MALRU: Miss-penalty Aware LRU-based Cache Replacement for Hybrid Memory Systems

Di Chen, Hai Jin, Xiaofei Liao, Haikun Liu*, Rentong Guo, Dong Liu
Services Computing Technology and System Lab/Big Data Technology and System Lab/Cluster and Grid Computing Lab
School of Computing Science and Technology, Huazhong University of Science and Technology, Wuhan, 430074, China
Email:{chendi, hjin, xfliao, hkliu, rtguo, liudong}@hust.edu.cn

***Abstract*—Current DRAM based memory systems face the scalability challenges in terms of storage density, power, and cost. Hybrid memory architecture composed of emerging *Non-Volatile 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 cache hit rate are not efficient either. In this paper, we take into account the asymmetry of 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 (MALRU) cache replacement policy for hybrid memory systems. MALRU is aware of the source (DRAM or NVM) of missing blocks and prevents high-latency NVM blocks as well as low-latency DRAM blocks with good temporal locality from being evicted. Experimental results show that MALRU improves system performance against LRU and the state-of-the-art HAP policy by up to 20.4% and 11.7% (11.1% and 5.7% on average), respectively.**

I. Introduction

In-memory computing is becoming increasingly popular for data-intensive applications in the big data era. However, current DRAM technology is hard to meet the continuously growing requirement of main memory due to low DRAM density and high power consumption [1], [2]. As a result, a large body of work pays more attentions on the emerging *Non-Volatile Memory* (NVM), such as *Phase Change Memory* (PCM) and *Spin Torque Transfer RAM* (STT-RAM). NVM technologies are promisingly considered as the alternatives of DRAM [3]–[6]. Nevertheless, NVMs have some disadvantages such as asymmetry of read/write latency and limited write endurance. They make NVM hard to be a direct substitute of DRAM. Therefore, hybrid memories comprising low-latency DRAM and large-capacity NVM become a practical way to build a large-scale main memory system.

Caches play an important role for bridging the large latency gap between CPU and main memory. Efficient use of cache can avoid abundant memory accesses. Existing cache replacement policies [7], [8] 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 plenties for DRAM and NVM are significantly different. The latency of evicting a NVM block is several times larger than that of evicting a DRAM block.

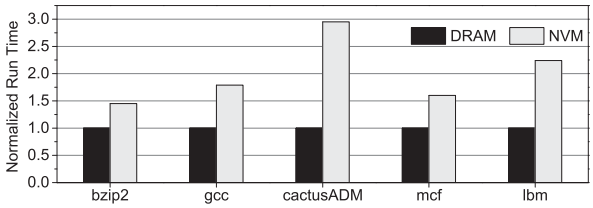

Fig. 1. The normalized execution time of SPEC 2006 benchmark in DRAM-based main memory and NVM-based main memory

Fig. 1 shows the normalized execution time of several applications from SPEC 2006 benchmark running in DRAM-based and NVM-based main memory. When the access latency of NVM is four 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 instruction* (MPKI) 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 plenties between DRAM and NVM should be considered. Also, current cache replacement policies that attempt 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 preferential choose a DRAM block as a victim upon cache replacement because evicting a 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.

In this paper, we take into account the asymmetry of LLC miss penalty on heterogeneous memories, and propose a *Miss penalty Aware LRU* (MALRU) cache replacement policy for hybrid memory systems. First, we propose 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 cachelines to distinguish the

978-3-9815370-8-6/17/$31.00 ©2017 IEEE

miss penalty of each DRAM or NVM blocks. Third, we partition the traditional LRU chain 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.

The contributions of our work 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 preferentially evicts low-latency DRAM blocks, and keeps high-latency NVM blocks and some low-latency 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.

(3) We evaluate MALRU with SPEC CPU 2006 benchmark, and compare MALRU with traditional *Least Recently Used* (LRU) and the state-of-the-art HAP policy in a hybrid memory system. Compared to *hybrid memory aware cache partitioning technique* (HAP) [9], our evaluation shows that MALRU improves application performance by up to 11.7% (5.7% on average), and achieves a mean speedup of 6.1% for single-thread applications and 4.8% for multi-programmed workloads.

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

II. Motivation

Cache hit rate and MPKI are no longer effective metrics for evaluating the cache performance in hybrid memory systems due to the asymmetrical cache miss penalty of heterogeneous memories. To better understand the reason that LRU performs inefficiently in hybrid memory systems, we illustrate the behavior of LRU algorithm in Fig. 2. Let N_i denote the address of a 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 4 entries, and the LLC hit overhead, DRAM block miss penalty, and NVM block miss penalty are 1, 10, 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 cache miss penalty.

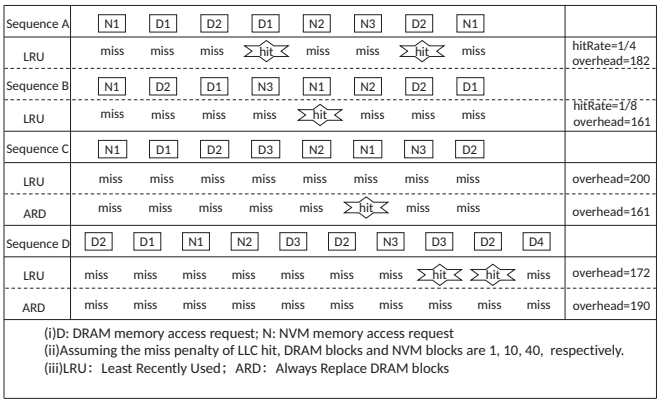

Fig. 2. LRU replacement policy performs inefficiently in hybrid memories

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, 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 MRU position, and the selected DRAM block is evicted. If there is no any DRAM block available, a NVM block at LRU position is evicted. We use sequence C to verify this method and find that ARD achieves better performance by improving the hit rate of NVM blocks, as shown in Figure 2. We have Principle 1 to improve cache performance in hybrid memory systems.

Principle 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, the ARD can cause cache thrashing. Sequence D in Fig. 2 demonstrates such a scenario. ARD increases the memory access overhead as it evicts the frequently accessed DRAM blocks. Some blocks with good temporal locality will be re-referenced in a near-immediate interval. We call them near-immediate re-referenced blocks. These blocks also should be kept in LLC to avoid cache thrashing. We have Principle 2 to improve the cache performance in hybrid memory systems.

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

Based on the two principles and cache re-reference interval prediction, we propose MALRU, an efficient cache replacement policy that keeps NVM blocks and near-immediate re-referenced DRAM blocks in LLC as more as possible.

III. Miss penalty-aware LRU Replacement

Fig. 3 shows the layout of LLC that is managed by miss-penalty aware cache replacement policy (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 leave NVM blocks more opportunities to retain in LLC. NVM blocks and some DRAM blocks with very short reuse distances are protected 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 victim section, the whole LLC degenerates to the traditional LRU stack. MALRU has to evict NVM blocks in the LRU position.

A. Detection of Cache Miss Penalty

To distinguish low-latency DRAM blocks and high-latency NVM blocks, MALRU adds a latency flag to denote different storage medium using one bit in each cache line. The flag denoting a high-latency NVM block is set to 1, otherwise the flag is set to 0. In the warm-up phase, MALRU records the latency of each LLC miss. With such statistics, MALRU uses OTUS algorithm [10], an image binarization algorithm, to determine a threshold which classifies all cache blocks into two categories. When a new entry is loaded into LLC, MALRU compares its memory access latency with the threshold, and then sets its latency flag correspondingly.

B. Re-reference Interval Distribution

Re-reference interval (or reuse distance) [11] of a cache block is the number of distinct memory accesses between two sequential accesses to the same data block. Re-reference interval is an important metric to evaluate data locality in LLC. A cache line is missed if its re-reference 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 re-reference interval of the cache line is set to $M+1$ uniformly in this paper. As a result, a cache block's re-reference interval ranges from 1 to $M+1$. If a cache block is hit in LLC, the position of the cache block in LRU chain is exactly the re-reference interval of this memory access.

Fig. 4 illustrates the process of re-reference interval statistics in MALRU. As sampling a small number of sets is sufficient to reflect the cache behaviors [12], MALRU only samples $\frac{1}{32}$ of the total sets to calculate the re-reference interval of DRAM blocks and NVM blocks, respectively. For each sampled cache set, MALRU uses 47 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 a NVM access sequence. We adopt the MALRU replacement policy to each sampled sets 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 re-reference interval distribution of the DRAM and NVM access sequences, as shown in Fig. 4.

C. Determine the Boundary of Reserved Section

As LLC hit rate is not an effective metric in hybrid memory environments, we propose AMAT to assess the LLC performance, as illustrated in Equation 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 $HitTime$ denotes the latency of LLC hits. We have the following equation.

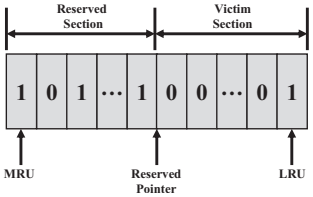

Fig. 3. Layout of MALRU-managed LLC set

$$\begin{aligned} AMAT = P_n * (HitTime + T_n * (1 - HitRate_n)) \\ + P_d * (HitTime + T_d * (1 - HitRate_d)) \end{aligned} \quad (1)$$

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

$$P_n + P_d = 1 \quad (2)$$

In the table of re-reference interval statistics, l_m denotes the total number of DRAM block accesses whose re-reference intervals are M in a given sampling epoch. Similarly, h_m corresponds to the total number of NVM block accesses with re-reference 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 j-th position of whole LRU stack is exactly the i-th DRAM block, and $\beta_{j,i}$ represents the probability of the j-th position of whole LRU stack is exactly the i-th 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 re-reference intervals from 1 to M, namely, the total hit rate of the individual DRAM block sequence is

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

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

Let $P_n(x = i)$ and $P_d(x = i)$ denote the probability of the i-th position in the whole LRU stack is a NVM block and a DRAM block, respectively. For example, the probability of re-reference interval i ($i \leq M$) of the DRAM block is the sum of probability on each position the i-th DRAM block can be found. $P_d(x = i)$ can be derived from the re-reference interval distribution tables. Finally, we can induce the hit rate of DRAM blocks in the hybrid memory access sequence:

$$HitRate_d = \sum_{i=1}^{M} (\sum_{j=i}^{M} \alpha_{j,i} * P_d(x = i)) \quad (4)$$

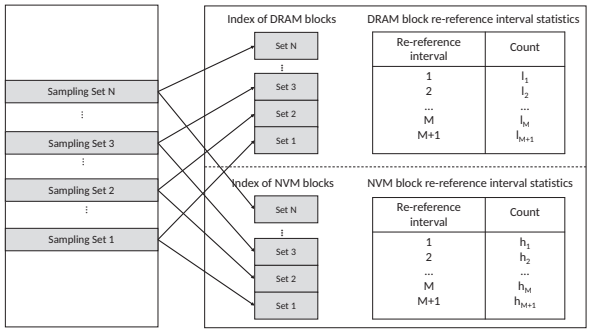

Fig. 4. The mechanism of obtaining the re-reference interval distribution of DRAM access sequence and NVM access sequence

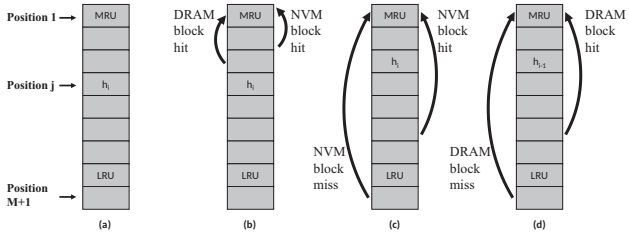

Fig. 5. The state transition diagram of $\alpha_{j,i}$

where $\alpha_{j,i}$ represents the probability of a cache block in j-th position of LRU stack is exactly the i-th DRAM block. Fig. 5(a) shows the state of $\alpha_{j,i}$, and Fig. 5(b), (c), and (d) show 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 re-referenced, as shown in Fig. 5(b). In Fig. 5(c), the current state is $\alpha_{j-1,i}$, a NVM block miss or a NVM block hit between position $j-1$ and $M+1$ will transfer the state to $\alpha_{j,i}$. In Fig. 5(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

$$\alpha_{j,i} * (P_d * \sum_{k=1}^{i-1} P_d(x=k) + P_n * \sum_{k=1}^{j-i} P_n(x=k)) \quad (5)$$

$$\alpha_{j-1,i} * P_n * \sum_{k=j-i}^{M+1} P_n(x=k) \quad (6)$$

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

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

$$\alpha_{j,i} = \begin{cases} \frac{\alpha_{j-1,i} * P_n * \sum_{k=j-i}^{M+1} P_n(x=k) + \alpha_{j-1,i-1} * P_d * \sum_{k=i}^{M+1} P_d(x=k)}{1 - P_d * \sum_{k=1}^{i-1} P_d(x=k) - P_n * \sum_{k=1}^{j-i} P_n(x=k)}, & j \leq n+1 \\ 0, & j > n+1 \end{cases} \quad (8)$$

TABLE I
System configurations and workloads

Processor	1-core/4-core CMP, 2GHz, out-of-order
L1 caches	32KB I-caches, 64KB D-caches 2-way associative, 64B cache line, 2-cycle latency
L2 caches	256KB, 4-way associative, 64B cache line, 10-cycle latency
L3(shared)	2MB, 16-way associative, 64B cache line 25-cycle latency, write-back
Main memory	DRAM:1GB (1 channel), NVM:3GB (3 channel), DRAM:150-cycle latency NVM:500-cycle read latency, 1000-cycle write latency
Workloads	single-thread: bzip2, gcc, cactusADM, mcf, milc, lbm, soplex mix1: (bzip2, gcc, milc, soplex) mix2: (mcf, lbm, cactusADM, hmmer) mix3: (bzip2, hmmer, lbm, sjeng) mix4: (libquantum, gcc, cactusADM, soplex)

where n is the begining position of the reserved section in MALRU. In the ideal case, no DRAM blocks exists 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 re-reference interval distribution tables.

We determine the position of reserved pointer by calculating AMAT from 1 to M, and the position in which AMAT achieves the minimum value is the reversed pointer in the next epoch. The major computation overhead is caused by sampling the LLC and determining the boundary of reserved section for each cache set periodically.

IV. Evaluation Methodology

A. System Configuration

We conduct our experiments on an integrated simulator, Gem5 [13] and NVMain [14]. Gem5 simulates the processor and cache, while NVMain simulates the hybrid main memory. Table I shows the system configurations and experimental workloads. We use 7 single-thread workloads and 4 multi-programmed workloads from the SPEC CPU 2006.

As the data distribution affects the system performance significantly, we use the address mapping scheme in NVMain to guarantee that memory accesses are evenly distributed in all memory channels.

B. Performance Evaluation

We compare MALRU with two other cache replacement polices LRU and HAP [9], which logically divides the LLC into NVM section and DRAM section and changes the size of NVM section dynamically. We refer the application performance using LRU algorithm as a baseline. Fig. 6 shows the normalized *Instructions Per Cycle* (IPC) for both single-thread and multi-programmed workloads. Compared to HAP and LRU, MALRU improves IPC by approximate 5.7% and 11.1%, respectively. For workloads *mcf*, *lbm*, and *mix2*, compared to LRU, MALRU improves application performance by 20.4%, 14.3%, and 15.7%, respectively. We classify the workloads that benefit from MALRU into two categories: cache thrashing and recency-friendly access patterns. For the thrashing access pattern, such as *mcf* and *lbm*, 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

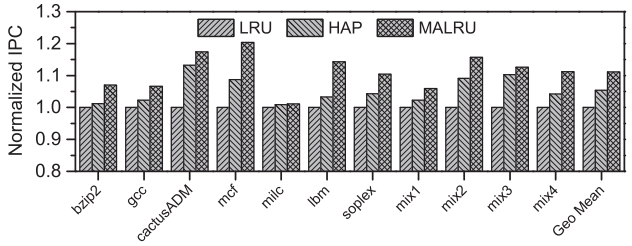

Fig. 6. The comparison of IPC among LRU, HAP, and MALRU

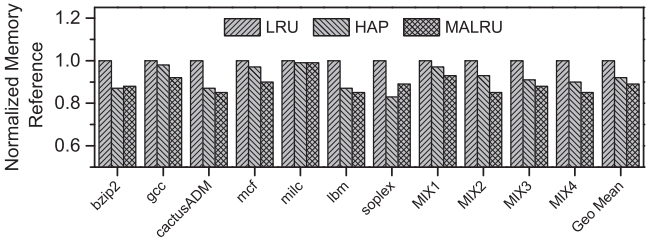

Fig. 7. Memory References Reduction using LRU, HAP, and MALRU

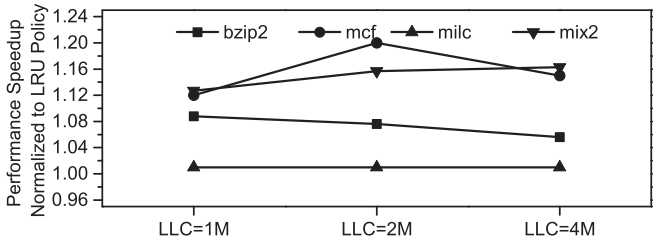

Fig. 8. MALRU sensitivity to cache size

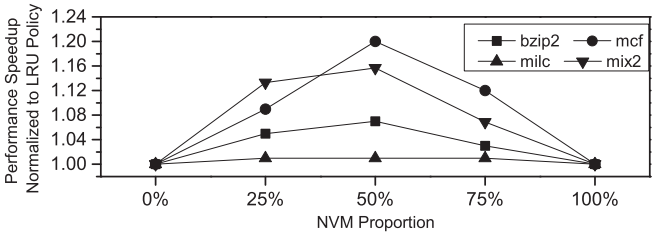

Fig. 9. MALRU sensitivity to the percentiles of NVM in main memory

reducing the total cache miss penalty. As NVM blocks are preferentially kept in LLC and may be re-referenced 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 most 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 Accesses

Fig. 7 shows the total number of memory accesses under three cache replacement policies. For recency-friendly workloads such as *bzip2* and *soplex*, HAP leads to fewer memory reference than MALRU. This is because NVM blocks in LRU is more likely to exceed the threshold in our configuration as the capacity of NVM is 3 times of DRAM. When a DRAM block is missed, MALRU would evict the first DRAM block from the victim section, while HAP has a more probability to evict a NVM block because the block in LRU position is evicted. For workloads with distant re-reference interval, MALRU leads to less memory accesses because both the DRAM and NVM blocks have distant re-reference interval, some NVM blocks are more likely hit because they have more chance to stay in LLC while DRAM blocks are replaced preferentially.

D. Sensitivity to Cache Size and NVM Proportion

Sensitivity to Cache Size. Fig. 8 presents the system performance improvement of MALRU varying with cache sizes. For workloads with short re-reference interval and good temporal locality, such as *bzip2*, the competition between DRAM blocks and NVM blocks is mitigated as the cache size increases. Hence, MALRU performs more similar as LRU when LLC capacity becomes larger.

For workloads with thrashing access pattern, when 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 [11]. 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.

Sensitivity to NVM Proportion. MALRU achieves better performance improvement when abundant NVM blocks compete with DRAM blocks in LLC. If only NVM data or DRAM data exists in LLC, the MALRU degenerates to LRU. Fig. 9 exhibits the MALRU performance varying with the proportion of NVM in main memory. Streaming access pattern such as *milc* shows 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. *Mcf* suffers performance degradation when the proportion of NVM increases since the MALRU selects NVM blocks with the distant re-reference interval as candidates for eviction, behaving more similar to LRU. When the NVM proportion becomes smaller, the NVM blocks with long re-reference interval is protected by preferential replacing DRAM blocks.

E. Hardware and Software Overhead

Table II shows the hardware overhead of MALRU for a 2 MB 16-way LLC with 64 byte cachelines. MALRU adds one bit of latency flag for each cacheline, and the total hardware overhead is 4KB. MALRU uses two tables to record the counts of different re-reference intervals for DRAM and NVM blocks, as shown in Fig.4. As LLC has 16 lines in each set, each distribution table of re-reference 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 47 bits to record the index of each sampled cacheline, and MALRU only samples $\frac{1}{32}$ of the total sets (64 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 hardware

TABLE II
Hardware overhead in MALRU

Source	Overhead	Total Overhead
Latency flag	1bit × # of cache line	4 KB
Re-reference interval counter	32bit × 17 × 2	136 B
Index of sampled blocks	2×16×47bit×# of set	11.75 KB
Reserved pointer	8bit × 1	1 B

overhead of MALRU is only 0.78% of the total capacity of LLC.

To determine the position of reserved pointer, we search the minimum AMAT in software periodically. The computation complexity of MALRU is $O(M^2)$. The performance overhead is negligible as M is only a constant (16).

V. Related Work

There have been a large body of work on cache replacement policies. Those studies can be classified into three categories: re-reference interval prediction, dead block prediction, and partition based replacement policies.

Re-reference interval based replacement policies [6], [11], [15], [16] predict a block that will be re-referenced furthest in the further and evict the block on a LLC miss. LRU replacement policy assumes that the most recently used block will be re-referenced soon and always replaces the cacheline in the LRU position. *Dynamic Insertion Policy* (DIP) [6] dynamically changes the insertion policy according to the cache access patterns. *Re-reference Interval Prediction* (RRIP) [11] further improves the DIP performance in mixed memory access patterns.

Some other work tries to reduce the miss rate of LLC by replacing the dead blocks in the cache. Dead blocks prediction [17], [18] based technologies improve cache efficiency by preferentially replacing dead blocks, because these blocks are unlikely re-referenced before they are evicted. Khan et al., sample the program counters to predict the dead blocks [17]. Ahn et al., observe that some blocks that have been written in LLC are not re-referenced again and become dead blocks [18]. They proposes a theoretical model to determine dead blocks in a STT-RAM based LLC and bypass LLC writes to these dead blocks.

Partition based replacement [9], [19], [20] policies classify the LLC blocks into several partitions. Those partitions correspond to different memory access behaviors, and are replaced separately. These policies propose sophisticated schemes to adjust the size of each partition dynamically. HAP [9] is the most similar work to our approach for hybrid memory systems. HAP logically divides the LLC into NVM and DRAM partitions and dynamically adjust the space of NVM partition. As HAP does not take reference intervals into account, it may have side effects in some cases.

VI. Conclusion

The conventional LRU-based cache replacement policies are no longer effective in hybrid memory systems. In this paper, we propose a new cache performance metric–AMAT and a miss penalty 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. Experimental results show that MALRU improves system performance over LRU and HAP by up to 20.4% and 11.7%, respectively, while incurring only 0.78% hardware overhead.

Acknowledgment

This work is partly supported by NSFC under grants No.61300040, 61672251, and National High-Tech Research and Development Program (863 Program) under grant No. 2015AA015303.

References

[1] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "Archshield: Architectural framework for assisting DRAM scaling by tolerating high error rates," in *Proc. of ISCA*, 2013, pp. 72–83.
[2] O. Mutlu, "Memory scaling: A systems architecture perspective," in *Proc. of IEEE International Memory Workshop*, 2013, pp. 21–25.
[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. of HPCA*, 2013, pp. 234–245.
[4] J. Ren, J. Zhao, S. Khan, Jongmoo, Y. Wu, and O. Mutlu, "ThyNVM: Enabling software-transparent crash consistency in persistent memory systems," in *Proc. of MICRO*, 2015, pp. 672–685.
[5] 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. of MICRO*, 2013, pp. 421–432.
[6] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. Steely, and J. Emer, "Adaptive insertion policies for high performance caching," in *Proc. of ISCA*, 2007, pp. 381–391.
[7] R. Azevedo, J. D. Davis, K. Strauss, P. Gopalan, M. Manasse, and S. Yekhanin, "Zombie memory: extending memory lifetime by reviving dead blocks," in *Proc. of ISCA*, 2013, pp. 452–463.
[8] M. Chaudhuri, "Pseudo-LIFO: The foundation of a new family of replacement policies for last-level caches," in *Proc. of MICRO*, 2009, pp. 401–412.
[9] W. Wei, D. Jiang, J. Xiong, and M. Chen, "HAP: Hybrid-memory-aware partition in shared last-level cache," in *Proc. of ICCD*, 2014, pp. 28–35.
[10] N. Otsu, "A threshold selection method from gray-level histograms," *IEEE Transactions on Systems, Man and Cybernnetics*, vol. 9, no. 1, pp. 62–66, 1983.
[11] A. Jaleel, K. B. Theobald, S. C. S. Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. of ISCA*, 2010, pp. 60–71.
[12] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proc. of MICRO*, 2006, pp. 423–432.
[13] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The GEM5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011.
[14] M. Poremba and Y. Xie, "NVMain: An architectural-level main memory simulator for emerging non-volatile memories," in *Proc. of IEEE Computer Society Annual Symposium on VLSI*, 2012, pp. 392–397.
[15] G. Kurian, S. Devadas, and O. Khan, "Locality-aware data replication in the last-level cache," in *Proc. of HPCA*, 2014, pp. 1–12.
[16] D. A. Jemenez, "Insertion and promotion for tree-based pseudolru last-level caches," in *Proc. of MICRO*, 2013, pp. 284–296.
[17] S. Khan, Y. Tian, and D. A. Jimenez, "Sampling dead block prediction for last-level caches," in *Proc. of MICRO*, 2010, pp. 175–186.
[18] J. Ahn, S. Yoo, and K. Choi, "DASCA: Dead write prediction assisted STT-RAM cache architecture," in *Proc. of HPCA*, 2014, pp. 25–36.
[19] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *Proc. of ISCA*, 2011, pp. 57–68.
[20] Z. Wang, S. Shan, T. Cao, J. Gu, Y. Xu, Y. Xie, and D. Jimenez, "WADE: Writeback-aware dynamic cache management for NVM-based main memory system," *ACM Transactions on Architecture and Code Optimization*, vol. 10, no. 51, 2013.