Miss Penalty Aware Cache Replacement for Hybrid Memory Systems

Hai Jin, *Fellow, IEEE*, Di Chen, Haikun Liu, *Member, IEEE*, Xiaofei Liao, *Member, IEEE*, Rentong Guo, and Yu Zhang, *Member, IEEE*

***Abstract*—Current DRAM-based memory systems face the scalability challenges in terms of memory density, energy consumption, and monetary cost. Hybrid memory architectures composed of emerging nonvolatile memory (NVM) and DRAM is a promising approach to large-capacity and energy-efficient main memory. However, hybrid memory systems pose a new challenge to on-chip cache management due to the asymmetrical penalty of memory access to DRAM and NVM in case of cache misses. Cache hit rate is no longer an effective metric for evaluating memory access performance in hybrid memory systems. Current cache replacement policies that aim to improve the cache hit rate are not efficient either. In this article, we take into account the asymmetry of the cache miss penalty on DRAM and NVM, and advocate a more general metric, average memory access time (AMAT), to evaluate the performance of hybrid memories. We propose a miss penalty aware LRU-based cache replacement policy (MALRU) for hybrid memory systems. MALRU is aware of the source (DRAM or NVM) of missing blocks and preserves high-latency NVM blocks as well as low-latency DRAM blocks with good temporal locality in the last level cache. The experimental results show that MALRU can improve system performance by up to 22.8% and 13.1%, compared to LRU and the state-of-the-art hybrid memory aware cache partitioning technique policy, respectively.**

***Index Terms*—Cache replacement, hybrid memory systems, least recently used (LRU), nonvolatile memory (NVM).**

I. Introduction

IN-MEMORY computing is becoming increasingly popular for data-intensive applications in the big data era. However, current DRAM technologies are hard to meet the continuously increasing requirement of main memory due to low memory density and high power consumption [1]. As a result, a large body of work pays more attention to the emerging nonvolatile memory (NVM), such as phase-change memory (PCM), spin-torque transfer RAM (STT-RAM), and 3D XPoint [2]. NVMs features byte-addressability, near-zero static power consumption, much higher density, and lower cost than DRAM [3], [4]. However, NVMs have some disadvantages, such as asymmetrical read/write latencies and limited write endurance. These drawbacks make NVM hard to be a direct substitute of DRAM. Therefore, hybrid memory systems comprising DRAM and NVM become a practical way to build a large-scale and high-performance main memory system.

Caches play an important role in bridging the performance gap between CPU and main memory. Efficient use of cache can avoid abundant memory accesses. Existing cache replacement policies [5], [6] are generally designed to improve the hit rate of last level cache (LLC). In DRAM-based main memory systems, a higher hit rate of LLC implies more efficient use of LLC. However, these cache replacement policies are no longer efficient in hybrid memory systems as the cache miss penalties for DRAM and NVM are significantly different. The latency of fetching an NVM block is several times larger than that of fetching a DRAM block.

Fig. 1 shows the normalized execution time of several applications from SPEC 2006 benchmarks running in DRAM-based and NVM-based main memory. When the access latency of NVM is 4 times higher than that of DRAM, these applications consume 45% to 195% more time in NVM-based memory than the execution in DRAM-based memory. For each application, the misses per kilo instructions (MPKIs) of LLC is the same in the two experiments. However, the application performance varies significantly due to the location of miss data in different main memories. This implies the LLC hit rate is no longer an effective metric for evaluating the memory performance in hybrid memories since the different cache miss penalties between DRAM and NVM should be considered. Also, current cache replacement policies that aim to improve the cache hit rate are no longer efficient in hybrid memory systems.

A new metric is needed to reflect the memory performance of hybrid memory systems. Intuitively, cache replacement algorithms such as LRU should preferentially choose a DRAM block as a victim upon cache replacement because evicting an NVM block suffers higher write latency than evicting a DRAM block. However, such a simple rationale may cause cache thrashing if a DRAM block with good data locality is evicted from the LRU stack (see more analysis in Section II). On a cache miss, it is challenging to minimize the cache miss penalty while maintaining good data locality.

Manuscript received May 14, 2019; revised August 16, 2019 and October 31, 2019; accepted December 29, 2019. Date of publication January 13, 2020; date of current version November 20, 2020. This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB1001603, and in part by the National Natural Science Foundation of China under Grant 61672251, Grant 61732010, Grant 61825202, and Grant 61929103. This article was recommended by Associate Editor C.-L. Yang. *(Corresponding author: Haikun Liu.)*

The authors are with the National Engineering Research Center for Big Data Technology and System, Service Computing Technology and System Laboratory, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China (e-mail: hjin@hust.edu.cn; chendi@hust.edu.cn; hkliu@hust.edu.cn; xfliao@hust.edu.cn; rtguo@hust.edu.cn; yuzh@hust.edu.cn).

Digital Object Identifier 10.1109/TCAD.2020.2966482

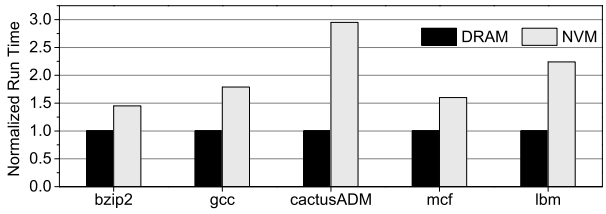


Fig. 1. Normalized execution time of SPEC 2006 benchmark in DRAM-based main memory and NVM-based main memory.

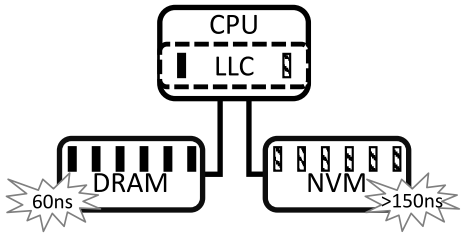


Fig. 2. Flat-addressable hybrid memory architecture.

In this article, we take into account the asymmetry of LLC miss penalty on heterogeneous memories and propose a miss penalty aware LRU-based cache replacement policy (MALRU) for hybrid memory systems. First, we advocate a more general performance metric—average memory access time (AMAT) to assess cache performance in hybrid memory systems. Second, we add a flag in the tag of cache lines to distinguish the miss penalty of each DRAM or NVM block. Third, we partition the traditional LRU stack into a reserved section and a regular victim section by calculating the minimum value of AMAT. MALRU keeps the high-latency NVM blocks and most frequently accessed DRAM blocks in the reserved section. Finally, we propose a simple yet effective policy to simplify the calculation of AMAT and hardware implementation of MALRU.

The contributions of this article are summarized as follows.

1) We investigate the asymmetry of LLC miss penalty on heterogeneous memories and observe the inefficiency of current cache replacement policies. We thus advocate the AMAT metric to assess the memory performance of hybrid memory systems.
2) We propose MALRU, a miss-penalty aware cache replacement policy. MALRU advocates two heuristics to guide cache replacement in hybrid memory systems. MALRU preferentially evicts low-latency DRAM blocks and keeps high-latency NVM blocks and some DRAM blocks with good data locality in a reserved section. MALRU periodically adjusts the reserved section to protect the most frequently accessed DRAM blocks that brings even more performance benefit than NVM blocks. Moreover, to mitigate the hardware complexity and software overhead of MALRU, we further propose a simple yet effective solution called MALRU-CR.
3) We evaluate MALRU and MALRU-CR with SPEC CPU 2006 benchmark, and compare MALRU and MALRU-CR with the traditional least recently used (LRU) and the state-of-the-art hybrid memory aware cache partitioning technique (HAP) policy in a hybrid memory system. Compared to conventional LRU, experimental results show that MALRU and MALRU-CR can improve system performance by up to 22.8% (11.8% on average) and 21.4% (10.5% on average), respectively. Compared to HAP [7], MALRU and MALRU-CR improve application performance by up to 13.1% (6.4% on average) and 10.7% (5.0% on average), respectively.

The remainder of this article is organized as follows. Section II introduces the background and related work. Section III motivates the penalty-aware cache replacement policy in hybrid memories. Section IV introduces the design of MALRU. Section V provides the experimental methodology and results. Section VI concludes this article.

II. Background and Related Work

In this section, we first briefly introduce hybrid DRAM/NVM memory architectures and then describe the previous cache replacement policies.

A. Hybrid Memory Architectures

Hybrid memory architectures comprising of traditional DRAM and emerging NVM have been widely studied since 2009 [8]–[14]. Previous studies have proposed two-hybrid DRAM/NVM main memory architectures: 1) hierarchical cache/memory architecture [15]–[18] and 2) flat-addressable (single address space) memory architecture [8]–[10]. We note that the recently available Intel Optane dc persistent memory [19] also offers two use modes: 1) Memory Mode and 2) App Direct Mode, which corresponds to the hierarchical cache/memory architecture and flat-addressable memory architecture, respectively. In the memory mode, the Optane DIMMs serve as cost-effective main memory, and DRAM acts as a cache to the Optane DIMMs. The DRAM cache is invisible to OSes and applications. In the App Direct Mode, applications and OS are aware of the two types of memory available. Fig. 2 shows a typical flat-addressable hybrid memory architecture in which both DRAM and NVM are uniformly organized as the main memory in a single flat address space. Both DRAM and NVM are attached to the memory bus, and are visible to the processors and OSes.

Generally, the hierarchical cache/memory architecture can deliver higher performance for application with good data locality, while the flat-addressable memory architecture is better for applications with random memory access patterns. Moreover, the flat-addressable architecture is able to offer more memory capacity and higher memory bandwidth for applications than the hierarchical cache/memory architecture. There is not a verdict on which memory architecture is better than another one. The design of DRAM/NVM architecture should systematically consider memory access characteristics of applications and memory management strategies in OSes.

This article chooses the flat-addressable hybrid memory architecture for the following considerations. Most popular big data applications usually reflect very poor data locality. In the hierarchy hybrid memory architecture, DRAM cache blocks are usually directly mapped to NVM pages for low hardware complexity, and thus each DRAM cache miss leads to a data

TABLE I
COMPARISON OF DIFFERENT FEATURES BETWEEN SSD, DRAM, PCM, RRAM, AND STT-RAM

Technology	Read Latency (ns)	Write Latency (ns)	Write Endurance (time)	Density	Cost
Flash SSD	25,000	200,000	10^5	High	Low(<0.1 $/Gb)
DRAM	60	60	$>10^{16}$	Low	Low(<1 $/Gb)
PCM	100-300	150-100,000	10^8	Medium/High	Medium(few $/Gb)
STT-RAM	2-20	2-20	10^{15}	Low	High(50 $ - 100 $/Gb)
RRAM	30-200	100-200	10^{10}	High	High(5,000 $/Gb)

transfer at the page granularity. However, it is unnecessary and costly to fetch infrequently accessed data from NVM to the DRAM cache. Previous studies [17], [18] have shown that large-footprint applications with poor data locality would suffer frequent DRAM cache misses and high memory bandwidth consumption between DRAM and NVM. In contrast, the flat-addressable hybrid memory architecture can offer more flexible memory management at the OS layer, which can exploit some heuristics to place frequently accessed (hot) data in DRAM explicitly. Furthermore, the OS can also migrate hot data from NVM to DRAM, and logically use the DRAM as an exclusive cache.

However, there remain challenges to efficiently utilize the hybrid memories in the flat-addressable hybrid memory architecture due to different performance, endurance, and cost characteristics of DRAM and NVM, as shown in Table I. Take memory access latency as an example, the read and write latencies of PCM are approximately 4.4× and 12× higher than that of DRAM [19], [20], respectively. When most of memory requests are distributed on NVM rather than DRAM, the applications usually suffer significant performance degradation. To improve the performance and energy efficiency of flat-addressable hybrid memory systems, previous studies [12]–[14], [21], [22] have focused on mitigating the impact of NVM latency on application performance by migrating hot NVM pages to DRAM. However, page migration usually relies on costly hot page monitoring and sorting [9], [14]. This article explores another direction to improve the performance of flat-addressable hybrid memory systems at the on-chip cache layer and propose a miss-penalty aware cache replacement scheme.

There are some other key NVM characteristics that also have a significant impact on the performance and lifetime of NVMs, such as access latency discrepancy due to IR drop [23], wear leveling [17], and shift overheads of racetrack memories [24]. Particularly, the write endurance of PCM is a major concern when it is used as the main memory. In the past decade, there have been many studies on improving NVM lifetime through hardware and software approaches, including wear-leveling [17], bit-write reduction [3], [5], DRAM buffering [8], [9], and so on. The advance of PCM technologies has demonstrated that the write endurance can be significantly improved to 10^{12} cycles [25]. Intel also claims that Intel Optane dc persistent memory is able to last 5 years' lifetime writes [26]. These technological evolutions allow PCM a promising memory alternative to complement DRAM.

In this article, we mainly study cache replacement policies for hybrid memory systems, and thus only considers the difference of access latency between NVM and DRAM. Although previous studies on other key NVM characteristics are orthogonal to this article, they are complementary techniques for constructing a high-performance and energy-efficiency hybrid memory system.

B. Related Work

On-chip cache is able to significantly narrow the performance gap between processors and main memory. However, due to the limited capacity of LLC, hot data may be evicted from LLC and later is rereferenced again. Cache contention can lead to severe cache thrashing problems. A sophisticated cache replacement mechanism should mitigate cache thrashing and improve the cache hit rate.

1) Cache Replacement for Homogeneous Memory: There have been a large body of work on cache replacement policies. LRU-based cache replacement policies have been widely studied in the past years. The primary goal of LRU and its derivative algorithms is mainly to improve the hit rate of LLC. These policies [27]–[34] usually work well in most cases. Without considering the impact of memory request queues in the memory controller and row buffers in memory devices, the LLC miss penalty is mainly attributed to a fixed memory access latency. A higher hit rate of LLC implies less memory requests to the main memory, and thus less total memory access delay during the execution of applications. As a result, the LLC hit rate determines the total data traffic between processors and main memory.

There are also several optimizations on LRU-based cache replacement, such as rereference interval prediction (RRIP), dead block prediction, and cache partitioning. Dynamic insertion policy (DIP) [27] improves the performance of LRU replacement policy by dynamically changing the insertion policy according to the cache access patterns. RRIP [28] further improves the DIP performance in mixed memory access patterns. ReMAP [35] takes into account the reuse distance, block access recency, and memory access cost, and gives them different weights to calculate the priority of each data block. The data blocks with the lowest priority are replaced preferentially. Some other work tries to reduce the miss rate of LLC by preferentially replacing the dead blocks in the cache [15], [36], because these blocks are unlikely rereferenced before they are evicted. Partition-based replacement [7], [37]–[39] policies classify the LLC blocks into several partitions. Those partitions correspond to different memory access behaviors, and are replaced separately.

Although previous LRU-based cache replacement policies [27]–[32] can achieve a very high cache hit rate, they do not effective anymore in flat-addressable hybrid memory architectures. Upon a cache miss, the data block

may come from DRAM or NVM. To fetch a cache line from NVM, the access latency is much higher than that of DRAM. Therefore, the amount of memory references is not the only factor in determining the LLC efficiency. Instead, the heterogeneous memory access latencies should be taken into account. In Section III, we show that LRU-based cache replacement policies do not work well for hybrid memories with asymmetrical access latencies.

2) Cache Replacement for Hybrid Memory Systems: There have been some studies on extending the traditional LLC replacement policies to adapt to hybrid memory systems. Writeback-aware dynamic cache (WADE) [38] is a partition-based cache management policy for NVM main memory. Due to the high latency and energy consumption when blocks are evicted to NVM main memory, WADE commits to mitigate performance degradation by reducing the write-back requests. WADE dynamically partition the LLC into frequent writeback list and infrequent writeback list, and highly used dirty blocks in frequent writeback list are kept in LLC. Thus, WADE can reduce total data requests to the NVM main memory, and achieve a higher cache hit rate for the frequent writeback list.

However, WADE does not work well in hybrid memory systems. First, there are four kinds of memory accesses in the LLC: 1) DRAM read; 2) DRAM write; 3) NVM read; and 4) NVM write. Correspondingly, four logic lists should be maintained in LLC, making LLC management more complicated. Second, in most applications, write operations are not in the critical path. Memory buses can be only driven in write mode or read mode at a time [40]. Write requests are serviced only when the amount of write requests has reached a given threshold. Therefore, WADE does not perform well in these cases when write operations are trial. Finally, because the performance gap between NVM and DRAM is much larger than the performance gap between NVM reads and NVM writes, it is more beneficial to keep NVM blocks rather than DRAM blocks in LLC.

HAP [7] is the most similar work to our approach for LLC management in hybrid memory systems. HAP logically divides the LLC into NVM and DRAM partitions, which are used to cache data blocks from NVM and DRAM, respectively. HAP argues that there should be an optimal range of NVM partition size. Beyond this range, the system performance would become poor. Therefore, HAP maps memory requests of a sampling set in different processing units with different thresholds of the NVM range. HAP figures out the range of NVM partition size in which the system has the minimum memory access overhead, and set the range of NVM partition in the next epoch. However, in HAP, the NVM partition size is only selected from five ranges. The result is just a local optimization. How to determine the number and range of each processing unit should be further discussed. Moreover, when the number of NVM blocks is kept within the appropriate range, HAP would evict the cache line at the LRU position upon an LLC miss. This eviction scheme without distinguishing the type of the evicted block may allow the number of NVM cache lines beyond the appropriate range.

DARP [41] is another similar work to MALRU. MALRU can be differentiated from DARP in cache filling and eviction strategies. MALRU always inserts a new cache block at the MRU position, while the position of new blocks inserted by DARP is dynamic. DARP always evicts cache blocks at the LRU location, while MALRU always evicts the first DRAM block in the victim section. The cache partition policies in MALRU and DARP are also different. DARP only limits the LLC space simply for all DRAM blocks, without considering the access frequency of DRAM blocks. In contrast, MALRU only stores cache blocks with low access frequency in the victim section. Moreover, DARP may not fully and efficiently use the scarce LLC. When all cache blocks are fetched from DRAM, the position for filling DRAM blocks in a cache set is still fixed, allowing a portion of cache lines unused. Therefore, DARP may inefficiently use the scarce LLC resource. In contrast, MALRU can be adaptively transformed into the LRU algorithm, and thus is compatible with both homogeneous and heterogeneous memory subsystems.

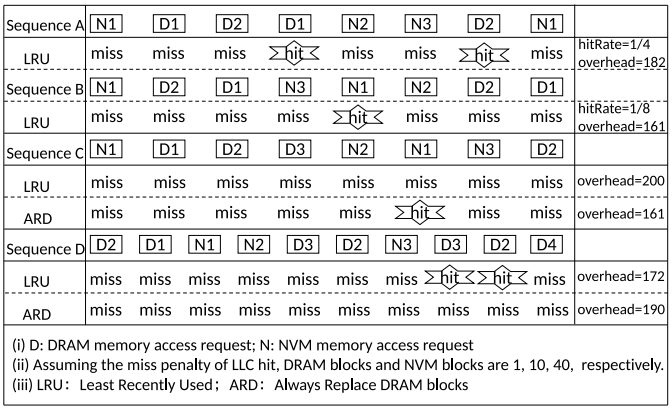


Fig. 3. LRU-based cache replacement policy is not effective in hybrid memory systems.

There are also some studies on the cache designs over hybrid hard disk drives [33], [34], which combines slow mechanical hard disk, fast NAND flash memory and DRAM in the same drive. Both DRAM and flash act as a cache of the disk, named as DRAM cache and NVC (NVM cache), respectively. Upon a DRAM cache miss, the missed data may be hit in the unpinned region of NVC or disk. In this case, the storage architecture of hybrid disk [33], [34] is somewhat similar to the cache/memory architecture proposed in this article. However, the cache replacement policy in [33] is significantly different from MALRU. The previous work [33] use two LRU lists (active and inactive) to manage the DRAM cache, and the NVC is also managed with the traditional LRU policy. Its cache replacement policy does not consider the heterogeneity of the underlying storage medium. In contrast, MALRU mainly considers different performance characteristics of DRAM and NVM to replace data blocks in LLC. Without considering the hardware implementation details, we think the policies proposed in MALRU is also applicable for the hybrid hard disk [33].

III. Motivation and Design Heuristics

In hybrid memory systems, the cache hit rate and MPKI are no longer effective metrics for evaluating the cache

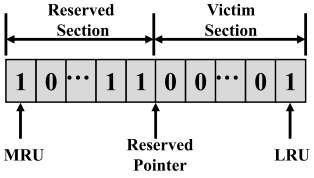


Fig. 4. Layout of MALRU-managed LLC sets.

performance due to the asymmetrical cache miss penalty of heterogeneous memories. In this section, we use an example to illustrate the inefficiency of the LRU-based cache replacement algorithm in hybrid memory systems.

To better understand the reason that LRU performs inefficiently in hybrid memory systems, we illustrate the behavior of the LRU algorithm in Fig. 3. Let N_i denote the address of an NVM cache line, D_i denote the address of a DRAM cache line. Sequence A and sequence B represent two different memory access sequences that mapped to the same cache set.

Assume the LLC cache set has four entries, and the LLC hit overhead, DRAM block miss penalty, and NVM block miss penalty are 1, 10, and 40, respectively. Applying the LRU replacement policy to sequence A and sequence B, the cache hit rates are 1/4 and 1/8, respectively. Although the hit rate of sequence A is greater than that of sequence B, the total memory access overhead of sequence A (182) is larger than that of sequence B (161).

Obviously, in hybrid memory, MPKI and hit rate of LLC are no longer effective metrics for evaluating the cache miss penalty, and LRU is not an efficient cache replacement algorithm either. To manage LLC more efficiently in hybrid memory systems, we should consider not only the data locality but also the asymmetry of the cache miss penalty. We thus advocate AMAT, a more general metric to assess the cache performance of hybrid memory systems.

Due to the expensive penalty of NVM block misses, an intuitive idea to improve LLC performance is reducing the miss of NVM blocks. In this guidance scheme, we always preferentially evict DRAM blocks from LLC and give NVM blocks more opportunities to stay in LLC.

On a cache miss, the policy that always replaces DRAM (ARD) blocks selects the first DRAM block from the LRU position to the MRU position, and the selected DRAM block is evicted. If there is no any DRAM block available, an NVM block at the LRU position is evicted. We use sequence C to verify this method and find that the total memory access overhead with ARD policy (161) is smaller than that with the LRU policy (200). Namely, ARD achieves better performance by improving the hit rate of NVM blocks, as shown in Fig. 3. Therefore, we have Heuristic 1 to improve cache performance in hybrid memory systems.

Heuristic 1: It is more beneficial to evict low-latency DRAM blocks than to evict NVM blocks from LLC.

Unfortunately, ARD cannot always perform better than LRU policy in hybrid memory systems. If DRAM blocks with temporal locality are evicted frequently, ARD can cause cache thrashing. Sequence D in Fig. 3 demonstrates such a scenario. The total memory access overhead of sequence D with ARD policy (190) is larger than that of LRU policy (172). ARD increases the memory access overhead as it evicts the frequently accessed DRAM blocks (block D2 and D3). Some blocks with good temporal locality will be rereferenced in a near-immediate interval. We call them near-immediate rereferenced blocks. These blocks also should be kept in LLC to avoid cache thrashing. Thus, we have Heuristic 2 to improve cache performance in hybrid memory systems.

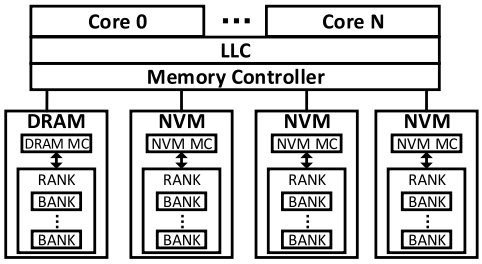


Fig. 5. Hybrid memory architecture in MALRU.

Heuristic 2: Near-immediate rereferenced DRAM blocks are valuable for cache hit rate and should be not evicted from LLC.

Based on the two heuristics and cache RRIP, we propose MALRU, an efficient cache replacement policy that keeps NVM blocks and near-immediate rereferenced DRAM blocks in LLC as more as possible.

IV. Miss Penalty-Aware Cache Replacement

Fig. 4 shows the layout of LLC that is managed by MALRU. MALRU divides LLC into a reserved section and a victim section through the reserved pointer. Low latency DRAM blocks with poor locality (larger reuse distance) are located in the victim section. These low latency blocks will be preferentially replaced, and thus NVM blocks have more opportunities to stay in LLC. NVM blocks and some DRAM blocks with very short reuse distances are kept in the reserved section.

On a cache miss, MALRU evicts data blocks from the LRU position to the reserved pointer. If there is no DRAM block available in the victim section, the whole LLC degenerates to the traditional LRU stack. MALRU has to evict NVM blocks in the LRU position.

A. Data Distribution in Hybrid Memories

Fig. 5 shows a typical architecture of hybrid memory systems. The main memory system contains one DRAM channel and three NVM channels. In our hybrid memory simulator, the address mapping scheme in MALRU is decoded as "subarray:row:rank:bank:channel:column" to improve memory level parallelism. This allows more even distribution of memory accesses across channels. This mapping scheme is also used by many hardware architectures, such as Intel Ivy Bridge architecture [42].

We apply this channel-interleaving address mapping scheme to our hybrid memory architecture, and data are placed on the four channels evenly in the granularity of page. Our memory access statistics also show that the memory access counts between different channels vary slightly.

To distinguish low-latency DRAM blocks and high-latency NVM blocks, MALRU adds a latency flag to reflect different

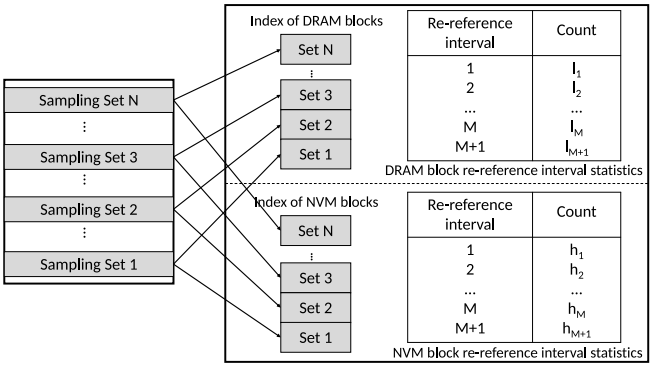


Fig. 6. Sketch of obtaining the rereference interval distribution of DRAM access sequence and NVM access sequence.

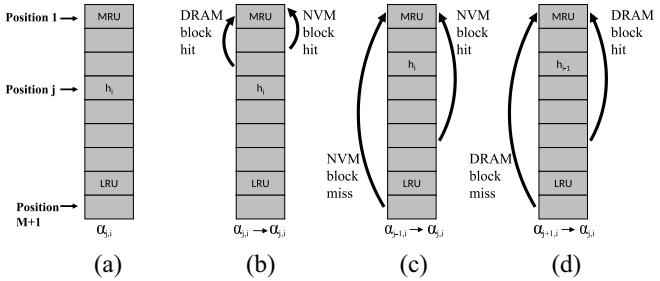


Fig. 7. State transition diagram. (a) $\alpha_{j,i}$. (b) $\alpha_{j,i}$ transfers to $\alpha_{j,i}$. (c) $\alpha_{j-1,i}$ transfers to $\alpha_{j,i}$. (d) $\alpha_{j-1,i-1}$ transfers to $\alpha_{j,i}$.

storage medium using one bit in each cache line. The flag of a high-latency NVM block is set to 1, otherwise, the flag is set to 0. We can simply check the physical memory addresses to identify whether a data block is from DRAMs or NVMs.

B. Rereference Interval Distribution

Rereference interval (or reuse distance) [28] of a cache block is the number of distinct memory accesses between the two sequential accesses to the same data block. Rereference interval is an important metric to evaluate data locality in LLC. A cache line is missed if its rereference interval is bigger than the total number of cache lines in a set of LLC. Otherwise, the requested cache line is hit.

Assume the total number of cache lines in a set of LLC is M. On a cache miss, the rereference interval of the cache line is set to $M+1$ uniformly in this article. As a result, a cache block's rereference interval ranges from 1 to $M+1$. If a cache block is hit in LLC, the position of the cache block in the LRU chain is exactly the rereference interval of this memory access.

Fig. 6 illustrates the process of rereference interval statistics in MALRU. As sampling a small number of sets is sufficient to reflect the cache behaviors [43], MALRU only samples (1/64) of the total sets to calculate the rereference interval of DRAM blocks and NVM blocks, respectively. For each sampled cache set, MALRU uses 31 bits (a major portion of tag) to index the NVM and DRAM blocks. DRAM accesses and NVM accesses are mapped to the DRAM blocks and NVM blocks, respectively. As a result, the hybrid memory access sequence is logically divided into a DRAM access sequence and an NVM access sequence. We adopt the MALRU replacement policy to each sampled set of LLC, and adopt LRU replacement policy to the sampled sets of DRAM and NVM access sequences. MALRU also maintains two tables to record the rereference interval distribution of DRAM and NVM access sequences, as shown in Fig. 6.

C. Determine the Boundary of the Reserved Section

As the LLC hit rate is not an effective metric in hybrid memory environments, we advocate AMAT to assess the LLC performance, as illustrated in (1). Let P_d and P_n be the proportion of DRAM and NVM accesses, respectively. Let HitRate_d and HitRate_n represent the hit rate of DRAM blocks and NVM blocks in the LLC, respectively. Let T_d and T_n denotes the average access latency of DRAM and NVM, respectively. The T_{LLC} denotes the latency of LLC hits. We have the following equation:

$$\begin{aligned}\text{AMAT} = {} & P_n * (T_{\text{LLC}} + T_n * (1 - \text{HitRate}_n)) \\ & + P_d * (T_{\text{LLC}} + T_d * (1 - \text{HitRate}_d)). \end{aligned} \tag{1}$$

Since each cache block is either a DRAM block or an NVM block, we have

$$P_n + P_d = 1. \tag{2}$$

In the table of rereference interval statistics, l_m denotes the total number of DRAM block accesses whose rereference intervals are m in a given sampling epoch. Similarly, h_m corresponds to the total number of NVM block accesses with rereference interval m. To calculate HitRate_n and HiteRate_d, we introduce variables $\alpha_{j,i}$ and $\beta_{j,i}$. $\alpha_{j,i}$ represents the probability of the jth position of whole LRU stack is exactly the ith DRAM block (as shown in Fig. 7), and $\beta_{j,i}$ represents the probability of the jth position of whole LRU stack is exactly the ith NVM block.

Assume the associative set of the LLC is M and the access sequence is a pure DRAM access sequence. If these DRAM blocks are handled separately, the hit rate of the pure DRAM access sequence *PD_HitRate* is the sum of the probabilities of all rereference intervals from 1 to M, namely, the total hit rate of the individual DRAM block sequence is

$$PD\_HitRate = \sum_{i=1}^{M} P(x = i) \tag{3}$$

where $P(x = i)$ is the hit rate of DRAM block with rereference interval i in the DRAM access sequence. It can be easily calculated from the distribution table of the DRAM block rereference interval.

Let $HD\_HitRate(x = i)$ denote the hit rate of the DRAM block with rereference interval i in the hybrid memory access sequence. The ith DRAM block ($HD\_HitRate(x = i)$) can be found from the ith position to the Mth position. Let $P_n(x = i)$ and $P_d(x = i)$ denote the probability of the ith position in the whole LRU stack is an NVM block and a DRAM block, respectively. For example, the probability of rereference interval i ($i \leq M$) of the DRAM block is the sum of probability on each position the ith DRAM block can be found.

$P_d(x = i)$ can be derived from the rereference interval distribution tables. Finally, we can induce the hit rate of DRAM blocks in the hybrid memory access sequence

$$\begin{aligned}\text{HitRate}_d &= \sum_{i=1}^{M} HD\_HitRate(x = i) \\ &= \sum_{i=1}^{M}\left(\sum_{j=i}^{M}\alpha_{j,i} * P_d(x = i)\right) \end{aligned} \tag{4}$$

where $\alpha_{j,i}$ represents the probability of a cache block in jth position of LRU stack is exactly the ith DRAM block.

Similarly, the hit rate of NVM blocks in the hybrid memory access sequence can be derived as

$$\text{HitRate}_n = \sum_{i=1}^{M}\left(\sum_{j=i}^{M}\beta_{j,i} * P_n(x = i)\right). \tag{5}$$

Fig. 7(a) shows the state of $\alpha_{j,i}$, and Fig. 7(b)–(d) shows the three states that can be transferred to state (a). If the current state is $\alpha_{j,i}$, the next state is still $\alpha_{j,i}$ when a block between position j and MRU is rereferenced, as shown in Fig. 7(b). In Fig. 7(c), the current state is $\alpha_{j-1,i}$, an NVM block miss or an NVM block hit between position $j-1$ and $M+1$ will transfer the state to $\alpha_{j,i}$. In Fig. 7(d), the current state is $\alpha_{j-1,i-1}$, namely, the $(j-1)$th position of the LRU stack is the $(i-1)$th DRAM block, a DRAM block miss or a DRAM block hit between position $j-1$ and $M+1$ will transfer the state to $\alpha_{j,i}$. The probabilities of the three state transition are

$$\alpha_{j,i} * \left(P_d * \sum_{k=1}^{i-1} P_d(x = k) + P_n * \sum_{k=1}^{j-i} P_n(x = k)\right) \tag{6}$$

$$\alpha_{j-1,i} * P_n * \sum_{k=j-i}^{M+1} P_n(x = k) \tag{7}$$

$$\alpha_{j-1,i-1} * P_d * \sum_{k=i}^{M+1} P_d(x = k). \tag{8}$$

The sum of these three portions should equal to 1, and then we get the value of $\alpha_{j,i}$

$$\alpha_{j,i} = \begin{cases} \frac{\alpha_{j-1,\ i} * P_n * \sum_{k=j-i}^{M+1} P_n(x=k) + \alpha_{j-1,\ i-1} * P_d * \sum_{k=i}^{M+1} P_d(x=k)}{1 - P_d * \sum_{k=1}^{i-1} P_d(x=k) - P_n * \sum_{k=1}^{j-i} P_n(x=k)} & j \le n+1 \\ 0, & j > n+1 \end{cases} \tag{9}$$

where n is the beginning position of the reserved section in MALRU. In the ideal case, no DRAM blocks exist in the victim section. Therefore, $\alpha_{j,i}$ equals to 0 when j is larger than the reserved pointer. Similarly, we can deduce $\beta_{j,i}$ based on the two rereference interval distribution tables.

We determine the position of the reserved pointer by calculating AMAT from 1 to M, and the position in which AMAT achieves the minimum value is the reserved pointer in the next epoch. Algorithm 1 shows the pseudo-code of determining the boundary of two sections. minAMAT and *boundaryPos* represent the minimum AMAT and the position of the reserved pointer, respectively. We set the reserved pointer from position 0 to *assoc-1* and calculate the AMAT (currentAMAT) in each position, where *assoc* represents the number of lines in a cache set. There is no need to generate a total order of the entries in LLC sets. Algorithm 1 just needs to return the position where AMAT gets the minimum value (minAMAT). The victim section ranges from the entry of the LRU position to the entry with the minimum AMAT.

Algorithm 1 Calculating the Position of the Reversed Pointer

Output: *boundaryPos*, the position of the reserved pointer in the next epoch

```
minAMAT, boundaryPos = 0;
for i = 0; i < assoc; i++ do
    /*Calculating the AMAT in each position on all sampling sets*/
    currentAMAT = CalAMAT(i);
    if i == 0 then
        minAMAT = currentAMAT;
        boundaryPos = 0;
    if currentAMAT <= minAMAT then
        minAMAT = currentAMAT;
        boundaryPos = i;
return boundaryPos;
```

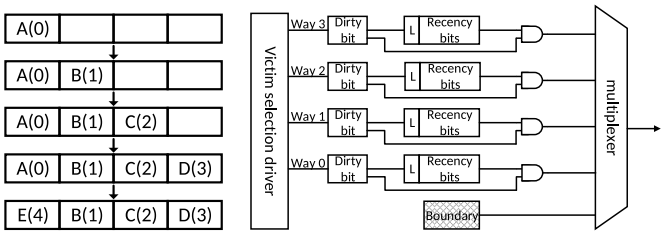


Fig. 8. Recency bits updating in MALRU. (a) Recency bits updating. (b) Logical circuit of finding victim blocks.

The major computation overhead is caused by sampling the LLC and determining the boundary of the reserved section for each cache set periodically.

D. Selection of Victim Blocks

When a new block is filled into the LLC, a block may be evicted from the victim section. Therefore, MALRU must figure out the position of victim block in the LLC. Normally, it is determined by the "recency bits" of each block [44]. Fig. 8(a) shows the processing of updating the recency bits in a memory access sequence $\langle A, B, C, D, E\rangle$. When a block hits in the LLC, its recency bit value is updated by adding one to the maximum recency bit in the set. Namely, in an LLC stack, the block with the minimum and maximum recency bit values are at the LRU position and the MRU position, respectively. Fig. 8(b) shows the logical circuit diagram of finding a given block. For the LRU policy, the multiplexer selects the blocks with minimal and maximal recency bits at the LRU and MRU positions, respectively. We modify this circuit to adapt to the MALRU policy. Taking the latency flag, recency bit and section boundary pointer as inputs, the multiplexer is able to find the first DRAM cache block in the victim section. Therefore, there is no need to sort blocks in a set according to their recency.

Algorithm 2 shows the pseudo-code of finding a victim block in hybrid memories with MALRU policy. The notations

Algorithm 2 Finding the Victim Block

Input: *addr*, the address of a memory request
Output: *setIndex*, the reference of the victim cache line

```
/*Find the index of cache set that the address mapped in LLC*/
set = extractSet(addr);
/*setIndex is used to find the block with the minimum recency value*/
/*Initialize currentRecency, currentRecency is the recency value the block
setIndex points to*/
currentRecency = 0;
/*Initialize the block in the LRU position as the victim block*/
setIndex = findMinRecency(set, currentRecency);
currentRecency = getRecencyValue(set, setIndex);
if !isDRAMType(set, setIndex) then
    way = assoc − 2;
    while way > boundary do
        setIndex = findMinRecency(set, currentRecency);
        currentRecency = getRecencyValue(set, setIndex);
        if isDRAMType(set, setIndex) then
            break;
        way − −;
    if way == boundary then
        setIndex = findMinRecency(set, 0);
return setIndex;
```

assoc and *boundary* represent the number of cache lines in a cache set and the position of the reserved pointer in the LRU stack, respectively. MALRU tries to find the first DRAM block from the LRU position (*assoc-1*) to the position of the reserved pointer (*boundary*). If no DRAM block is found in the victim section, MALRU evicts the block in the LRU position as all blocks in the victim section are NVM blocks. Blocks in the reserved section (position *0* to the *boundary*) will not be replaced to avoid cache thrashing.

We note that MALRU would first evict DRAM blocks prior to NVM blocks. However, as most LLCs are usually configured as write-back mode, dirty cache blocks are not written back to main memory immediately. Upon a cache eviction, a dirty block should be written back to main memory, while a clean block only needs to be invalidated in the LLC. It takes much more time to evict a clean block than to evict a dirty block. Intuitively, MALRU should evict clean blocks prior to dirty blocks. There are mainly two orders for evicting different cache blocks: 1) *MALRU-DDNN:* DRAM clean block > DRAM dirty block > NVM clean block > NVM dirty block and 2) *MALRU-DNDN:* DRAM clean block > NVM clean block > DRAM dirty block > NVM dirty block. We compare the performance of MALRU when using the two different orders for evicting cache blocks. Our experimental results show that it is more beneficial to always evict clean blocks first (MALRU-DNDN) than MALRU-DDNN.

The LRU algorithm needs to maintain a total order relation between all valid cache lines in a cache set. The space overhead for tracking the LRU states is $(n + 1) \times \log(n)$, where n represents the cache associativity [45]. Due to the hardware complexity of LRU in implementation, most on-chip caches use simpler algorithms to approximate the LRU algorithm, such as not recently used (NRU) [28], [46] and Clock [21], [47]. MALRU can be easily implemented with those LRU-based on-chip cache replacement algorithms.

Fig. 9 shows the implementations of our MALRU policy when it is integrated with NRU and Clock algorithms. In both

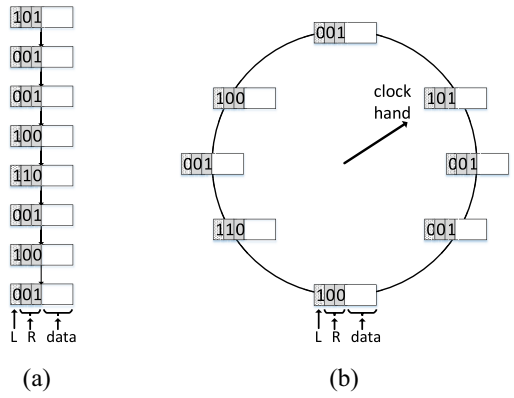


Fig. 9. MALRU is integrated with (a) NRU and (b) clock algorithms.

two extended algorithms, we add a one-bit latency flag (L) to distinguish cache blocks between DRAM and NVM, and extend the one-bit reference flag (R) in the vanilla NRU and Clock algorithms to two bits. The R flag is initialized as 0 and set to 1 when the block is accessed. Once a block is selected as the boundary of the victim section and reserved section, this block's reference flag is set to 2. In this way, the two-bit reference flag can reflect the boundary of the two sections, and thus the reserved pointer in our original design is not required. NRU algorithm always selects victim blocks at a fixed position (for example, the header of LLC sets), and thus the LRU pointer in MALRU is not required. The Clock algorithm starts the selection of victim blocks from the clock hand, and thus it can be treated as the LRU pointer in MALRU. Therefore, in both NRU and Clock algorithms, the victim section ranges from the header of LLC sets or the clock hand to the section boundary (R flag equals to 2).

For other LRU-based page cache replacement algorithms, such as the adaptive replacement cache algorithm (ARC) [48], they can be also integrated with MALRU. However, because it requires four LRU lists to track both recently used pages and least frequently used pages, and their corresponding eviction history for both lists. The hardware overhead of ARC is even 4 times higher than the LRU algorithm [45]. We need to make a tradeoff between the benefit and the space overhead when applying ARC to the onchip LLC.

E. MALRU Complexity Reduction

In each time slot, MALRU should calculate the size of the reserved section to minimize the AMAT. This requires to calculate $\alpha_{j,i}$ and $\beta_{j,i}$ iteratively. Although (9) seems a little complex, the computation complexity of determining the position of the reserved pointer is just $O(M^2)$. Because the parameter M is the number of lines in a cache set (typically 16 and 20), the computation overhead is acceptable. However, because MALRU should periodically sample the LLC and adjust the reserved pointer at runtime, the cumulative performance overhead is not trivial when using a software implementation.

On the other hand, cache replacement algorithms should also take into account the complexity of hardware implementation. A new cache replacement policy in hybrid memories

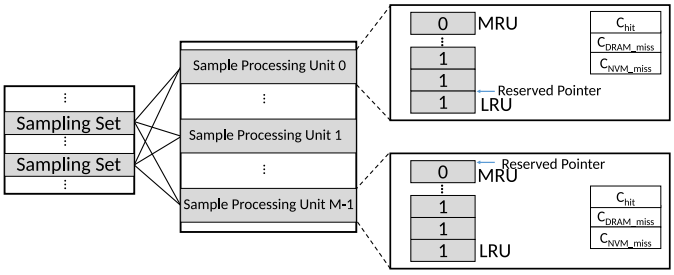


Fig. 10. MALRU-CR policy for estimating LLC hit rate, miss rate of DRAM blocks and NVM blocks.

should make less intrusive hardware modifications to the current cache design. Because the hardware overhead of MALRU is higher than LRU, it is essential to reduce the computation complexity. Therefore, we seek to simplify the MALRU to minimize the hardware modifications while still following the basic design heuristics of MALRU.

Although the rereference distance table adopted in MALRU is able to provide more runtime memory access statistics, and thus describe data access behaviors of applications much better than the LRU, it results in higher hardware complexity and computation overhead. As a result, we abandon the rereference distance table and propose a simple yet effective sampling mechanism to estimate the LLC hit rate, miss rate of DRAM blocks and NVM blocks in the LLC.

Fig. 10 illustrates our new design for simplifying the MALRU policy, called MALRU complexity reduction (MALRU-CR). It follows the same design heuristics of MALRU but can minimize the modifications of current cache design. The reserved section and protected section are still reserved, and our goal is still to find an optimal partition to minimize the AMAT. For an M-way set associative LLC, the sampling sets is mapped to M sample processing units. These processing units can be marked as unit 0 to unit $M-1$. In unit i, the reserved pointer points to the ith cache line in the LRU stack. We only use three counters in each sample processing unit to record the hit counts of LLC, miss counts of DRAM blocks and miss counts of NVM blocks in each sampling epoch. At the end of sampling epoch, we get the AMAT of each unit by simply calculating the product of access latency and access counts for each kind of memory, as shown in

$$\text{AMAT} = T_{\text{LLC}} * \frac{C_{\text{hit}}}{C_{\text{access}}} + T_d * \frac{C_{\text{DRAM\_miss}}}{C_{\text{access}}} + T_n * \frac{C_{\text{NVM\_miss}}}{C_{\text{access}}} + E \quad (10)$$

where C_{hit}, $C_{\text{DRAM\_miss}}$, and $C_{\text{NVM\_miss}}$ represent the hit counts of LLC lines, miss counts of DRAM blocks and miss counts of NVM blocks, respectively. C_{access} is the total counts of sampled data blocks, and E reflects the additional memory access latency caused by the cache write-back policy, write queuing in memory controller and so on [49], [50]. E can be treated as a factor to calibrate the AMAT. However, when we try to find the minimum AMAT to determine the boundary of the victim section, we do not need to figure out the exact value of E because the comparison of any two AMAT would offset the same E in each sample processing unit.

TABLE II
SYSTEM CONFIGURATIONS AND WORKLOADS

Processor	1-core/4-core CMP, 2 GHz, out-of-order
L1 caches	32 KB I-caches, 64 KB D-caches 2-way associative, 64B cache line, 2-cycle latency
L2 caches	256 KB, 4-way associative 64B cache line, 10-cycle latency
L3(shared)	2 MB (4 MB for multi-programmed), write-back 64B cache line, 25-cycle latency, 16-way associative
Main memory	DRAM: 1 GB (1 channel), 150-cycle access latency, 4.14 and 4.53 pJ/bit read and write energy consumption. PCM: 3 GB (3 channel), 500 and 1000 cycles read and write latencies, respectively. 2.47 and 16.82 pJ/bit read and write energy consumption.
Workloads	SPEC CPU2006: bzip2, gcc , mcf, milc, soplex, omnetpp, hmmer mix1: (bzip2, gcc, omnetpp, hmmer) mix2: (mcf, lbm, cactusADM, gcc) mix3: (namd, astar, xalancbmk, soplex) mix4: (milc, gromacs, lbm, mcf)

At the end of each sampling interval, MALRU-CR gets the AMAT in each sample processing unit. The position where we get the minimum AMAT is the optimal reserved pointer in the next epoch. In practice, we find that only four sample processing units in a 16-way set associative LLC is able to achieve very good application performance. The reserved pointers of the four units point to the 0th, 4th, 8th, and 12th cache line of the LRU stack in each set, respectively. This sampling mechanism can significantly simplify the computation complexity of AMAT and hardware implementation.

V. EVALUATION

A. System Configuration

We conduct our experiments on an integrated simulator, Gem5 [51] and NVMain [52]. Gem5 simulates the processor and cache, while NVMain simulates the hybrid main memory. Although MALRU is applicable for a variety of hybrid memory systems with asymmetrical memory access latencies, it is more beneficial to apply our proposal to hybrid memory systems in which the performance gap between NVM and DRAM is significant. As a result, we use PCM as NVM main memory in our experiments. The DRAM performance parameters, DRAM energy parameters, and NVM energy parameters are referred to the DDR3-1333 and NVM chip modules in the NVMain simulator, respectively. The NVM performance parameters are referred to data in the public literature [15], [44], [53], [54]. It should be noted that there is a significant difference among the PCM parameters in these public studies, and we choose a reasonable range from them.

Table II shows the system configurations and experimental workloads. We use seven single-thread workloads and four multiprogrammed workloads from the SPEC CPU 2006. These workloads represent some typical memory access patterns of cache thrashing (mcf), recency-friendly (soplex and gcc), and irregular access (mix).

As data distribution in the hybrid memory system can significantly affect application performance, we use the channel-interleaving address mapping scheme in NVMain to guarantee

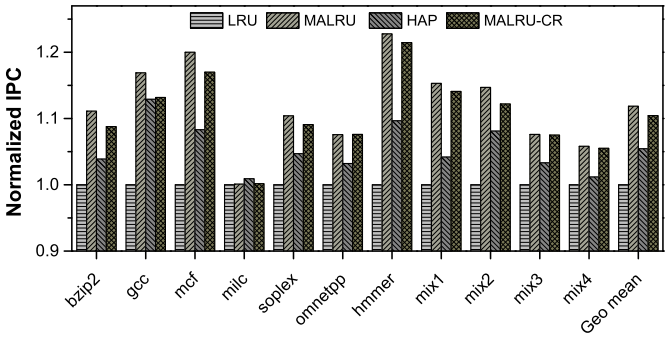


Fig. 11. Comparison of IPC among LRU, MALRU, HAP, and MALRU-CR.

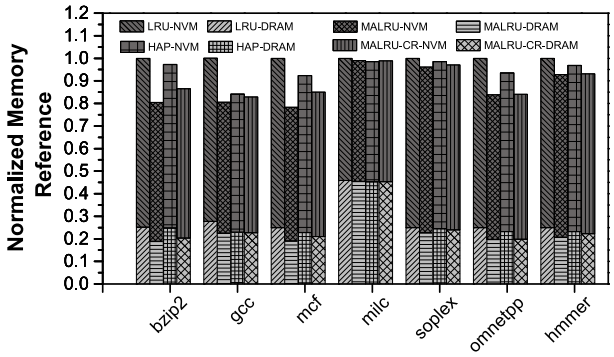


Fig. 12. Memory reference reduction using LRU, MALRU, HAP, and MALRU-CR.

an even data distribution across all memory channels. This setting is to make fair and stable comparisons in our experiments.

B. Performance Evaluation

We compare MALRU and MALRU-CR with two other cache replacement policies LRU and HAP. HAP divides the LLC into the NVM section and DRAM section logically and changes the size of the NVM section dynamically. We refer the application performance with the LRU algorithm as a baseline.

Fig. 11 shows the normalized instructions per cycle (IPC) for both single-thread and multiprogrammed workloads. Compared to HAP and LRU, MALRU improves IPC by 6.4% and 11.8% on average, respectively. For workloads *hmmer*, *mcf*, and *mix1*, compared to LRU, MALRU improves application performance by 22.8%, 20.5%, and 15.3%, respectively. MALRU-CR achieves similar performance to MALRU. The slight performance gap is due to a little accuracy degradation in predicting the reserved pointer by MALRU-CR.

We classify the workloads that benefit from MALRU into two categories: 1) cache thrashing and 2) recency-friendly access patterns. For the thrashing access pattern such as *mcf*, the working sets of these workloads are much larger than the cache size, and thus leads to cache thrashing. MALRU improves these workloads' performance by reducing the total cache miss penalty. As NVM blocks are preferentially kept in LLC and may be rereferenced before being evicted, the hit rate of NVM blocks is increased while the hit rate of DRAM blocks is not reduced. For recency-friendly access patterns such as *soplex*, MALRU evicts the DRAM blocks with the least probability of being accessed in the next epoch. MALRU increases the hit rate of most recently accessed DRAM and NVM blocks and thus improves system performance.

C. Reduction of Memory References

Fig. 12 shows the DRAM and NVM memory references under four cache replacement policies, all normalized to the LRU policy. For recency-friendly workload *gcc*, MALRU and HAP achieve 16.9% and 10.9% performance improvement, and reduce memory references by 20.3% and 15.8%, respectively. The gap between total memory reference reduction is smaller than the gap between performance improvement. This is because NVM blocks may occupy a majority of LLC space in our configuration, since the capacity of NVM is 3 times of DRAM. Upon a cache miss, MALRU would evict the first DRAM block from the victim section, while HAP has a more probability to evict an NVM block because the block in LRU position is evicted. Therefore, MALRU achieves higher performance improvement by keeping more NVM blocks in LLC most application data are located in the NVM. For workloads with long rereference intervals, MALRU leads to fewer memory accesses because both the DRAM and NVM blocks have a long rereference interval. Some NVM blocks are more likely hit because they have more chances to stay in LLC while DRAM blocks are replaced preferentially.

We find that there is no definite positive correlation between performance improvement and memory access reduction for different applications. For example, *gcc* and *hmmer* achieve 16.9% and 22.8% performance improvement while getting 20.3% and 7.17% memory reference reduction, respectively. The number of read requests is 4.43 times of the number of write requests in *gcc*, while the ratio of read requests to write requests is 1.07 in *hmmer*. Namely, a large proportion of NVM write requests are reduced by MALRU in *hmmer* than that in *gcc*. As NVM write latency is 6 times higher than that of DRAM read, the application performance can be significantly improved even when decreasing a very few NVM writes. This implies that the ratio of reads to writes on NVM has a significant impact on application performance.

D. Energy Consumption

Previous studies [15], [55] shows the main memory consumes over 40% of total system energy consumption. A majority of memory energy consumption is attributed to the dynamic potion. In hybrid DRAM/NVM memory systems, the energy consumption can be reduced by decreasing the memory references, especially, for NVM write operations. Fig. 13 presents the energy consumption of hybrid memories by adopting the four policies. Compared to LRU policy, MALRU, MALRU-CR, and HAP can reduce energy consumption by 10.9%, 4.1%, and 8.8% on average, respectively. The reason behind this is that MALRU and MALRU-CR are able to provide NVM blocks more opportunities to remain in LLC, and thus reduce write operations to the NVM. As NVM writes consume approximate 7 times higher energy than NVM reads, an application can reduce more memory energy consumption if the proportion of NVM writes is significantly reduced.

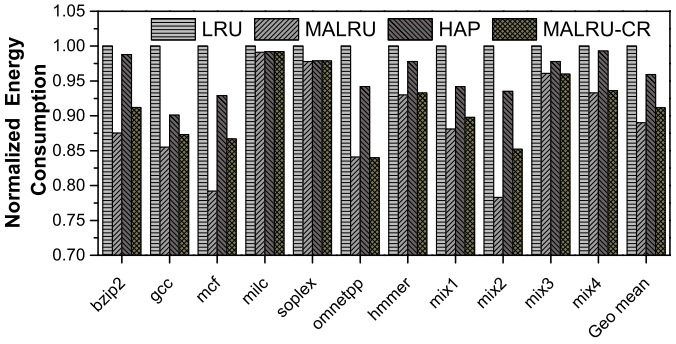


Fig. 13. Total energy consumption using LRU, MALRU, HAP, and MALRU-CR.

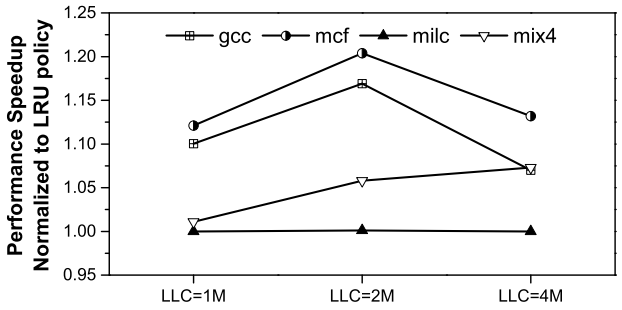


Fig. 14. MALRU sensitivity to cache size.

MALRU is able to significantly reduce the energy consumption of *mcf* by 22% because *mcf* demonstrates a cache-thrashing memory access pattern, and MALRU can also reduce the memory references of *mcf* by almost an equivalent degree, as shown in Fig. 12. Since milc is a data streaming program, there is very little opportunity for data reuse in LLC. As a result, MALRU, HAP, and MALRU-CR all lead to a slight reduction of memory references compared to LRU, as shown in Fig. 12. Correspondingly, there is a trial reduction of energy consumption, as shown in Fig. 13.

E. Sensitivity Studies

1) Sensitivity to Cache Size: Fig. 14 presents the system performance improvement of MALRU with different cache sizes. For workloads with short rereference interval and good temporal locality, such as *gcc*, the competition between DRAM blocks and NVM blocks is mitigated as the cache size increases. Hence, MALRU performs more similar as LRU when the LLC capacity becomes larger.

For workloads with the thrashing access pattern, in which the cache size is larger than the working set, the workload tends to be a recency-friendly workload. Workload *mcf* has a knee in the working set which is a litter larger than 2 MB, as demonstrated in [28]. Therefore, the performance of *mcf* is enhanced at the 2 MB LLC size as more NVM blocks hit. When the size of LLC increases to 4 MB, *mcf* tends to be a recency-friendly workload as the working set is smaller than the cache size, LRU achieves more performance improvement than MALRU.

2) Sensitivity to NVM Proportion: Fig. 15 exhibits the MALRU performance varying with the proportion of NVM in the main memory, all normalized to the LRU policy. When the

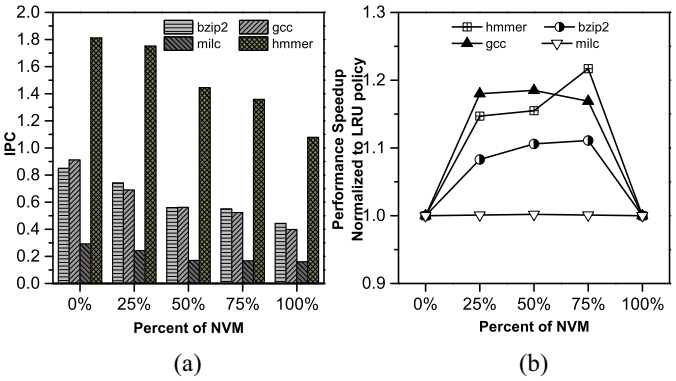


Fig. 15. MALRU sensitivity to the proportion of NVM in main memory. (a) Application IPC. (b) Performance speedup.

proportion of NVM increases, the IPC of the MALRU system decreases, as shown in Fig. 15(a). However, the performance of MALRU relative to LRU does not necessarily decline, as shown in Fig. 15(b).

Applications with streaming access pattern such as *milc* show rather stable performance when the proportion of NVM in main memory increases. The reason is that most data is read-only once and the read latencies of NVM and DRAM are comparable. Both *hmmer* and *gcc* show recency-friendly memory access patterns. However, the trend of performance improvement for *hmmer* and *gcc* are adverse when the proportion of NVM increases to 75%. To figure out the reason, we account the read/write requests, and observe that the read/write ratio of *hmmer*, *gcc* are 1.07, and 4.43. When the proportion of NVM becomes larger, the competition among NVM blocks would become severe. If an application shows a larger proportion of writes such as *hmmer*, the LLC competition would mainly come from NVM writes and NVM reads, rather than NVM accesses and DRAM accesses. As the performance gap between NVM write and NVM read (2 times in Table II) is smaller than the gap between NVM access and DRAM access (3.3 times in Table II), the relative performance improvement of MALRU is reduced.

As a result, MALRU achieves higher performance improvement when abundant NVM blocks compete with DRAM blocks in LLC. When there is only NVM data or DRAM data in LLC, the MALRU degenerates to LRU, and the block in the LRU position is always evicted, and thus MALRU achieves equivalent performance to LRU.

3) Sensitivity to Hot Data Distribution: Fig. 16 shows the impact of hot data distribution on different cache replacement policies, all normalized to LRU. Here, we only show the results of migrating 25% of the top hot pages to DRAM. LRU-M and MALRU-M represent the policies applying data migration to LRU and MALRU, respectively.

Compared to LRU and MALRU, LRU-M and MALRU-M improve application IPC by up to 28.3% and 26.8% on average, respectively. We note that the memory monitoring and hot page migration overhead are not considered. Therefore, the real system can not achieve such a performance improvement. Compared to LRU-M, MALRU-M can still improve application performance by 10.6% on average, which is comparable

TABLE III
HARDWARE AND SOFTWARE OVERHEAD IN MALRU, HAP, AND MALRU-CR

Overhead /Policies	Storage Overhead			Computation Overhead	Energy Overhead
	Items	Overhead	Total Overhead		
MALRU	Latency flag	1 bit × # of cache line	4 KB	0.1881%	0.9%
	Re-reference interval counter	32 bit × 17 × 2	136 B		
	Index of sampled blocks	2 × 16 × 31 bit × # of set	3.875 KB		
	Reserved pointer	8 bit × 1	1 B		
HAP	Latency flag	1 bit × # of cache line	4 KB	0.0085%	1.28%
	Miss counter	32 bit × 2 × 5	40 B		
	Index of sampled blocks	5 × 16 × 31 bit × # of set	9.6875 KB		
	Partition threshold counter	8 bit × 2 × 6	12 B		
MALRU-CR	Latency flag	1 bit × # of cache line	4 KB	0.0068%	0.99%
	Access counter	32 bit × 3 × 4	48 B		
	Index of sampled blocks	4 × 16 × 31 bit × # of set	7.75 KB		
	Reserved pointer	8 bit × 5	5 B		

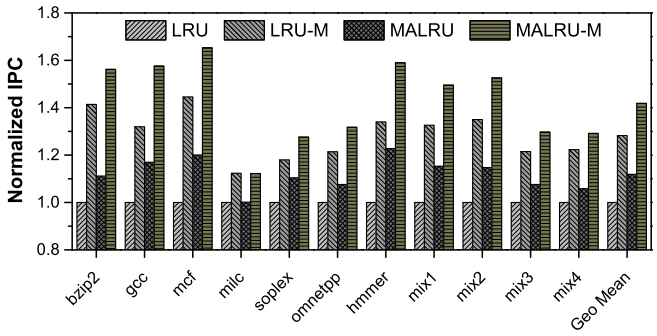


Fig. 16. MALRU sensitivity to hot data distribution.

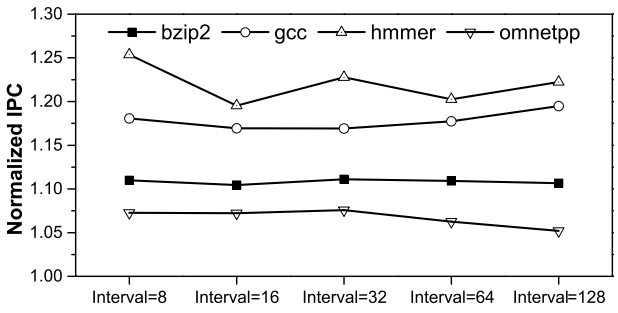


Fig. 17. MALRU sensitivity to sampling intervals.

to the improvement of MALRU against LRU. This implies MALRU is still effective when more hot data is distributed on fast DRAM. MALRU is complementary to the hot data migration policies in hybrid memory systems.

4) Sensitivity to Sampling Interval: Fig. 17 shows that the performance improved by MALRU varies with the sampling interval of cache sets, all normalized to LRU. For each experiment, we exponential increase the number of cache sets (interval) for sampling. We find that although there are slight fluctuations when the sampled data blocks decreases. MALRU shows rather stable performance even when the sampling interval increases from 8 to 128. To this end, we set 64 as a default sampling interval in our system.

F. Hardware and Software Overhead

Table III shows the storage overhead of MALRU, HAP, and MALRU-CR for a 2 MB 16-way set associative LLC with 64 byte cachelines. MALRU adds one bit of latency flag for each cacheline, and the total storage overhead is 4 KB. MALRU uses two tables to record the counts of different rereference intervals for DRAM and NVM blocks, as shown in Fig. 6. As LLC has 16 lines in each set, each distribution table of the rereference interval has 17 entries. The counter is reset in each sampling epoch (50 million instructions), so 32 bits are enough for storing the counter value. MALRU uses 31 bits to record the index of each sampled cacheline, and MALRU only samples (1/64) of the total sets (32 sets in this case). To indicate the reserved section in each cache set, we need 8 bits to store the reserved pointer. Overall, the additional storage overhead of MALRU is only 0.391% of the total capacity of LLC.

MALRU-CR is an efficient yet easy-to-implement simplification of MALRU. MALRU-CR uses four sampling units. Three counters are used to record the LLC hit counts, LLC miss counts of DRAM blocks, and LLC miss counts of NVM blocks. These counters can be implemented by special hardware circuits, and thus can be updated with the access to LLC in parallel. The storage overhead of MALRU-CR is 0.576% of the total capacity of LLC. For HAP, it should maintain five sampling units with different thresholds for NVM partitioning. The total storage overhead of HAP is 0.671%.

To determine the position of the reserved pointer, we search the minimum AMAT periodically. The computation complexity of MALRU is O(M^2). The performance overhead can be negligible as M is only a constant (16). Although the latency of MALRU and MALRU-CR cannot be fully hidden, we note that the computation of AMAT can be done by hardware and software cooperatively in the background, and thus the latency can be removed from the critical execution path of programs. To this end, we simulate the computation of AMAT by software to estimate the cost. We run the program 1000 times on a physical machine and measure the average time of AMAT computation. At last, we model the cost of AMAT computation in the simulator Gem5. We have shown the total overhead of the AMAT computation in Table III. MALRU, HAP, and MALRU-CR only spend 0.1881%, 0.0085%, and 0.0068% of application execution time, respectively.

MALRU adds some hardware for sampling and statistics, and thus leads to extra area and energy consumption. Since the simulator Gem5 can not model the cache energy consumption,

we use another tool CACTI [56] to calculate the cache energy overhead caused by MALRU. CACTI is a widely used analytical tool that can model the LLC. It takes a set of cache parameters as input and models its access latency, power, and area. Specifically, we add these additional hardware circuits to the cache tag. By modifying the size of the tag, we can calculate the energy cost of each cache replacement policy. Table III shows the energy overhead caused by the extra hardware circuit in the three policies. Compared to LRU, MALRU, HAP, and MALRU-CR only cause 0.9%, 1.28%, and 0.99% more energy consumption, respectively.

VI. Conclusion

The conventional LRU-based cache replacement policies are no longer effective in hybrid memory systems. In this article, we advocate a new cache performance metric—AMAT and a rereference interval and reserved section aware LRU-based cache replacement policy for hybrid memory systems. NVM blocks and partial frequently accessed DRAM blocks with good temporal locality are protected in the reserved section to reduce cache miss penalty. We compare MALRU with LRU and the state-of-the-art HAP policies using several workloads. The experimental results show that MALRU improves system performance over LRU and HAP by up to 22.8% and 13.1%, respectively, while incurring only 0.391% hardware overhead.

References

[1] O. Mutlu, "Memory scaling: A systems architecture perspective," in *Proc. IEEE Int. Memory Workshop (IMW)*, 2013, pp. 21–25.

[2] A. Foong and F. Hady, "Storage as fast as rest of the system," in *Proc. IEEE Int. Memory Workshop (IMW)*, 2016, pp. 1–4.

[3] J. Wang, X. Dong, Y. Xie, and N. P. Jouppi, "i^2WAP: Improving non-volatile cache lifetime by reducing inter- and intra-set write variations," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2013, pp. 234–245.

[4] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi, "Kiln: Closing the performance gap between systems with and without persistence support," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2013, pp. 421–432.

[5] R. Azevedo, J. D. Davis, K. Strauss, P. Gopalan, M. S. Manasse, and S. Yekhanin, "Zombie memory: Extending memory lifetime by reviving dead blocks," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2013, pp. 452–463.

[6] M. Chaudhuri, "Pseudo-LIFO: The foundation of a new family of replacement policies for last-level caches," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2009, pp. 401–412.

[7] W. Wei, D. Jiang, J. Xiong, and M. Chen, "HAP: Hybrid-memory-aware partition in shared last-level cache," in *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, 2014, pp. 28–35.

[8] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: A hybrid PRAM and DRAM main memory system," in *Proc. Design Autom. Conf. (DAC)*, 2009, pp. 664–669.

[9] L. E. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," in *Proc. Int. Conf. Supercomput. (ICS)*, 2011, pp. 85–95.

[10] Y.-T. Chen *et al.*, "Dynamically reconfigurable hybrid cache: An energy-efficient last-level cache design," in *Proc. Conf. Design Autom. Test Europe (DATE)*, 2012, pp. 45–50.

[11] *2016 Analyst Conference*. Accessed: May 1, 2019. [Online]. Available: http://files.shareholder.com/downloads/ABEA-45YXOQ/0x0x875021/4BEAA02E-BBC2-402C-A51D-B3B2C6B8C3D4/Winter_Analyst_Day_2016.pdf

[12] D. Knyaginin, G. N. Gaydadjiev, and P. Stenström, "Crystal: A design-time resource partitioning method for hybrid main memory," in *Proc. Int. Conf. Parallel Process. (ICPP)*, 2014, pp. 90–100.

[13] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen, "Exploiting program semantics to place data in hybrid memory," in *Proc. Int. Conf. Parallel Archit. Compilation (PACT)*, 2015, pp. 163–173.

[14] R. Salkhordeh and H. Asadi, "An operating system level data migration scheme in hybrid DRAM-NVM memory architecture," in *Proc. Conf. Design Autom. Test Europe (DATE)*, 2016, pp. 936–941.

[15] J. Ahn, S. Yoo, and K. Choi, "DASCA: Dead write prediction assisted STT-RAM cache architecture," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2014, pp. 25–36.

[16] Z. Wang, D. A. Jiménez, C. Xu, G. Sun, and Y. Xie, "Adaptive placement and migration policy for an STT-RAM-based hybrid cache," in *Proc. IEEE 20th Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2014, pp. 13–24.

[17] C.-H. Chen, P.-C. Hsiu, T.-W. Kuo, C.-L. Yang, and C.-Y. M. Wang, "Age-based PCM wear leveling with nearly zero search cost," in *Proc. Design Autom. Conf. (DAC)*, 2012, pp. 453–458.

[18] C.-Y. Su *et al.*, "HpMC: An energy-aware management system of multi-level memory architectures," in *Proc. Int. Symp. Memory Syst.*, 2015, pp. 167–178.

[19] J. Izraelevitz *et al.*, "Basic performance measurements of the Intel Optane DC persistent memory module," *CoRR*, vol. abs/1903.05714, pp. 1–60, Mar. 2019.

[20] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2009, pp. 2–13.

[21] S. Lee, H. Bahn, and S. H. Noh, "CLOCK-DWF: A write-history-aware page replacement algorithm for hybrid PCM and DRAM memory architectures," *IEEE Trans. Comput.*, vol. 63, no. 9, pp. 2187–2200, Sep. 2014.

[22] T. Hirofuchi and R. Takano, "RAMinate: Hypervisor-based virtualization for hybrid main memory systems," in *Proc. 7th ACM Symp. Cloud Comput.*, 2016, pp. 112–125.

[23] H. Zhang, N. Xiao, F. Liu, and Z. Chen, "Leader: Accelerating RERAM-based main memory by leveraging access latency discrepancy in crossbar arrays," in *Proc. Conf. Design Autom. Test Europe (DATE)*, 2016, pp. 756–761.

[24] A. A. Khan, F. Hameed, R. Blaesing, S. Parkin, and J. Castrillon, "ShiftsReduce: Minimizing shifts in racetrack memory 4.0," *CoRR*, vol. abs/1903.03597, pp. 1–27, Mar. 2019.

[25] S. Kim, G. W. Burr, W. Kim, and S.-W. Nam, "Phase-change memory cycling endurance," *Phase Change Mater. Electron. Photon.*, vol. 44, no. 9, pp. 710–714, 2019.

[26] F. T. Hady. *Intel Optane Technology Delivers New Levels of Endurance*. Accessed: Aug. 3, 2019. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-technology/delivering-new-levels-of-endurance-article-brief.html

[27] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2007, pp. 381–391.

[28] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2010, pp. 60–71.

[29] G. Kurian, S. Devadas, and O. Khan, "Locality-aware data replication in the last-level cache," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2014, pp. 1–12.

[30] D. A. Jeménez, "Insertion and promotion for tree-based Pseudolru last-level caches," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2013, pp. 284–296.

[31] N. Xiao, Y. Zhao, F. Liu, and Z. Chen, "Dual queues cache replacement algorithm based on sequentiality detection," *Sci. China Inf. Sci.*, vol. 55, no. 1, pp. 1–9, 2012.

[32] J. Zhang, M. Guo, C. Wu, and Y. Chen, "Toward multi-programmed workloads with different memory footprints: A self-adaptive last level cache scheduling scheme," *Sci. China Inf. Sci.*, vol. 61, no. 1, pp. 1–14, 2017.

[33] Y.-J. Kim, S.-J. Lee, K. Zhang, and J. Kim, "I/O performance optimization techniques for hybrid hard disk-based mobile consumer devices," *IEEE Trans. Consum. Electron.*, vol. 53, no. 4, pp. 1469–1476, Nov. 2007.

[34] T. Bisson and S. A. Brandt, "Flushing policies for NVCache enabled hard disks," in *Proc. IEEE Conf. Mass Storage Syst. Technol. (MSST)*, 2007, pp. 299–304.

[35] A. Arunkumar and C.-J. Wu, "ReMAP: Reuse and memory access cost aware eviction policy for last level cache management," in *Proc. IEEE Int. Conf. Comput. Design (ICCD)*, 2014, pp. 110–117.

[36] S. M. Khan, Y. Tian, and D. A. Jiménez, "Sampling dead block prediction for last-level caches," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2010, pp. 175–186.

[37] D. Sánchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *Proc. Int. Symp. Comput. Archit. (ISCA)*, 2011, pp. 57–68.

[38] Z. Wang *et al.*, "WADE: Writeback-aware dynamic cache management for NVM-based main memory system," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 51, pp. 1–21, 2013.
[39] Intel. (2015). *Improving Real-Time Performance by Utilizing Cache Allocation Technology*. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf
[40] J. Zhao, O. Mutlu, and Y. Xie, "FIRM: Fair and high-performance memory control for persistent memory systems," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2014, pp. 153–165.
[41] G. Jia, G. Han, J. Jiang, and L. Liu, "Dynamic adaptive replacement policy in shared last-level cache of DRAM/PCM hybrid memory for big data storage," *IEEE Trans. Ind. Informat.*, vol. 13, no. 4, pp. 1951–1960, Aug. 2017.
[42] *Power Management of the Third Generation Intel Core Micro Architecture Formerly Codenamed IVY Bridge*. Accessed: May 1, 2019. [Online]. Available: http://www.hotchips.org/wp-content/uploads/hc_archives/hc24/HC24–1-Microprocessor/HC24.28.117-HotChips_IvyBridge_Power_04.pdf
[43] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. Int. Symp. Microarchit. (MICRO)*, 2006, pp. 423–432.
[44] *NVM Technologies*. Accessed: May 1, 2019. [Online]. Available: http://research.cs.wisc.edu/sonar/tutorial/01-technology.pdf
[45] A. Ooka, S. Eum, S. Ata, and M. Murata, "Compact CAR: Low-overhead cache replacement policy for an ICN router," *IEICE Trans. Commun.*, vol. 101, no. 6, pp. 1366–1378, 2018.
[46] H. T. W. Paper. (2002). *Inside the Intel Itanium 2 Processor*. [Online]. Available:http://research.cs.wisc.edu/sonar/tutorial/01-technology.pdf
[47] A. S. Tanenbaum and H. Bos, *Modern Operating Systems*, 2nd ed. Vancouver, BC, Canada: Prentice-Hall, 2001, p. 218.
[48] N. Megiddo and D. S. Modha, "Outperforming LRU with an adaptive replacement cache algorithm," *Computer*, vol. 37, no. 4, pp. 58–65, Apr. 2004.
[49] *Cache Replacement Policies*. Accessed: Aug. 3, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Cache_replacement_policies
[50] A. J. Smith, "Design of CPU cache memories," in *Proc. IEEE TENCON*, 1987, pp. 1–10.
[51] N. Binkert *et al.*, "The Gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.
[52] M. Poremba and Y. Xie, "NVMain: An architectural-level main memory simulator for emerging non-volatile memories," in *Proc. Comput. Soc. Annu. Symp. VLSI (ISVLSI)*, 2012, pp. 392–397.
[53] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proc. 16th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2011, pp. 91–104.
[54] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell, "Consistent and durable data structures for non-volatile byte-addressable memory," in *Proc. 9th USENIX Conf. File Storage Technol.*, 2011, p. 5.
[55] M. S. Ware *et al.*, "Architecting for power management: The IBM POWER7 approach," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2010, pp. 1–11.
[56] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, "CACTI 5.1," HP Lab., Palo Alto, CA, USA, Rep. HPL-2008-20, 2008, pp. 1–75.

Hai Jin (Fellow, IEEE) received the Ph.D. degree in computer engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 1994.

He is a Cheung Kung Scholars Chair Professor of computer science and engineering with HUST. He worked with the University of Hong Kong, Hong Kong, from 1998 to 2000, and as a Visiting Scholar with the University of Southern California, Los Angeles, CA, USA, from 1999 to 2000. He has coauthored 22 books and published over 800 research papers. His research interests include computer architecture, cloud computing, big data processing, and network security.

Dr. Jin was awarded a German Academic Exchange Service Fellowship to visit the Technical University of Chemnitz, Chemnitz, Germany, in 1996 and the Excellent Youth Award from the National Science Foundation of China in 2001. He is the Chief Scientist of ChinaGrid and the Chief Scientists of National 973 Basic Research Program Project of Virtualization Technology of Computing System, and Cloud Security. He is a fellow of CCF and a member of ACM.

Di Chen received the B.E. degree from the Huazhong University of Science and Technology, Wuhan, China, 2013, where he is currently pursuing the Ph.D. degree.

His current research interest is hybrid memory systems.

Haikun Liu (Member, IEEE) received the Ph.D. degree from the Huazhong University of Science and Technology (HUST), Wuhan, China.

He is an Associate Professor with the School of Computer Science and Technology, HUST. His current research interests include in-memory computing, virtualization technologies, cloud computing, and distributed systems.

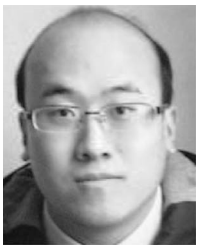
Dr. Liu was a recipient of outstanding doctoral dissertation award in Hubei province, China.

Xiaofei Liao (Member, IEEE) received the Ph.D. degree in computer science and engineering from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2005.

He is currently a Professor with the School of Computer Science and Engineering, HUST. His research interests are in the areas of system virtualization, system software, and cloud computing.

Rentong Guo received the Ph.D. degree from the Huazhong University of Science and Technology, Wuhan, China, in 2017.

His research interests are in the areas of caching systems and distributed systems.



Yu Zhang (Member, IEEE) received the Ph.D. degree in computer science from the Huazhong University of Science and Technology (HUST), Wuhan, China, in 2016.

He is currently an Associate Professor with the School of Computer Science, HUST. His research interests include computer architecture, system software, runtime optimization, programming model, and graph processing.

Dr. Zhang is a member of CCF, ACM, and USENIX.