and tradeoffs, they do not aim to completely replace DRAM in the near future due to its longer write operation latency, higher dynamic energy consumption and even limited endurance. For example, PCM is expected to have 2X higher read latency, up to 5X write latency and 5X~10X lower bandwidth than DRAM [44,49,65,68,71]. Now, it is common wisdom to integrate NVM with DRAM to form a hybrid/tiered memory system (DRAM-NVM) to mitigate NVM's downsides while leveraging its low leakage and high-density benefits [34,65,78,83]. Yet, previous work still has room for improvement from the angle of OS. To achieve a desirable performance on hybrid memory systems, it would be ideal that the memory management mechanism in OS kernel could be aware of architecture features, memory characteristics and applications' memory access behaviors, and then guide and optimize the data mapping across entire memory hierarchy.

2.2 Data Replacement at Memory Hierarchy

2.2.1 Memory Bank Utilization

The main memory bank system is often the bottleneck of the overall throughput [65]. To improve bank-level parallelism, while conventional approaches use physical address interleaving [52,53] or an XOR scheme [84] to distribute physical pages across different banks as evenly as possible, they fail to consider the online memory behaviors. As a result, the hot (active) pages, which receive more memory accesses and have a higher impact on performance at runtime, are often distributed unevenly across banks. We evaluate the number of hot pages mapped to each bank, and compare the hot pages number

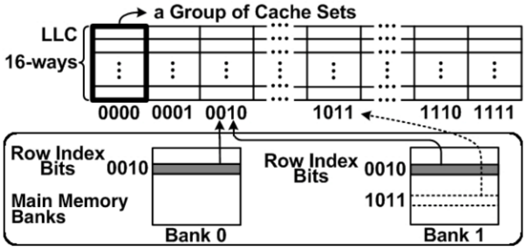


Fig.1. Cache and Memory bank associated address mapping.

of every bank with that of the coldest bank (i.e., the one with the least hot pages), to represent the hot page mapping imbalance across banks[1]. On average, for applications in SPECCPU 2006, the bank imbalance is 28.2%. GemsFDTD, as an example, exhibits a high bank imbalance, up to 52.5%, indicating some banks have significantly more hot pages than others. Things got even worse in some cloud cases. For example, we test Memcached and show its average bank imbalance is 66.0% at runtime. Our views indicate there are hot banks that suffer severe bank conflicts, harming the row-buffer locality, and we believe this brings significant performance loss in the context of long-running cloud computing environments.

Using NVM (e.g., PCM [44]), the bank imbalance can lead to more severe performance loss than on DRAM, as each bank conflict will bring additional cost than on DRAM, especially for cases with lots of write operations [44,65,69]. By rebalancing bank accesses via data migration, underutilized memory banks can share the responsibilities of these “hot” ones, and therefore the bank-level interferences can be greatly reduced. Reducing one conflict on an NVM bank can has an up to ~10X greater impact on performance than reducing the same conflict on DRAM. Moreover, as NVM system often has more memory banks than DRAM system, rebalancing NVM bank accesses can help to distribute these memory accesses across more banks, improving the overall bank-level parallelism and bandwidth.

2.2.2 Cache and Bank Associated Data Mapping

Bank-level balancing alone is not sufficient; in many cases, although memory accesses are nearly balanced across banks, the cache utilization is still very low. We further study the cache-bank associated address mapping in modern architecture [52,53,65] and find that there are some overlapped bits that index both of the row address in banks and the cache sets. Thus, “blindly” balancing bank utilization without taking into account the data block’s corresponding cache address (i.e., row address in a specific memory bank) will lead to cache conflicts. We show a typical example in Fig.1. Suppose there are two groups of pages (part of the row bits are 0010) residing in two banks: Bank 0 and Bank 1. Their data blocks will be mapped into the same cache set, denoted as 0010, which may cause cache conflicts. If we map a group of pages from one bank into a different row, denoted as 1011 in Bank1, the cache conflicts will be eliminated, while still maintaining bank balance.

This motivates us, to pursuit higher memory utilization, it is essential to consider cache activities hierarchically in tandem with bank allocation and mapping. For NVM, this consideration is especially meaningful, as cache misses to NVM have a higher cost than those for read. Therefore, (1) we should try to reduce the number of memory accesses that go to NVM by reducing the cache conflicts; (2) even on a specific NVM bank, we should map data blocks onto carefully selected rows to avoid cache conflicts, as some of the row bits also index the cache sets.

2.2.3 Memory Channel Effects

In a system using hybrid DRAM-NVM (e.g., MCHA), channel scheduling is crucial, as multiple channels connect different types of memory and provide different bandwidths. Mapping data that performs better on appropriate memory types will benefit the overall system performance. For example, if stream-like pages are mapped on an NVM channel, they will definitely consume the limited NVM bandwidth, leading to poor overall bandwidth utilization and performance. Since the limitation is the available bandwidth (NVM’s bandwidth is naturally lower than DRAM), we should have a new bandwidth scheduling policy for hybrid memory system. Furthermore, due to the longer latency and endurance for write operations on NVM, mapping data that is frequently rewritten into an NVM channel is not a good choice.

3 HyMM: Monitoring Memory Accesses Patterns on Hybrid Memory System

The primary purpose of our design is to keep the hot data especially for those with write-intensive access patterns within DRAM. The major challenges are identifying them with a low overhead, and then moving hot data between memories. This section highlights our design, i.e., HyMM, which is used for identifying the memory access patterns for workloads with high memory footprints.

3.1 Overview of HyMM

Previous studies [53,54,87] clear and check (i.e., sampling) the page *access_bit* in a PTE during continuous sampling passes to determine page hotness. HyMM (**Hy**brid **M**emory **M**onitor) employs this approach. Furthermore, to capture write and read patterns, HyMM monitors the *dirty_bit* (also in PTE) to capture page-level write/read behaviors. For a hot page, a *dirty_bit* of 0 indicates the page is being used for reading and 1 indicates that this page has been modified. HyMM monitors not just the access type but also the location of the access pages. By examining the values of the bank index bits in PFN (Page Frame No.) [51,52] and counting the hot pages assigned to each memory bank, HyMM can obtain the bank balance information. We carefully designed the core data structure for HyMM and use a page shadow array (each element is a raw byte) and bit manipulation to track the memory access patterns. For applications with a relatively small memory footprint (e.g., benchmarks in SPECCPU

[1] The memory system is with 64 128MB DRAM banks. And, the memory controller is with the widely used page-level interleaving scheme [51,65]. We have the similar stories on platforms used i7/i3/Xeon E5 series CPU with the XOR (Sandy/Ivy Bridge) and page-interleaving scheme.

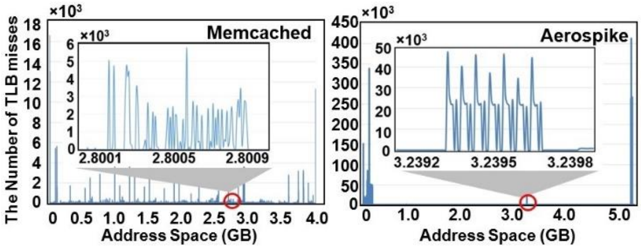


Fig.2. The number of TLB miss in two cloud applications' relative virtual address space with a snapshot. Hot regions are those with relative more TLB misses.

2006 that use below 1GB memory), sampling all pages in their memory space by monitoring the *access_bit* to discern the page-level memory patterns is practical [22,53,87]. However, for cloud applications, workloads often have a much larger memory footprint, thus sampling the entire memory address space to obtain the memory features (e.g., access frequency, hotness ranking) is not quite cost-effective and often leads to inaccurate sampling results [26]. The approach used in HyMM is described in Sec. 3.2.

3.2 Sampling Hot/Cold Regions

HyMM monitors the number of TLB misses instead of the *access_bit* to find Hot/Cold regions. HyMM is an OS module enhanced from [37] (the prior work can merely get the overall number of TLB misses for a specific app). Adding a shadow array in VMA, HyMM can get TLB misses for each page. Generally, cold pages that are rarely accessed have only a small number of TLB misses, while hot pages usually incur a large number of TLB misses, if they are frequently touched. In a program with a large working set, hot pages will be swapped in/out of the TLB repeatedly. By monitoring the number of TLB misses, we can obtain the distribution of TLB misses for the entire memory region and further divide the memory address space into many regions based on hot or cold patterns.

Pages that are "very hot" might be kept in the TLB and thus cause fewer TLB misses than cold pages do. We want to find out how many pages are in this category. In our experiments, the pages whose TLB miss count is below 10 in the sampling period (5s) are classified as cold pages[2], and we further check their *access_bit* to discern whether they are hot pages or not. Here "hot" refers to pages that are touched in 3 consecutive scan intervals (2s). The experiment reveals that this special category of pages can be ignored for two reasons. Firstly, the proportion of such special pages is quite small. Taking Memcached as an example, only 0.18% of the pages on average is actually hot, but is misclassified as cold. Secondly, the active content of these pages will likely reside in LLC for a long time without accesses to main memory.

We illustrate Memcached and Aerospike in Fig.2 to show the effectiveness of our approach. The memory pages in a range of specific memory addresses that exhibit high TLB misses are considered as hot pages. HyMM classifies pages with more TLB misses as hot. HyMM can find out the hot regions for these 2 cloud applications in their memory address spaces. Note that hot and cold pages are relative, because we want to select relatively hot pages for migrations (details are in section 4).

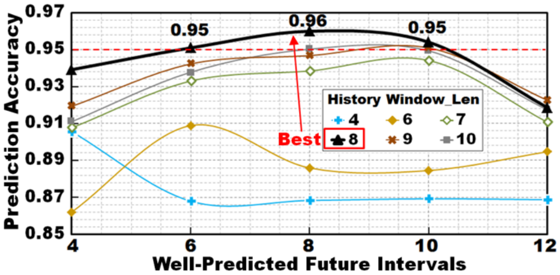


Fig.3. History window and prediction effectiveness. This figure summarizes extensive and diverse cases from all of the SPECCPU 2006 apps and cloud workloads averaged together.

3.3 Monitoring Write/Read Patterns

In the next step, we will analyze the pages in the hot region and examine their write and read patterns by examining the *dirty_bit* in their PTEs. Previous studies [26,35,58] show designs that predict future memory access patterns using recently monitored memory page-level access patterns. We are challenged by two questions: 1) how much history information should be used to capture valid memory patterns? And, 2) how long can will the memory patterns continue in the coming future? In order to address these questions, we analyze a large number of memory traces with records of write (i.e., *dirty_bit*=1) and read (i.e., *dirty_bit*=0) patterns (the sampling interval is 2s) from SPEC and cloud workloads, e.g., Memcached, Aerospike and etc., to reveal the predictive power of the latest history pattern records for finding the future duration of the current state, and the prediction accuracy for different history lengths. We denote the length of the history as window_len (each window has window_len total write/read records). A page is considered Write-Domain (WD) when at least half of the entries in the history window have non-zero dirty bits, otherwise it is Read-Domain (RD). As shown in Fig.3, in the case where the window_len is 8 (i.e., only the latest 8 consecutive history records are used in prediction), we can predict the memory access pattern with 96% accuracy on average. A short history (e.g., window_len is 4/6/7) does not have enough information to achieve an accurate prediction (with an accuracy below than 95%); on the other hand, an over-length history larger than 8 brings more noise data, thus hurting the prediction accuracy and increasing the sampling overheads. Therefore, we predict the future memory pattern using a history of length 8. According to the statistics shown in Fig.3, a memory pattern predicated using a history trace is expected to keep stable for 10 sampling intervals 95% of the time which is considered sufficiently long to avoid the "thrash-out" phenomena caused by miss-prediction and to avoid unnecessarily migrating pages.

Sub-region Sampling: Monitoring all pages at all times to collect 8 history records, even for a specific hot region, is quite expensive and impractical in a cloud computing ap-

[2] The constants in our design (current and following sec.) are empirical values based on the analyses of all programs from SPECCPU 2006 and cloud computing workloads. Thus, we conclude that our approach may work well in many real cases. These values can be adjusted as necessary in the conditions of extreme environment changes.

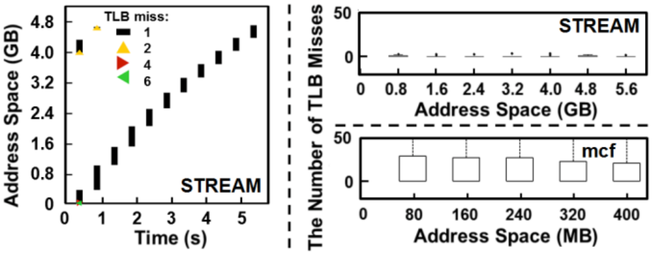
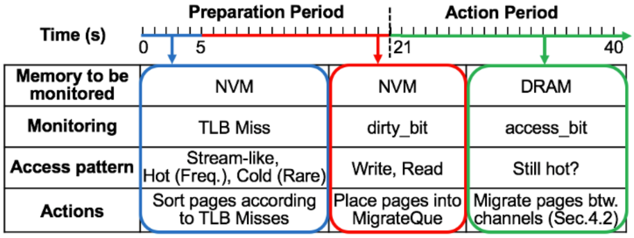


Fig.4. Stream vs. Non-stream (mcf) access pattern.



Memory to be monitored	NVM	NVM	DRAM
Monitoring	TLB Miss	dirty_bit	access_bit
Access pattern	Stream-like, Hot (Freq.), Cold (Rare)	Write, Read	Still hot?
Actions	Sort pages according to TLB Misses	Place pages into MigrateQue	Migrate pages btw. channels (Sec.4.2)

Fig.5. HyMM in a nutshell (a case).

plication due to its large memory footprint. To further reduce the overhead, we use a technique we call sub-region sampling. The technique has two steps: first, it divides the pages in a hot region into sub-regions; second, it uses a single page as the sample for each sub-region. As the first step, for a specific hot region, we scan and check all of the memory pages' *dirty_bits* to identify whether they are modified (write) or not (read). After a few intervals (three 2-second intervals by default), adjacent pages that show the same read/write dominance are grouped into sub-regions. Then, one page in each sub-region is randomly selected as a representative, and only this representative is monitored in the next 5 monitoring intervals. Doing so is reasonable, as our comprehensive experiments show that for a specific write or read region, 98.8% pages exhibit similar features on average. Discussed before, we collect 8 history records for these sampled pages to predict the future patterns for a specific sub-region according to the dominant patterns.

3.4 Detecting Pages with Stream Access Patterns

Besides the hotness and write/read patterns, monitoring TLB misses can easily identify stream-like memory usage. Fig.4 shows a clear streaming pattern in the address space of STREAM [13]. The left sub-figure shows the TLB miss frequency is 1 per page per half-second interval for the touched pages (indicated by black bars). With the time passing from left to right, the memory access exhibits a stable progression over the entire memory space from bottom to top. STREAM shows highly regular TLB miss counts for its pages, incurring an identical number of TLB miss per page. Fig.4 further shows that the distribution of the TLB miss count for pages in a range of specific memory addresses in box graph. The variation is minimal for STREAM (i.e., IQR[3] [11,12] is 0.28 on average). In contrast, for mcf, which does not have streaming access, the TLB miss count has considerable variations among the pages in a range of specific memory address (i.e., IQR is 24.4 on average), showing large variation in TLB miss counts. HyMM can easily meature and identify stream-like access pattern in practice.

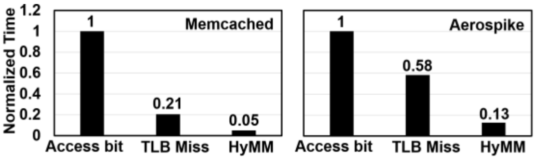


Fig.6. Sampling overhead comparisons.

3.5 Using HyMM on Hybrid Memory Systems

Fig.5 shows a case for the time overview of the HyMM design, including monitoring TLB misses, tracking the *access_bit* and *dirty_bit* in PTE. On hybrid memory system, application's data is initially stored in NVM. At first, HyMM monitors TLB misses for 5 seconds to discern hot and cold page regions, and then enables *dirty_bit* usage to identify the write/read patterns using sub-region sampling (eight 2-second sampling intervals by default). So far, in Fig.5, this period analyzes NVM data, selects pages whose TLB miss count is 10 or higher, and sorts them in the *MigrateQue* for later migration. We call it the **preparation period**. It lasts for 21 seconds (s). Note that to reduce the monitoring overhead in practice, HyMM stops monitoring for a specific page once its miss count reaches an upper bound (i.e., 200). The parameters can be tuned accordingly. An **action period** is launched immediately after the preparation and run in the next 20s. In this period, the queued pages are moved to DRAM and then tracked only by monitoring the *access_bit* with a low frequency. The alternation of preparation and action then repeats. These parameters can be adjusted[2].

We conduct experiments to show the advantages of HyMM using Memcached (4GB) and Aerospike (6GB); our experimental results are shown in Fig.6. While different approaches generate similar page hotness results (at most 3.9% and 5.2% differences for Memcached and Aerospike, respectively), the average sampling overhead of HyMM is around 1/5th that of the approaches that only use TLB misses counting [26,37] and 1/10th that of approaches that only use *access_bit* sampling [53,87]. Additionally, HyMM not only identifies the hot and cold pages with a lower sampling overhead but also can capture the read and write patterns at runtime via monitoring the *dirty_bit*. HyMM works well in practice as it introduces new monitoring methods while incorporating the advantages of previous methods. HyMM samples a small number of pages for these applications and the amortized overhead is quite low. More details are in Sec. 6.1.

4 Memos

4.1 Overview of Memos

This section details the Memos design. In Fig.7, with HyMM, Memos obtains the workloads' memory access patterns and then the Full Hierarchy Memory Management Framework leverages the information to schedule memory resources across the entire memory hierarchy (Sec.4.2). The framework has two key components: a two-tiered Buddy System that manages the NVM and DRAM in a hybrid way (Sec.5.1), and a highly efficient data migration engine (Sec.5.2). The physical address is split into

[3] The interquartile range (IQR) is a measure of variability in Box graph. Large IQR means large variation and unstable [11].

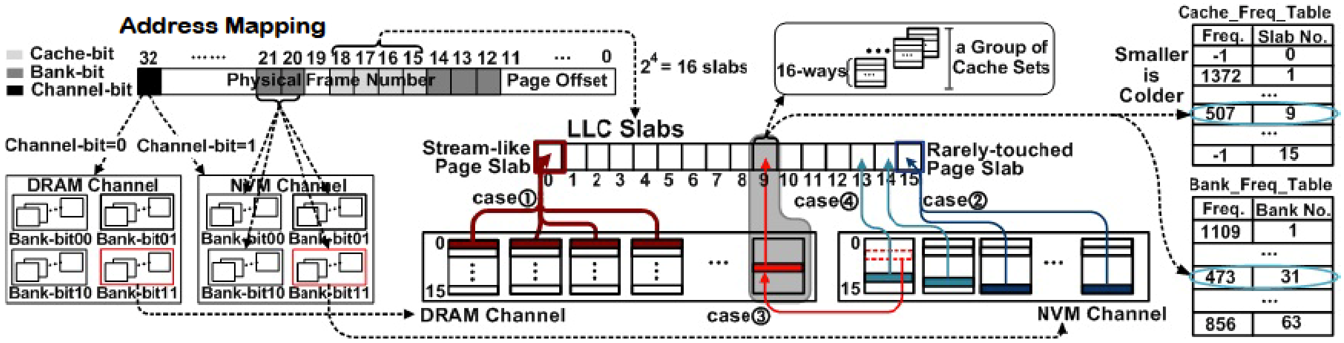


Fig.8. A case for address mapping of MCHA on a typical i7 machine [21,53] and four typical cases of Memos's working process.

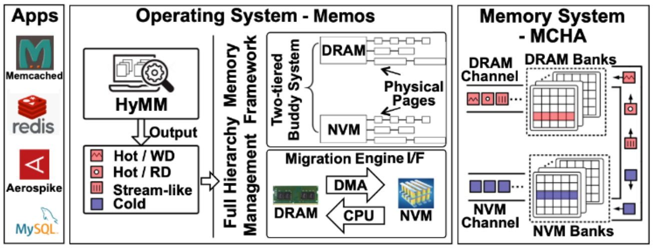


Fig.7. Overview of Memos (w/HyMM as a component).

DRAM and NVM segments. Moreover, memory pages will be migrated across channels when their patterns change.

4.2 Full Hierarchy Management Framework

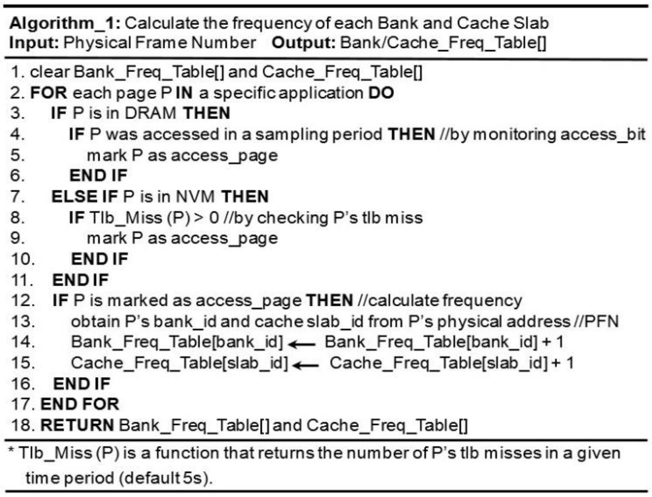
Fundamental Framework: We construct the memory management framework by leveraging the Page-Coloring approach [48,51,52,53]. As the address mapping illustrated in Fig.8, for a 4K-size page (0~11 bits denote the offset within the page) on a typical 64-bit architecture, bit 32 is used as the channel-bit. Therefore, by selecting a physical page with a specified value (0 or 1) at bit 32, we can control which channel, and consequently which memory segment (DRAM or NVM) to accommodate the page. For cache resource, on our experimental platform, each unique combination of cache index bit values (bits 15,16,17,18 in the page frame denote cache sets index, and also some rows in a memory bank, as shown in Fig.1), or cache-set color, dictates a slab (1/16 of the total 8MB LLC capacity) of LLC resource. Thus, we can adjust cache resource allocation by leveraging these bits. Moreover, for the memory bank resource in both NVM and DRAM channels, we monitor the bank utilization and enable the bank scheduling by using the bank index bits (bits 21,20,14,13 and 12 in Fig.8). Usually, bits 21,20 are used as a combination to uniquely dictate a group of 8 banks (called a bank-group color), and Memos can assign additional bank groups by using more than one bank-group colors. Leveraging these bits that indicate the different type of memory resources, Memos forms previously unused full hierarchy allocation approaches.

Channel Allocation: At the channel level, Memos selects a memory medium to map pages to according to their online write/read features and aims to maximize the overall bandwidth provided by both the DRAM and NVM channels used together. Memos attempts to place the hot pages (i.e., freq-touched/stream-like) onto DRAM, especially for those with WD features. WD pages are more likely to be moved to DRAM than RD pages, as on NVM the longer write operation latency incurs a more significant performance loss. Cold pages are kept in NVM to save energy and reserve DRAM space for stream-like, hot, and WD pages. At runtime, Memos monitors pages using HyMM and migrates them when the access pattern changes.

Bandwidth Scheduling: Memos' design goal is to maximize the combined bandwidth for both DRAM and NVM, which aims to avoid a significant decrease in NVM bandwidth that cannot be compensated by an increase in DRAM bandwidth. During an action period (as shown in Fig.5), Memos migrates up to 10,000 hot pages from the *MigrateQue* to DRAM and then monitors the resulting bandwidth on both the DRAM and NVM channels. When the DRAM bandwidth improvement is less than that gained in the previous epoch, Memos will reduce the number of the migrated pages (i.e., 1/5th that of the previous migration by default) at the next period. And, if Memos finds the NVM bandwidth decreases drastically more than the DRAM bandwidth increases, then it will stop migrating pages to DRAM for the upcoming period; moving more pages will not improve bandwidth utilization due to DRAM bandwidth is near saturated, but can further decrease NVM bandwidth, hurting the overall bandwidth. Finally, Memos may migrate some pages back to NVM to compensate for the NVM loss.

Cache and Bank Associated Allocation: Besides, Memos tries to hierarchically place data blocks according to the memory hierarchy details, thus avoiding memory conflicts and improving the memory utilization.

From the view of Memos, LLC is partitioned into 16 slabs (in Fig.8) by using LLC index bits (e.g., 15, 16, 17, 18 bits), and each slab denotes a group of LLC sets (i.e., 512 cache sets on our platform). Memos uses Algorithm_1 to record the utilization of each cache slab and memory bank in Cache/Bank_Frequency_Table. Each table is an array of integer-unsigned long pairs representing the id of bank or cache slab and the corresponding number of hot pages mapped to it, i.e., <Bank/Cache_Slab ID, Freq.>. The tables are in OS kernel. Our system has at most 16 cache slabs and 160 banks, thus the memory occupation is at most 2.1MB. The allocation process works as fellow:



Algorithm_1: Calculate the frequency of each Bank and Cache Slab
Input: Physical Frame Number **Output:** Bank/Cache_Freq_Table[]

```
clear Bank_Freq_Table[] and Cache_Freq_Table[]
FOR each page P IN a specific application DO
  IF P is in DRAM THEN
    IF P was accessed in a sampling period THEN //by monitoring access_bit
      mark P as access_page
    END IF
  ELSE IF P is in NVM THEN
    IF Tlb_Miss (P) > 0 //by checking P's tlb miss
      mark P as access_page
    END IF
  END IF
  IF P is marked as access_page THEN //calculate frequency
    obtain P's bank_id and cache slab_id from P's physical address //PFN
    Bank_Freq_Table[bank_id] ← Bank_Freq_Table[bank_id] + 1
    Cache_Freq_Table[slab_id] ← Cache_Freq_Table[slab_id] + 1
  END IF
END FOR
RETURN Bank_Freq_Table[] and Cache_Freq_Table[]
```

* Tlb_Miss (P) is a function that returns the number of P's tlb misses in a given time period (default 5s).

(1) By default, the LLC slabs are further divided into three segments, i.e., the slabs for stream-like, rarely-touched and the freq-touched pages. The design tries to map all of the stream-like pages into a small specific reserved slab (i.e., stream-like slab 0 in Fig.8), isolating them so the LLC can help to avoid interfering with other data, especially the data from the NVM channel. Meanwhile, all those rarely-touched pages are mapped together into another reserved slab (slab 15), as usually these pages consume very small cache capacity. Moreover, seen from Fig.8, larger LLC quotas, from slab 1~14, are used for freq-touched pages (i.e., freq-touched slabs).

(2) A PFN encodes both cache and bank access. Thus through iteratively recording pages' accessing times by monitoring TLB misses and *access_bit* in Algorithm_1, Memos records the corresponding cache slab and bank utilization in Cache/Bank_Freq_Table. As demonstrated in Fig.8, a lower frequency value means a lower utilization. When moving pages between NVM and DRAM, Memos will place them to the underutilized banks (i.e., these lower frequency banks in Bank_Freq_Table) for better bank parallelism, thus avoiding bank conflicts caused by blind mapping. Simultaneously, by placing pages to the rows whose index bits are associated with the low utilization cache slabs in Cache_Freq_Table, data can be loaded to these underutilized cache slabs. Doing so, as demonstrated in Fig.1, Memos can help to improve both cache and memory bank utilization while reducing the memory conflicts in those "hot" regions at memory hierarchy.

(3) If the associated memory regions (i.e., rows) in target bank in (2) are not free, Memos will try to select other underutilized slabs in Cache_Freq_Table accordingly, whose associated rows are still in this bank. If the memory banks in the DRAM channel cannot provide sufficient capacity, Memos will just migrate $N = \sum_{i=0}^{BANK-1}\sum_{j=0}^{ROW\_GROUP-1}(FMC_{ij}/Page\_Size)$ pages with higher migration priority (i.e., higher number of TLB misses) in *MigrateQue*, where FMCij denotes free memory capacity (FMC) of rows in jth row_group (corresponds to jth cache slab) within ith DRAM bank.

(4) Memos will enlarge the reserved slabs if the associated memory capacity cannot meet the special requirements (e.g., stream-like application with large memory footprint).

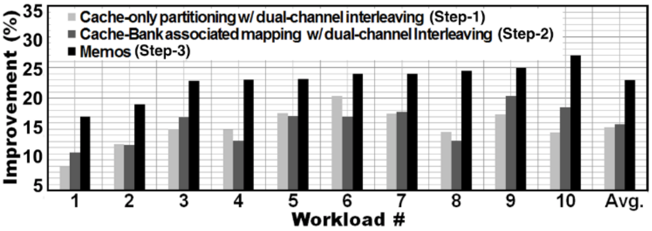


Fig.9. Overall performance throughput improvement.

(5) As shown in action period in Fig.5, the hot pages in DRAM are also tracked by monitoring the *access_bit* with the low sampling frequency. When Memos finds the pages are cold (*access_bit* is 0 in 2 consecutive 2-second intervals), it will migrate them to NVM, saving more space for hot pages. This is the Memos' reclamation process for the DRAM.

4.3 Cases on Overall System Working Process

To better understand the overall scheduling process, Fig.8 shows several typical cases. As illustrated in case①, pages that exhibit stream-like patterns are mapped into cache slab 0. Meanwhile, they are distributed into different memory banks for better bank-level parallelism (i.e., 15~18 bits, denoting both the row and cache set index, are with the value of 0). Similar things happen in cache slab 15, which is reserved for rarely-touched pages, especially for these pages with relatively lower access frequency that are kept in the NVM side (case②). For NVM, ideal bank-level parallelism can hide the expensive access latency (sometimes raised by memory interferences), as NVM systems are able to have more memory banks than DRAM. In case③, Memos migrates a hot/WD page from NVM to DRAM across channels. It first selects the coldest memory bank (for higher bank parallelism and overall utilization) and then maps the page to the row associated with the cache slab with the lowest utilization (slab 9 in Cache_Freq_Table). Moreover, the NVM channel can also provide bandwidth. As shown in case④, data blocks from RD pages can be loaded to cache directly through the NVM channel. Note that for the rest of the RD pages with the stream-like feature in NVM, Memos will also map them to the reserved slab 0. Even for a specific channel, hot pages are migrated from highly utilized banks to lower ones to balance the overall utilization. On average, our full hierarchy memory management framework outperforms newly proposed policies by around 10.0% (details are in Section 4.4).

4.4 Effectiveness on Throughput and QoS on Real System

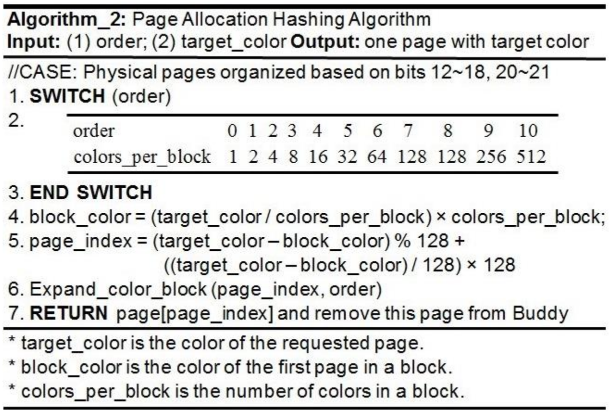
This section has 3-steps experiments on the platform with an Intel i7-series/2.8Ghz CPU to show the effectiveness of our above-mentioned new approach. We employ 10 workloads, and each of them consists of 4~8 applications from SPECCPU 2006 (refer to appendix; at least one application has the stream-like pattern). We use DRAM in both of the DRAM and NVM channels. In Step-1, we only enable cache partitioning [48] in the LLC to migrate the memory interferences. In Step-2, we enable cache and bank associated mapping but without the memory channel-level consideration [53], i.e., mapping memory pages evenly across two channels even for the pages with highly aggressive stream-like patterns. Step-1 and Step-2 are with dual-channel interleaving scheme. In Step-3, i.e., Memos, we enable our newly proposed full hierarchical approach, which not only has the cache and bank associated mapping but also schedules memory pages across channels, such as confining pages with stream-like patterns into a specific channel for reducing the bandwidth contention for the NVM channel. The general baseline is the unmodified Linux kernel. Fig.9 illustrates the experimental results. Step-3 outperforms Step-2 by 7.2% (up to 11.0%) on average. Step-3 achieves an average 23% performance gain over the baseline. Moreover, Step-2 outperforms the Step-1, as it can improve the memory utilization at LLC and bank simultaneously as discussed in Sec.4.2 and Fig.8. Note that, with the channel-interleaving scheme, neither of the approaches of cache and bank associated mapping (Step-2) and the cache-only partitioning (Step-1) can constrain pages with aggressive accesses into a specific channel, thus causing all-to-all memory interferences in all channels. Our new approach can reduce these interferences. As the program behavior varies, the magnitude of the improvement also varies, from the lowest 17.0% to the highest 28.1%. The improvement is strongly correlated with the number of stream accesses.

Moreover, our results show that bank imbalance is reduced by 60.1~69.9%, and the cache misses are reduced by 27.4% on average across both channels, with a 42.1~50.0% reduction on the NVM channel. These benefits also contribute to the improvement of QoS (indicated by Max Slowdown [53,65,66]) by 23.6% on average. Take workload 9 as an example. Memos improves QoS by 34.1%, while the other two schemes (i.e., Step-1, Step-2) improve QoS by 13.2% and 19.4%. Memos outperforms them by 20.9% and 14.7%, respectively. For the bandwidth on MCHA, as mentioned before, Memos does not merely maximize the DRAM bandwidth while hurting the bandwidth in other channels. In our experiments, Memos tries to maximize the combined bandwidth of all memory channels, thus improving the overall bandwidth on MCHA by 24.6% on average. To sum up, the experiment shows our memory framework is effective.

5 Kernel Modules and Emulation Platform

5.1 Two-tiered Buddy System in Memos

To support Memos' full hierarchy memory framework, we are the first in designing a two-tiered Buddy System by extending Page-Coloring to reorganize the free pages in Linux kernel with the channel, bank and cache bits in PFN simultaneously. With the channel bit, we reorganize all physical pages into two sub-buddies logically, one for pages in NVM and the other for pages in DRAM. In each sub-buddy, we can still use other index bits (cache/bank bits) to allocate resources. By doing so, Memos tags resources according to hierarchy details, material features, and therefore can efficiently allocate them accordingly. Nine bits (21,20,18~12 bits) in PFN form a set of 2^9=512 colors. Memos uses Algorithm_2 which works as a hashing index to allocate pages corresponding to any cache slabs, channel and NVM/DRAM banks with O(1) time consumption, even for the cloud applications. The primary memory allocation interface in the kernel is *alloc_resource (int channel_id, int cache_slab, int bank_id)*, which is used to obtain a group of memory resources. By adding resource control parameters into *Task_Struct* (denotes *Process* in Linux kernel), users can leverage this interface to map the applications' data heap according to their requirements.



Algorithm_2: Page Allocation Hashing Algorithm
Input: (1) order; (2) target_color **Output:** one page with target color

```
//CASE: Physical pages organized based on bits 12~18, 20~21
1. SWITCH (order)
2.      order              0  1  2  3  4   5   6   7    8    9    10
        colors_per_block   1  2  4  8  16  32  64  128  128  256  512
3. END SWITCH
4. block_color = (target_color / colors_per_block) × colors_per_block;
5. page_index = (target_color – block_color) % 128 +
                  ((target_color – block_color) / 128) × 128
6. Expand_color_block (page_index, order)
7. RETURN page[page_index] and remove this page from Buddy
```

* target_color is the color of the requested page.
* block_color is the color of the first page in a block.
* colors_per_block is the number of colors in a block.

5.2 Data Migration Engine

To improve the efficiency of data migration, we design and implement a new engine, which combines CPU and DMA-based page migration. First, using the page copy primitive in OS kernel, we implement a lock involved CPU-based page migration approach. This approach locks the pages and the process cannot modify them during migration, therefore ensuring the data consistency. CPU migration performs better than DMA, as DMA frequency is lower than that of CPU and its initialization time is non-negligible. Upon migrating 10,000 pages, CPU approach takes 43ms, whereas DMA takes 57ms in our experiments. Most importantly, lock involved CPU migration is effective in the cases when WD pages are moved from NVM to DRAM, as we need to ensure data are consistent with migration.

We devise a DMA-based lockless migration approach to migrate pages from DRAM to NVM. As mentioned before, since cold pages are likely to be evicted from DRAM, it is not a good choice to waste the CPU time on migrating these inactive pages. Instead, Memos uses a lockless DMA approach: before migration, a page's *dirty_bit* is set to 0, and not locked when migrated through the DMA channel. We check whether these pages are modified during migration by checking the *dirty_bit* in PTE after the migration finishes. We then create the new PTEs for the successfully migrated pages, whose *dirty_bit* are 0 (i.e., not modified during migration), and discard the dirty pages. The modified pages are still left on DRAM and will be considered for future reclaim (migration), and the freed pages on DRAM are added to a free list. This is a cost-effective approach in cloud environments, as DMA migration is in parallel with CPU operations and will not occupy CPU for migrating cold data.

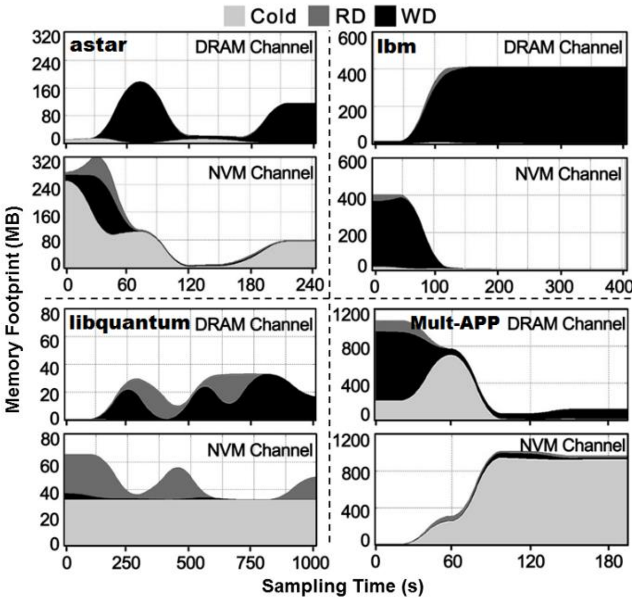


Fig.10. Amount of cold data, hot pages w/ WD and RD patterns in several typical applications in SPECCPU 2006.

TABLE 1. PARAMETERS OF NVM, DRAM AND CACHE [34,69, 54,56]

L1 cache	32KB instruction cache, 32KB data cache, 64B cache block
L2 cache	256KB data cache, 64B cache block
L3 cache	8MB data cache, 64B cache block
DRAM system	t_{RCD}=5 (cy), t_{RP}=5 (cy), t_{WR}=6 (cy), endurance=N/A, read (write) energy=1.17 (0.39) pJ/bit, standby power=1W/GB, refresh power=0.032W/GB, 512MB/bank
NVM system	t_{RCD}=22 (cy), t_{RP}=60 (cy), t_{WR}=6 (cy), endurance=10^7, read (write) energy=2.47 (16.82) pJ/bit, 4GB/bank

Compared with the original DMA approach that needs to lock migrated pages, the lockless approach does not block the processes so that they can use these data during the migration, hence the overall system performance will not be negatively affected. Moreover, it is possible that these modified pages are the active pages, which should not be evicted incorrectly. In such cases, the lockless approach offers a chance to correct the eviction decision, as these modified pages are discarded and not be evicted out of DRAM. In practice, Memos uses CPU with on-demand approach to migrate pages from NVM to DRAM, and adaptively enables lockless DMA migration to evict cold pages from DRAM. We show two of the interfaces. The interface *migrate_cpu (struct page * src, stuct page * des)* is evoked when Memos moves a specific number of hot and WD pages to DRAM. In contrast, with Scatter-Gather [8] mode, after the DMA initiation, DMA migration approach iteratively uses the interface *dma_memcpy_pg_to_pg (dma_channel, oldpage, newpage)* to move pages. Thus it can efficiently move a large number of pages with discrete addresses.

5.3 Emulation of MCHA and Methodology

All of the above-mentioned components are implemented in the Linux kernel. Besides, we emulate MCHA using the channel-partitioning approach [52,66] to divide the memory address space into DRAM and NVM segments on a server with an Intel i7-860/2.8Ghz CPU and DDR3 memory. Memos runs on it. In the experiment, we use the PIN tool to collect workloads' traces after the warm-up period and feed them into an x86 multicore hybrid memory simulator. The simulator's framework is based on the open source hybrid memory simulator [20,58]. We enhanced this simulator's cache with Dinero IV [2] and its memory with DRAMsim2 [1] including the NVM configuration. Moreover, we record the overheads from the OS at runtime such as the PTE updates and page migrations (e.g., CPU uses 16800 cy per 4K page migration using CPU copy and 4200 cy for DMA migration) as well as the sampling overheads using HyMM, and parameterize them into this simulator. Our simulator has memory controllers for DRAM and NVM, respectively. Each controller is with FR-FCFS scheduling policy, 64-bit channel, 64-entry read request queue and 32-entry write buffer. Table 1 shows more parameters, and memory bank information can be found in Sec.6.2.

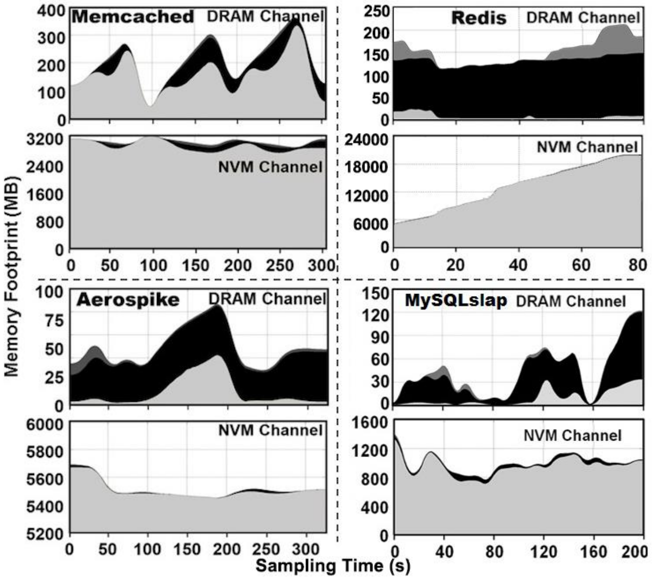


Fig.11. Amount of cold data, hot pages w/ WD and RD patterns in cloud computing cases[4] w/ 1.6GB~24GB footprints.

6 Evaluations

6.1 Effectiveness of Memos on Real System

We firstly test Memos by using SPEC applications. In our experiments, we initially map applications to NVM, whose physical address space starts from 4GB (i.e., NVM channel bit=1 in Fig.8), as NVM might be used as storage in practice. We report the breakdown of WD/RD hot and cold pages for astar, lbm, libquantum and multi-app case that includes several applications run together. Generally, Memos moves hot pages to the DRAM channel, while keeping cold data in the NVM channel. Taking astar in Fig.10 as an example, in the DRAM channel, the footprint of WD pages increases stably at the beginning, indicating that hot pages with WD features are migrated to DRAM continuously. Meanwhile, the number of WD and RD

[4] Memcached (twitter dataset, with random requests, 195K/s random requests [4]), Redis (redis-benchmark [10] with default 50 Clients, 290K/s requests with Zipfian distributions), Aerospike (C Client benchmarks [15] with 145K/s operations), and MySQLslap (default setup w/ 40 connections. These workloads are widely used in big data and cloud cases [38].

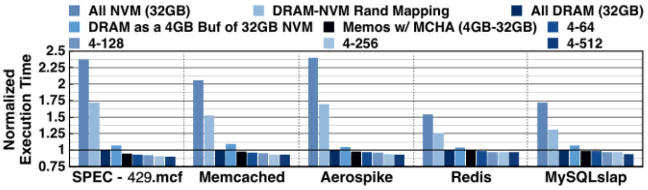


Fig.12. Execution time normalized to 32GB DRAM. Lower is better; x-y means MCHA with xGB DRAM and yGB NVM.

pages in the NVM channel shows a decreasing trend, illustrating these frequently used pages are migrated into DRAM while pages with relative low memory access frequency (cold pages) are left in NVM.

The curve often fluctuates, shown in astar, as it may exhibit diverse and dynamically changing memory access behaviors. For these write-heavy workloads, such as lbm in Fig.10, Memos identifies these WD pages, and migrates them into DRAM channel, and thus we can see most of its memory pages are in DRAM channel. The libquantum figure shows a similar trend. However, libquantum has many cold pages, and therefore Memos keeps them in NVM channel. The bottom-right subfigure shows the metrics for a multi-programmed workload including several SPEC applications. We observe that Memos can handle the diverse memory access patterns well by segregating pages into different memory sub-systems based on memory access frequency and write/read patterns.

Fig.11 shows that Memos works well for these cloud computing applications. For these applications, only a small number of pages are hot, and the hot/cold status of these pages changes frequently. During a preparation period, HyMM can detect these hot pages with overheads of 0.15s, 0.72s 1.18s and 0.79s for Memcached, MySQLslap, Redis, and Aerospike, respectively. For example, there are overall 245 sub-regions in Aerospike (each has 1299 pages on average). HyMM only monitors 0.08% of the pages out of the 1299*245 pages, accounting for only 0.02% of the pages in its 6GB memory space. In contrast, just tracking the *access/dirty_bit* is not cost-effective. It takes 3~6s and is still less accurate when identifying the access frequency and write/read patterns than HyMM, because monitoring has to periodically walk the page table for these applications' whole memory spaces with several GB to 24GB.

6.2 Overall Performance on Emulation Platform

To show the advantages of using NVM, employing Memcached, we conduct a set of experiments by putting 5.0%~20.0% of the data into main memory and leaving the rest of them in the disk, emulating the cases when the main DRAM cannot accommodate the entire working set. The total data size of Memcached is 10GB in our experiments. Our experimental results show that 6.7%~16.7% requests suffer from the long latency of disk (~100X longer than accessing DRAM), leading to around 15.0% throughput lost on average. The long latency disk accesses hurt users' experience. In comparison, all data can fit into the NVM due to its high density and low power, and they are considerably faster than disk accesses (~20X).

Fig.12 shows the normalized execution time of the typical configurations across diverse benchmarks. On average, relative to the baseline 32GB DRAM system, the systems with all NVM perform worse due to NVM's longer latency. Moreover, in the hybrid DRAM-NVM cases, if the memory pages are randomly mapped between memories, the NVM latency is also an un-negligible factor and leading to lower overall performance. In contrast, in the cases where 4GB DRAM is used as a buffer for NVM (similar to [71]), we get nearly DRAM-level performance, as most of the hot pages are moved to DRAM at runtime by hardware. This approach has around 4% higher execution time than all DRAM cases due to the overhead of hardware sampling and data migration. Memos provides near or even better performance on MCHA than system with all DRAM (around 5% benefits, on average). This is because: 1) our experimental results show that on average 83.2% of hot pages with write patterns are migrated to DRAM at runtime, therefore the overall latency of the hybrid memory system is approximately the same as the DRAM-only system; 2) Memos can use the NVM channel to provide data to the CPU, thus having a higher overall bandwidth than only using DRAM; 3) the full hierarchy mechanism in Memos can reduce the memory interferences across the entire memory hierarchy, thus reducing the average memory access time; 4) HyMM and data migration engine have low overheads.

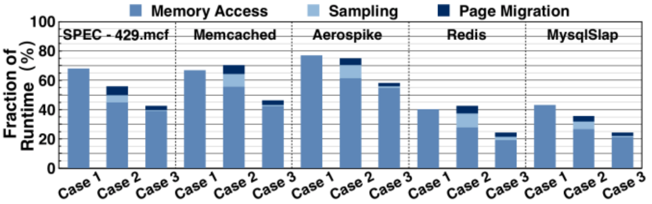


Fig.13. Performance breakdown. In Case 1, memory pages are randomly mapped between DRAM and NVM; In Case 2, memory pages are mapped using previous sampling and migration approaches [57,87][5]; Case 3 uses Memos. Lower is better, indicating more computing time.

Moreover, we test the scalability of Memos. In our experiments, we use NVM with the capacity from 32GB to 512GB, and the number of bank increases from 8 to 128 (4GB/bank). We find Memos scale well and the overall performance is even better with larger NVM capacity, as it can have more memory banks work in parallelism.

Performance Breakdown: Fig.13 shows the performance breakdown of Memos' runtime across several typical cloud workloads. The runtime includes memory accesses time, sampling, page migration overhead and computing time. The most time-consuming part is memory access. Without any optimizations, data are randomly mapped between DRAM and NVM (case 1). Due to NVM's longer latency, the overall time costs of memory accesses vary between 40.3% (Redis with fewer memory accesses) and 76.9% (Aerospike). Prior efforts [57,87] conduct a page-level sam-

[5] We implement a baseline system (case 2) that absorbs the core ideas in related OS-level work [57,87]. [57] heavily relies on the PTE walkers, and the clock-hand algorithm (its Fig.6) for placing pages is complicated in practice. And, [57] is w/o details on memory arch-level optimization and doesn't consider the migration method and overheads. [87] samples PTEs in a jumping approach, but it doesn't provide the WD info., and it is not accurate for the poor locality cases in cloud environments.

pling and move the hot/WD pages to DRAM, thus reducing the memory access latency (Case 2) [5]. However, due to the high sampling and page migration overheads, especially for the cloud workloads, the overall performance improvement is limited. In the case of Memcached and Redis, the overall performance degrades because the sampling and page migration overheads offset the benefits.

Mentioned in Sec.3, Memos' sampling overhead is less than 1/10 of that of prior efforts, and the migration overhead is roughly reduced by more than half due to using DMA evicting cold pages (Sec.5). More importantly, illustrated in Fig.13, the full hierarchy memory mechanism further reduces the memory latency (Sec.4), and thus the overall computing time is significantly improved by around 20%.

TABLE 2. 128GB NVM'S LIFETIME (YRS) W/ AND W/O MEMOS.

Bench/Policies/ Time (years)	Rand Map	Memos w/ MCHA	Memcached	0.50	12.2
mcf	0.27	9.3	Aerospike	0.43	10.1
hmm	3.6	15.8	Redis	0.52	13.4
Hmm,xal,mcf	0.24	9.1	Mysqlslap	1.44	15.6

6.3 Details on Energy, Lifetime and Migration Overhead

Energy: We use Micron System Power Calculator [19]. For NVM system, we use the values of read-power, write-power and idle-power of NVM relative to DRAM. Memos with MCHA (4GB DRAM and 32/128GB NVM) saved 29% and 82.5% energy on memory systems compared to 32GB and 128GB all DRAM, respectively. The saving energy is mainly from the near zero refresh operation on NVM.

NVM Lifetime Improvements: For lifetime calculation, we model the NVM with the cell write endurance of 10^7 in Table 1. The NVM is operated at 64 bytes blocks. Also, we emulate the NVM that uses an effective write leveling scheme (e.g., Start-Gap [70]), thus the overall NVM manages to achieve an overall lifetime which is 95% of the average NVM cell lifetime. Experimental results (using the model in [71]) show that Memos on MCHA can improve the NVM life by up to 34X (mcf) against the policy that randomly maps memory pages between DRAM and NVM, as Memos moves 83.2% of Hot and WD pages out of NVM, on average. Details are in Table 2. Besides SPEC applications, NVM can also have longer lifetime in the cloud computing workloads.

Data Migration Overhead: The migration only happens in the 20s action period in each 40s interval (in Fig.5), and the amortized overhead is low. Moreover, our lockless DMA migration engine can share the burden for CPU. For example, in case of Aerospike, Memos needs to evict 3376 cold pages from DRAM to NVM when it runs to 40th second. With our DMA-based lockless migration approach, 98.6% of them are migrated via DMA and thus saving CPU time for migration. 1.4% of them are modified (be active again) during the migration, and therefore our approach correctly keeps them in DRAM as discussed in Sec.5.2.

7 Related Work and Discussions

(1) New Memory Systems. Many studies design the new memory architecture [25,32,34,46,71,88], as well as typical studies in [27,42,56,72,82,86] optimize the memory controller logic, buffer organization, write operations, and row-buffer locality for NVM performance and security. Further work studies the memory management and task allocation accordingly [36,64,74,75,77], and even for big data and virtualization environments [41,44,89]. For high reliability and availability, latest studies redesign systems (e.g., databases) for platforms that use NVM [24,59,85] and end client devices [43]. The approach in [44] extracts OS-level information about an application's memory usage to avoid page migrations on hybrid memory systems. The work in [71] is a starting point to address NVM's challenges for main memory systems by using DRAM as a buffer of NVM. This approach benefits the overall system performance, NVM lifetime, and reduces the write traffic. [33] uses NVM in GPU architecture. Memos is an orthogonal design with these studies, and it is cost-effective on the platforms with the horizontally organized hybrid DRAM and NVM memories (e.g., Lenovo's ThinkSystem SD650 servers [18] has 3D XPoint DIMM at the same level with DDR DIMM at memory hierarchy). **(2) NVM Allocator.** The work in [7] provides an open source NVM allocator for both persistent and volatile usage. SSDAlloc [29] provides an API to users for using SSDs on hybrid memory systems. [76] describes a file-only principle for NVM management, having a constant time memory operation that is independent of size. Our work is complementary to these efforts. Memos tries to maximize the memory utilization across the entire hybrid memory hierarchy, while still being transparent to users and applications. **(3) Page-Coloring.** Many previous studies [48,51,53] use cache/bank-indexing bits in physical address mapping scheme to partition cache, banks and channels for performance. Our work differs in the involved address bits, including not only the cache and bank bits, but also the channel bits simultaneously. However, as our platform only has 1 channel bit, this could be a limitation in our experiments. If a platform has more channel bits, Memos could be extended to support such cases by organizing the free pages to multi-tiered. **(4) Monitoring Memory Access Patterns.** Previous work [35,40,62,74] conduct online profiling by leveraging hardware performance counters. Recent efforts also design the OS-level memory page behavior monitoring approaches by referencing TLB misses [26] and *access_bit* [80]. HyMM includes the advantages of monitoring the *access/dirty_bit* and TLB misses, and can obtain the memory hierarchy utilization on the fly at the OS level on commodity systems. Specifically, HyMM can detect page-level reads/writes and stream-like memory patterns, which are critical to consider in a hybrid memory environment. **(5) Page Migration on Hybrid Memory Systems.** [73] designs migration queues for DRAM and NVM. [31] proposes a concurrent migration approach that can migrate multiple pages efficiently. In [50], memif redefines the DMA engine configuration to improve migration performance. In contrast, Memos' migration engine enhances DMA with a lockless approach and adaptively enables the CPU and lockless DMA enhancements accordingly. **(6) Huge Page.** HyMM supports huge page sampling. Huge pages allow Memos to use NVMs, but the resulting fragmentation is a challenge. The work in [67] skips hybrid page blocks during compaction and [47] proposes asynchronous allocation to create contiguous memory spaces. Our future work will try to optimize Memos using these latest huge page policies. **(7) Wire Delays.** Due to the page limit, we only used average cache delay for all slices for which the results showed performance improvements over the baseline. However, the results could be further improved using adaptive NUCA designs [30,39]. For example, we could place hot sets into cache slices close to the cores and place infrequently accessed data into slices far away from the cores. **(8) Mis-classifications and Rectification in HyMM.** If the applications change dynamically and change the memory footprint, causing the hot/cold classification become outdated, Memos will re-sampling and rectify the misclassifications in next loop and enable migration accordingly. For instance, the above-mentioned "very hot" pages that cannot be found by monitoring TLB misses in section 3.2 could be detected and reclassified in the next sampling, as the memory pattern changes. The similar approach is also mentioned in [26]. **(9) Memos and the Environments.** We use an Intel i7-860 CPU as an example in our experiments, as its address mapping can be leveraged [51~54]. This CPU supports adjusting its address mapping policy to the one with fewer overlapped bits that index both cache sets and DRAM banks by modifying the configurations in BIOS, i.e., more bits (15~18 bit) only index the cache sets. In this paper, we use such an address mapping, as we want to show the results without many uncertain factors, such as reducing the bandwidth and the number of available memory banks brought by using overlapped bits [48,53,54]. And, we also do not consider the I/O [60,61] and network issues [90] in current design. On a multi-socket platform, Memos could be extended to support by organizing the free pages to tiers according to the number of sockets. And, we would try to test Memos in virtualized environments with more resources in future work.

8 Conclusions

This paper designs Memos for hybrid memory management. It includes HyMM for capturing the online memory utilization for workloads with large memory footprints within low overhead; Full hierarchy memory framework to schedule the memory resources across the entire memory hierarchy according to memory patterns and NVM/DRAM's features; and a hybrid page migration mechanism, which combines a lockless DMA engine and CPU-based migration approach. Our experimental results show that Memos works well on the platform with the hybrid DRAM-NVM memory system. Memos can be deployed on systems equipped with Fast-Slow memories potentially.

Acknowledgement

We thank the reviewers for their valuable comments and the researchers who helped us on the previous manuscripts of this paper. We also thank the previous student members in Sys-Inventor lab for conducting some of the experiments. The corresponding author is Lei Liu. Part of the technique in this paper is based on Lei Liu's previous work in ISCA, PACT, TC, TACO, and ICCD [51~55], and the corresponding author of these studies [51~55] is Lei Liu.

This project is supported by the National Key Research and Development Plan of China under Grant No. 2017YFB1001602, and the NSF of China under Grant No. 61502452.

References

[1] DRAMSim2. http://www.eng.umd.edu/~blj/dramsim/

[2] DineroIV Trace-Driven Uniprocessor Cache Simulator. http://pages.cs.wisc.edu/~markhill/DineroIV/

[3] Intel perfmon. http://oprofile.sourceforge.net/docs/intel-perfmon-events.php

[4] Memcached. http://memcached.org/

[5] Pin 2.14. https://software.intel.com/sites/landingpage/pintool/docs/71313/Pin/html

[6] Standard Performance Evaluation Corporation. http://www.spec.org/cpu2006

[7] NVM Library. http://pmem.io/nvml, 2014

[8] https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt

[9] Redis: https://redis.io/topics/benchmarks

[10] https://redis.io/topics/benchmarks

[11] https://en.wikipedia.org/wiki/Interquartile_range

[12] https://en.wikipedia.org/wiki/Box_plot

[13] STREAM: https://www.cs.virginia.edu/stream/ref.html

[14] Aerospike: http://www.aerospike.com/

[15] https://github.com/aerospike/aerospike-client-c/tree/master/benchmarks

[16] dev.mysql.com/doc/refman/5.7/en/mysqlslap.html

[17] https://mariadb.com/kb/en/library/mysqlslap/

[18] https://lenovopress.com/lp0636-thinksystem-sd650-direct-water-cooled-server

[19] https://www.micron.com/support/tools-and-utilities/power-calc

[20] "Utility-Based Hybrid Memory Management Simulator," https://github.com/CMU-SAFARI/UHMEM, 2017.

[21] Intel® 64 and IA-32 Architectures Developer's Manual. http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-manual-325462.pdf

[22] Intel. Intel and Micron Produce Breakthrough Memory Technology. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/, 2015. [Online accessed 29-April-2016].

[23] T. P. Morgan. Intel shows off 3D XPoint memory performance. https://www.nextplatform.com/2015/10/28/intel-shows-off-3d-xpoint-memory-performance/, Oct.2015.

[24] J. Arulraj, A. Pavlo, and S. R. Dulloor, "Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems," In SIGMOD, 2015.

[25] B. Amrutur and M. Horowitz, "Speed and Power Scaling of SRAM's," In IEEE Solid-State Circuits, 2000.

[26] N. Agarwal and T. F. Wenisch, "Thermostat: Application-transparent Page Management for Two-tiered Main Memory," In ASPLOS, 2017.

[27] A. Awad, P. K. Manadhata, Y, Solihin, S. Haber and W. Horne, "Silent Shredder: Zero-Cost Shredding for Secure Non-Volatile Main Memory Controllers," In ACM SIGARCH Computer Architecture News, 2016.

[28] A. Awad, A. Basu, S. Blagodurov, Y. Solihin and Gabriel H. Loh, “Avoiding TLB shootdowns through self-invalidating TLB entries,” In PACT, 2017.

[29] A. Badam and V. S. Pai, “SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy,” In NSDI, 2011.

[30] B. M. Beckmann, M. R. Marty and D. A. Wood, “ASR: Adaptive selective replication for CMP caches,” In Micro, 2006.

[31] S. Bock, B. Childers, R. Melhem and D. Mosse, “Concurrent Migration of Multiple Pages in Software-Managed Hybrid Main Memory,” In ICCD, 2016.

[32] A. M. Caulfield, A. De, J. Coburn, T. I. Mollow, R. K. Gupta, S. Swanson, “Moneta: A High-performance Storage Array Architecture for Next-generation, Non-volatile Memories,” In Micro, 2010.

[33] S. Chen, et al, “Efficient GPU NVRAM Persistence with Helper Warps,” In DAC, 2019.

[34] G. Dhiman, R. Ayoub, T. Rosing, “PDRAM: A Hybrid PRAM and DRAM Main Memory System,” In DAC, 2009.

[35] Ramos. L. E, Gorbatov. E, Bianchini. R. “Page Placement in Hybrid Memory Systems,” In ICS, 2011.

[36] M. Fu, D, Feng, Y. Hua, X. He, Z. Chen, W. Xia, F. Huang, Q. Liu, “Accelerating Restore and Garbage Collection in Deduplication-based Backup Systems via Exploiting Historical Information,” In USENIX ATC, 2014.

[37] J. Gandhi, A. Basu, M. H. Hill, and M. M. Swift, “A Tool to Instrument x86-64 TLB Misses,” In SIGARCH Computer Architecture News (CAN), 2014.

[38] R. Han, et al, “Benchmarking Big Data Systems: A Review,” In IEEE Transactions on Services Computing (TSC), 2017.

[39] J. Huh, C. Kim, H. Shafi, L. Zhang, D. Burger, and S. W. Keckler, “A NUCA Substrate for Flexible CMP Cache Sharing,” In ICS, 2005.

[40] A. Jaleel, H.H. Najaf-Abadi, S. Subramaniam, S.C. Steely, and J. Emer, “CRUISE: Cache Replacement and Utility-Aware Scheduling,” In ASPLOS, 2012.

[41] M. R. Jantz, C. Strickland, K. Kumar, M. Dimitrov, and K. A. Doshi, “A Framework for Application Guidance in Virtual Memory Systems,” In VEE, 2013.

[42] L. Jiang, B. Zhao, Y. Zhang, J. Yang, B. Childers, “Improving Write Operations in MLC Phase Change Memory,” In HPCA, 2012.

[43] S. Kannan, A. Gavrilovska, K. Schwan, “Reducing the Cost of Persistence for Nonvolatile Heaps in End User Devices,” In HPCA, 2014.

[44] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan, “HeteroOS: OS Design for Heterogeneous Memory Management in Datacenter,” In ISCA, 2017.

[45] S. Kanev, J. P. Darago, K. Hazelwood, T. Parthasarathy, R. Moseley, G. Wei and D. Brooks, “Profiling a Warehouse-scale Computer,” In ISCA, 2015.

[46] E. Kultursay, M. Kandemir, A. Sivasubramaniam, O. Mutlu, “Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative,” In ISPASS, 2013.

[47] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, E. Witchel, “Coordinated and Efficient Huge Page Management with Ingens,” In OSDI, 2016.

[48] J. Lin, Q. Lu, X. Ding, Z. Zhang, X. Zhang, and P. Sadayappan. “Gaining Insights into Multicore Cache Partitioning: Bridging the Gap Between Simulation and Real Systems,” In HPCA, 2008.

[49] J. Liu, B. Jaiyen, R. Veras, O. Mutlu, “RAIDR: Retention-Aware Intelligent DRAM Refresh,” In ISCA, 2012.

[50] F. X. Lin and X. Liu, “memif: Towards Programming Heterogeneous Memory Asynchronously,” In ASPLOS, 2016.

[51] L. Liu, et al, “A Software Memory Partition Approach for Eliminating Bank-Level Interference in Multicore Systems,” In PACT, 2012.

[52] L. Liu, et al, “BPM/BPM+: Software-based Dynamic Memory Partitioning Mechanisms for Mitigating DRAM Bank-/Channel-level Interferences in Multicore Systems,” In ACM TACO, 2014.

[53] L. Liu, et al, “Going Vertical in Memory Management: Handling Multiplicity by Multi-policy,” In ISCA, 2014.

[54] L. Liu, et al, “Rethinking Memory Management in Modern Operating System: Horizontal, Vertical or Random?” In IEEE Trans. on Computers (TC), 2016.

[55] L. Liu et al, “Memos: A Full Hierarchy Hybrid Memory Management Framework,” In ICCD, 2016.

[56] B. C. Lee, E. Ipek, O. Mutlu, D. Burger, “Architecting Phase Change Memory as a Scalable DRAM Alternative,” In ISCA, 2009.

[57] S. Lee, H. Bahn, SH. Noh, “CLOCK-DWF: A Write-History-Aware Page Replacement Algorithm for Hybrid PCM and DRAM Memory Architectures,” In IEEE Trans. on Computers (TC), 2014.

[58] Y. Li, S. Ghose, J. Choi, J. Sun, H. Wang and O. Mutlu, “Utility-Based Hybrid Memory Management,” In Cluster, 2017.

[59] J. Lindstom, D. Das, T. Mathiasen, D. Arteaga, N. Talagala, “NVM aware MariaDB database system,” In NVMSA, 2015.

[60] F. Lv, et al, “Dynamic I/O-Aware Scheduling for Batch-Mode Applications on Chip Multiprocessor Systems of Cluster Platforms,” In JCST, 2014.

[61] F. Lv, et al, “WiseThrottling: A New Asynchronous Task Scheduler for Mitigating I/O Bottleneck in Large-Scale Datacenter Servers,” In J. of Supercomputing, 2015.

[62] H. T. Mai, K. H. Park, H. S. Lee, C. S. Kim, M. Lee, S. J. Hur, “Dynamic Data Migration in Hybrid Main Memories for In-Memory Big Data Storage,” In ETRI Journal, 2014.

[63] M. Marinella, “The Future of Memory,” In IEEE Aero-space Conference, 2013.

[64] J. C. Mogul, E. Argollo, M. Shah, P. Faraboschi, “Operating System Support for NVM+DRAM Hybrid Main Memory,” In HotOS, 2009.

[65] O. Mutlu, “Main Memory Scaling: Challenges and Solution Directions,” In More than Moore Technologies for Next Generation Computer Design, 2015.

[66] S. P. Muralidhara, L. Subramanian, O. Mutlu, M. Kandemir, and T. Moscibroda,“Reducing Memory Interference in Multicore Systems via Application-Aware Memory Channel Partitioning,” In Micro, 2011.

[67] A. Panwar, A. Prasad, K. Gopinath, “Making Huge Pages Actually Useful,” In ASPLOS, 2018.

[68] Y. Park, S. K. Park, K. H. Park “Linux Kernel Support to Exploit Phase Change Memory,” In Proceedings of Linux Symposium, 2010.

[69] Moinuddin K. Qureshi, Gurumurthi. S, Rajendran. B, “Phase Change Memory: From Devices to System,” In Synthesis Lectures on Computer Architecture, 2011.

[70] Moinuddin K. Qureshi, M. Franchescini. V. Srinivasan, “Enhancing Lifetime and Security of PCM-Based Main Memory with Start-Gap Wear Leveling,” In Micro, 2009.

[71] Moinuddin K. Qureshi, V. Srinivasan, and J. A. Rivers. “Scalable High Performance Main Memory System Using Phase-change Memory Technology,” In ISCA, 2009.

[72] Moinuddin K. Qureshi, M. Franceschini, and L. Lastras Montano, “Improving Read Performance of Phase Change Memories via Write Cancellation and Write Pausing,” In HPCA, 2010.

[73] R. Salkhordeh and H. Asadi, “An Operating System Level Data Migration Scheme in Hybrid DRAM-NVM Memory Architecture,” In DATE, 2016.

[74] H. Seok, Y. Park, KH. Park, “Migration Based Page Caching Algorithm for a Hybrid Main Memory of DRAM and PRAM,” In SAC, 2011.

[75] Daniel J. Sorin, Milo M. K. Martin, Mark D. Hill, David A. Wood, “Safetynet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery,” In ISCA, 2002.

[76] Michael M. Swift, “DRAFT: Towards O(1) Memory,” In HotOS, 2017.

[77] W. Tian, Y. Zhao, L. Shi, Q. Li, J. Li, CJ. Xue, M. Li, E. Chen, “Task Allocation on Nonvolatile-Memory-Based Hybrid Main Memory,” In TVLSI, 2013.

[78] X. Wu, J. Li, L. Zhang, E. Speight, R. Rajamony, Y. Xie, “Hybrid Cache Architecture with Disparate Memory Technologies,” In ISCA, 2009.

[79] X. Xiang, C. Ding, H. Luo, B. Bao, “HOTL: A Higher Order Theory of Locality,” In ACM SIGARCH Computer Architecture News, 2013.

[80] M. Xie, et al, “SysMon: Monitoring Memory Behaviors via OS Approach,” In APPT, 2017.

[81] C. Xue, Y. Zhang, Y. Chen, G. Sun, J. Yang, H. Li, “Emerging Non-Volatile Memories: Opportunities and Challenges,” In CODES+ISSS, 2011.

[82] H. Yoon, J. Meza, R. Ausavarungnirun, R. Harding, and O. Mutlu, “Row Buffer Locality Aware Caching Policies for Hybrid Memories,” In ICCD, 2012.

[83] Wang. Z, Jiménez. D. A, Xu. C, “Adaptive Placement and Migration Policy for an STT-RAM-Based Hybrid cache,” In HPCA, 2014.

[84] Z. Zhang, Z. Zhu, X. Zhang, “A Permutation-based Page Interleaving Scheme to Reduce Row-Buffer Conflicts and Exploit Data Locality,” In Micro, 2000.

[85] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, “Mojim: A Reliable and Highly-Available Non-Volatile Memory System,” In ASPLOS, 2015.

[86] L. Zhang, B. Neely, D. Franklin, D. Strukov, Y. Xie, F. T. Chong, "Mellow Writes: Extending Lifetime in Resistive Memories through Selective Slow Write Backs,” In ISCA, 2016.

[87] X. Zhang, S. Dwarkadas, K. Shen, “Towards practical page coloring-based multicore cache management,” In EuroSys, 2009.

[88] P. Zhou, B. Zhao, J. Yang, Y. Zhang, “A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology,” In ISCA, 2009.

[89] R. Zhou, T. Li, “Leveraging Phase Change Memory to Achieve Efficient Virtual Machine Execution,” In VEE, 2009.

[90] Y. Hu, T. Li, "Enabling Efficient Network Service Function Chain Deployment on Heterogeneous Server Platform," In HPCA, 2018.

Lei Liu is an associate professor in Institute of Computing Technology (ICT), CAS, where he leads the Sys-Inventor Lab. His research interests include OS, memory architecture and computer architecture. His efforts are published in ISCA, PACT, DAC, IEEE TC, ACM TACO, and etc.

Shengjie Yang is a student member of Sys-Inventor Lab in ICT, CAS. His work includes the OS design and optimizations on memory systems. He is supervised by Lei Liu.

Lu Peng is a full professor at Louisiana State University (LSU). His research interests include computer architecture, microarchitecture, memory hierarchy and performance analysis. His efforts are published in ISCA, HPCA, DAC, ICCD, ICS, and etc.

Xinyu Li is a student member of Sys-Inventor Lab in ICT, CAS. His work includes OS design and GPU programming. He is supervised by Lei Liu.

Appendix: The workloads’ information is listed as below. As mentioned in our paper, each workload has at least one libquantum, which provides stream-like memory pattern.
Workload1: omnetpp, libquantum, mcf, Xalan
Workload2: omnetpp, libquantum*2, lbm, astar
Workload3: bzip2, libquantum*2, soplex, Xalan
Workload4: libquantum*3, mcf, soplex, astar
Workload5: libquantum*2, cactusADM, tonto, mcf, bzip2
Workload6: libquantum*3, gobmk, gcc, deal2, soplex
Workload7: libquantum*2, lbm, h264ref, namd
Workload8: omnetppp, libquantum*3, sjeng, leslie3d
Workload9: soplex, libquantum*3, gcc, bzip2, Xalan
Workload10: omnetpp, libquantum*4, mcf, soplex, astar