HAP: Hybrid-Memory-Aware Partition in Shared Last-Level Cache

WEI WEI, DEJUN JIANG, JIN XIONG, and MINGYU CHEN, State Key Laboratory of Computer Architecture Institute of Computing Technology, Chinese Academy of Sciences; University of Chinese Academy of Sciences

Data-center servers benefit from large-capacity memory systems to run multiple processes simultaneously. Hybrid DRAM-NVM memory is attractive for increasing memory capacity by exploiting the scalability of Non-Volatile Memory (NVM). However, current LLC policies are unaware of hybrid memory. Cache misses to NVM introduce high cost due to long NVM latency. Moreover, evicting dirty NVM data suffer from long write latency. We propose hybrid memory aware cache partitioning to dynamically adjust cache spaces and give NVM dirty data more chances to reside in LLC. Experimental results show Hybrid-memory-Aware Partition (HAP) improves performance by 46.7% and reduces energy consumption by 21.9% on average against LRU management. Moreover, HAP averagely improves performance by 9.3% and reduces energy consumption by 6.4% against a state-of-the-art cache mechanism.

CCS Concepts: • **Hardware** → **Non-volatile memory**; • **Theory of computation** → *Caching and paging algorithms*; • **Computer systems organization** → *Processors and memory architectures*;

Additional Key Words and Phrases: Hybrid memory, CPU cache

ACM Reference format:

Wei Wei, Dejun Jiang, Jin Xiong, and Mingyu Chen. 2017. HAP: Hybrid-Memory-Aware Partition in Shared Last-Level Cache. *ACM Trans. Archit. Code Optim.* 14, 3, Article 24 (September 2017), 25 pages.
<https://doi.org/10.1145/3106340>

1 INTRODUCTION

As the number of CPU cores in data-center servers increases, the number of concurrent running threads also increase. Data-center servers require large capacity memory to simultaneously serve multiple threads' working sets. However, well-known DRAM shortcomings (high refresh energy [19] and limited scalability [46]) make DRAM-only solutions unattractive for constructing large capacity memory system. Fortunately, emerging Non-Volatile Memory (NVM) technologies, such as PCM [25], STT-RAM [6], ReRAM [13], and 3D-xPoint [10] have better scalability and do not require refreshing. Thus, hybrid DRAM/NVM memories are proposed to extend memory capacity

Extension of conference article: W. Wei, D. Jiang, J. Xiong, M. Chen, HAP: Hybrid-memory-aware partition in shared last-level cache. In *Proceedings of the International Conference on Computer Design*, 2014.

This work is supported by the National Key Research and Development Program of China under grant No. 2016YFB1000302, National Science Foundation of China under grant Nos. 61502448, 61521092, and 61379042.

Authors' addresses: W. Wei, 68 Shunan North Road, China Minsheng Bank, Beijing China, 101300; email: weiwei01@ict.ac.cn; D. Jiang, J. Xiong, and M. Chen, 6 Kexueyuan North Road, Institute of Computing Technology Chinese Academy of Sciences, Beijing China, 100190; emails: {jiangdejun, xiongjin, cmy}@ict.ac.cn.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2017 ACM 1544-3566/2017/09-ART24 \$15.00

<https://doi.org/10.1145/3106340>

Fig. 1. Hybrid memory architectures.

for data-center servers. However, the relatively long read and write latencies raise challenges to the adoption of NVMs in hybrid memory.

Hybrid memory systems are usually divided into two types. One type is to use a small amount of DRAM as a cache to hide long latencies of NVMs (as shown in Figure 1(a)). In such systems [15, 27, 47], precious DRAM space cannot be used efficiently as it is not added to the main memory capacity. Moreover, for applications that have weak locality, performance benefiting from the DRAM cache decreases but cache management overheads increase.

The second type (as shown in Figure 1(b)) [18, 23, 31, 50] considers both DRAM and NVM as a part of main memory and manage them under a single physical address space. This type of hybrid memory fully utilizes the spaces of the two memory media and is more efficient. Thus, we work on this type in this article. Due to NVMs' longer latencies, most existing systems focus on migrating the rarely accessed data into NVM and the frequently accessed ones into DRAM, to provide high performance.

However, given DRAM and NVM sharing the Last-Level-Cache (LLC), cache management schemes also control the number of read and write operations on DRAM and NVM. Unfortunately, current cache mechanisms [11, 22, 26, 29, 34, 36, 39, 45] do not consider the asymmetric characteristics of DRAM and NVM, which may also cause significant performance decrease.

Designing hybrid memory aware LLC management is non-trivial, which mainly faces three challenges. (1) Current LLC performance metric, such as Misses Per-Kilo Instructions (MPKI), cannot exactly reflect cache performance when using hybrid memory. The reason is that it does not consider the asymmetric costs of DRAM and NVM data misses. Within a hybrid memory, two equal MPKIs do not indicate the same cache performance. For example, if DRAM data account for all cache misses for one MPKI, the cache performance is obviously better than the case when MPKI is same but NVM data account for all misses. (2) The competition for LLC space between DRAM and NVM data significantly affect the cache performance. For example, if the whole cache space is used to cache NVM data, then it will reduce NVM data misses but dramatically increase DRAM data misses. As a result, cache performance also decreases. This tells us that we should carefully partition DRAM and NVM spaces to reduce the total miss cost. (3) Writing back vast dirty NVM cache lines also hurts cache performance due to the long NVM write latency. Generally, the memory controller can buffer write-back operations from LLC in a write queue. However, due to the long NVM write latency, the write queue easily becomes fully occupied, which blocks the subsequent memory requests.

In this article, we carefully address the three challenges above. We first propose a new performance metric, TMPKI, to describe LLC miss cost with an underlying hybrid memory. By adding a latency weight for each NVM data miss, TMPKI can reflect the total miss cost per-kilo instructions. We then propose a HAP cache mechanism. HAP improves performance by (1) distinguishing

cachelines caching NVM data from those caching DRAM data, (2) logically dividing the cache into two partitions and restricting the NVM partition size in an appropriate range to guarantee the total cost to be a small value, (3) giving NVM dirty data a second chance to reside in LLC (i.e., 2-chances policy) to reduce write-back costs, and (4) dynamically adjusting the thresholds of NVM partition sizes every a period to maintain a high performance anytime.

Overall, the contributions of our work are as follows:

- We propose a new cache performance metric, TMPKI, which takes the asymmetric costs of DRAM and NVM data misses into account. The metric thus can guide cache management for the LLC on the top of hybrid memory systems.
- We propose HAP, a hybrid-memory aware cache partition mechanism. HAP includes two mechanisms: HAP-partition and 2-chances. HAP-partition dynamically adjusts the NVM and DRAM cache partition sizes according to access behaviors and guarantees TMPKI to be a small value all the time. HAP-2-chances further reduces NVM data write-back costs by giving NVM dirty data second chances to reside in LLC.
- We propose a thorough evaluation of HAP-partition and HAP-2-chances and analyze their impacts on IPC, hybrid memory references and write energy consumption for a variety of workloads, which includes traditional SEPC benchmarks and typical large memory applications (data mining, graph processing, image processing, and HPC).
- We further integrate our cache mechanisms with a state-of-the-art page migration mechanism, RaPP [31]. Experiment results show that HAP can be compatible well to page migration mechanisms. Exploiting the two technologies both achieves higher performance.

The organization of this article is as follows. Section 2 describes the background of hybrid memory and motivations of designing an efficient cache mechanism for the LLC in hybrid memory system. Section 3 describes the design and implementation of HAP. Sections 4–5 present our experimental methodology and results. Section 6 describes the related work about the cache mechanisms, and Section 7 concludes our article.

2 BACKGROUND AND MOTIVATIONS

2.1 Hybrid Memory Systems

The well-known shortcomings of DRAM (high refresh energy and limited scalability) make it difficult to construct large-capacity memories using DRAM-only memory systems. Compared to DRAM, emerging non-volatile memory technologies such as PCM, STT-RAM, ReRAM, and 3D-xPoint require no refresh and promise better scalability. However, NVM usually has 2–3 times longer read latency and much longer write latency than DRAM [52]. Thus, hybrid memories are proposed as an alternative for building large-capacity memories.

Figures 1(a) and (b) shows the two types of hybrid memory organizations. The inclusive one uses DRAM as a cache to hide high NVM access latencies, which reduces the utilization of DRAM capacity. The exclusive one maps DRAM and NVM into a single physical address space, which fully exploits the capacities of both DRAM and NVM. In this work, we target on the exclusive organization. As shown in Figure 1(b), the main memory consists of several DRAM and NVM DIMMs. These DIMMs share the memory bus and LLC. Memory operations issued by the CPU are first served by L1 and L2 caches. Misses in the L2 cache are sent to the LLC. The LLC caches data both from NVM and DRAM with a write-back policy.

As shown in Figure 2, two steps are generally executed when a miss happens in LLC. First, LLC controller selects a victim line (i.e., D in Figure 2(a)) to be evicted. If the line is dirty, then it should be written back to the hybrid memory. Otherwise, it is just invalidated. Second, LLC controller

Fig. 2. Operations after LLC missing data (i.e., G).

loads the missing data (i.e., G in Figure 2(b)) from the hybrid memory. Obviously, the second step always triggers a read operation to the hybrid memory for both read and write misses. Here, these read operations result in load costs. In the first step, the cost is introduced only when a dirty cache line is evicted from the LLC. In such case, writing dirty cache lines brings evict cost.

Next, we analyze both the load cost and the evict cost.

2.2 Load Cost Analysis

Since NVM's read latency is much higher than that of DRAM, the load cost from one NVM data miss is much higher than that from one DRAM data miss. Therefore, we can not use the total number of misses as a metric to evaluate the performance of cache mechanisms. Equation (1) shows the calculation of the total load cost. D_{mc} and N_{mc} represent the DRAM data misses and NVM data misses, respectively. Likewise, L_{dram} and L_{nvm} represent the access latencies of DRAM and NVM, respectively. L_{ratio} indicates that the read latency of NVM is L_{ratio} times longer than that of DRAM.

$$\begin{aligned} Total_{cost} &= D_{mc} \times L_{dram} + N_{mc} \times L_{nvm} \\ &= (D_{mc} + N_{mc} \times L_{ratio}) \times L_{dram}. \end{aligned} \quad (1)$$

From Equation (1), we can observe that total miss counts (sum of D_{mc} and N_{mc}) can not reflect total load cost. We thus propose a new metric, TMPKI, to represent the load cost. As shown in Equation (2), transformed misses (T_{mc}) adds a weight for each NVM data miss, whose value is equal to the latency ratio (i.e., L_{ratio}),

$$T_{mc} = D_{mc} + N_{mc} \times L_{ratio}. \quad (2)$$

Obviously, to achieve high performance, we should always maintain TMPKI to be small. A simple approach is caching more NVM data to reduce the numbers of NVM misses. However, we conduct experiments and observe that caching NVM data as much as we can does not achieve best performance. The experiments are described as follows.

We select a few applications (shown in Table 2) and run them in a full system simulator (described in Section 4). For each application, we run it with different proportions of NVM lines in the LLC. We then analyze the IPCs and NVM data miss counts for each configuration.

Here, we show the results of two applications (cactusADM and soplex) as examples. As shown in Figure 3(a),¹ after the proportion of NVM lines exceeds 25%, the NVM data miss count is nearly unchanged while the IPC dramatically decreases while the proportion increases. Similar observations can be obtained in Figure 3(b). When the proportion is smaller than 75%, the IPC increases along with the NVM data miss count decreases. However, when the proportion is larger than 75%,

¹Note that the value, 0, in the figures means that LLC only cache DRAM data. Inversely, 100% means that LLC only cache NVM data.

Fig. 3. The impact of NVM data misses on performance.

the NVM data miss count is nearly unchanged while the IPC decreases. We have similar observations for all applications. The reason is that after the proportion of NVM lines exceeds a certain value, LLC stores more stale NVM lines (i.e., not be referenced anymore) and DRAM data incurs more cache misses due to the smaller cache space for them.

Therefore, the results indicate that applications can obtain performance benefits when the proportion of NVM lines is maintained within an appropriate range. When the proportion is smaller than the low bound of the range, LLC performance decreases due to large number of cache misses from NVM data. When the proportion is larger than the upper bound of the range, performance also decreases due to increasing misses from DRAM data.

2.3 Evict Cost When LLC Evicts Dirty Data Underlying with a Hybrid Memory

Unlike load cost, the evict cost usually generates little impact on performance. There are two reasons. First, the evict cost only happens when the victim data are dirty. Second, the memory controller normally has a write buffer for pending requests. CPU has to stall only when the buffer is full. Otherwise, there is no delay even when writebacks happen. However, given the long NVM write latency, the write buffer in the hybrid memory controller easily becomes fully occupied if too much NVM dirty data are evicted. In this case, the evict cost becomes large. Recent work [49] shows that the evict cost contributes 21% of the total execution time on average, in the hybrid memory. We thus argue that LLC should avoid evicting dirty NVM lines to reduce the numbers of write operations to the NVM. However, if LLC only evicts clean NVM data, the total misses increase, because massive clean data could be re-accessed after being evicted. Moreover, when clean lines are all evicted, LLC has to evict dirty lines, which also introduces NVM write operations. Therefore, we should carefully decide when the dirty data should be evicted.

We run applications with their best proportions of NVM lines in LLC, which are obtained from the experiments in Section 2.2. In such doing, we minimize the impact of DRAM and NVM data competing for LLC space. Under such configurations, we further give NVM dirty lines different numbers of chances to reside in LLC. For example, 1-chance means that a dirty line must be evicted if it is in the LRU position when eviction occurs. Likely, 2-chances means that a dirty line can still reside in LLC if it should be evicted for the first time. It must be evicted if it is in the LRU position when eviction occurs again.

Experiment results show that 2-chances policy averagely reduces NVM writes 25.2% and improves IPC 13.2% compared with 1-chance policy. We also use the two applications (cactusADM and soplex) as examples. As shown in Figure 4(a) and 4(b), 2-chances policy reduces NVM writes by 22.2% and 52.8% compared with 1-chance policy, respectively. The reason is that many of dirty NVM lines are modified again after these lines are given a second chance. These operations thus hit in LLC and cause nothing to the hybrid memory. As a result, 2-chances policy increases

Fig. 4. The impact of evicting NVM dirty lines on performance.

performance by 7.2% (cactusADM) and 37.4% (soplex) compared with 1-chance policy, respectively. Moreover, 3-chances and 4-chances policies show similar NVM writes and performance to 2-chances policy. However, the more chances given to the dirty NVM lines, the more clean lines are evicted from LLC. This could instead introduces cache misses for the lines that are re-accessed again. Then, more cache misses again result in increasing data evictions. When there is no NVM clean line in LLC, it has to evict NVM dirty lines, which in turn introduces more NVM writes to the memory. As a result, the evict-clean-lines-only policy performs worst. Take cactusADM as an example, compared with 1-chance policy the number of cache misses increases by 68.3% with evict-clean-lines-only policy. Consequently, the number of NVM writes increases by 8.9% and IPC decreases by 38.3%.

The results indicate that giving each NVM dirty line a second chance to reside in LLC is sufficient to make a good tradeoff between the evict cost and the numbers of misses.

3 HYBRID MEMORY AWARE PARTITION

In this section, we present our HAP cache mechanisms. HAP includes two mechanisms, partitioning and 2-chances. HAP distinguishes cache lines from NVM and DRAM. It logically divides the cache space into two partitions for DRAM and NVM lines. The count of lines in a partition represents the partition size. For example, if a NVM line enters LLC, HAP considers the NVM partition size increases. Given the challenges described above, HAP restricts the NVM partition size (L_{NVM}) in an appropriate range ($T_{low} \sim T_{high}$). If L_{NVM} is less than T_{low} , then the performance dramatically decreases due to an significant increase in NVM data misses. Likewise, if L_{NVM} is greater than T_{high} , the performance also dramatically decrease due to an significant increase in DRAM data misses. To obtain high performance anytime, HAP dynamically adjusts the two thresholds (i.e., T_{low} and T_{high}) every a period. HAP also implements 2-chances policy so it reduces the evict cost as well as maintain a small miss count. We next describe the design and implementation of HAP mechanisms.

3.1 HAP Basic Partition Mechanism

Figure 5 shows the overview of the framework that supports HAP. Each line has a data type bit to identify whether the data are from DRAM or NVM. In our current implementation, when a new line enters into LLC, the cache controller checks its memory address to set the data type bit. A NVM Line Counter is added to count the current number of NVM lines (i.e., current NVM partition size) in LLC. The counter value increases by 1 when an NVM line enters into LLC. Correspondingly, it decreases by 1 when an NVM line is evicted.

HAP adjusts the NVM partition size through evicting different types of lines. When a miss occurs and triggers a replacement, there are two cases when HAP must choose an appropriate type of line as the evicted data:

Fig. 5. HAP framework overview.

Case 1: Current value of NVM Line Counter is greater than T_{high} . This means that LLC has more NVM data than it should have. In this case, HAP should not hold more NVM data. Thus, if the missed data are NVM data, HAP picks a LRU line from NVM lines as the victim.

Case 2: Current value of NVM Line Counter is smaller than T_{low} . This means that LLC has more DRAM data than it should have. In this case, HAP should not hold more DRAM data. Thus, if the missed data is DRAM data, HAP picks a LRU line from DRAM lines as the replacement victim.

Case 3: Except the two cases, HAP picks the line in the LRU position without distinguishing its type. In such doing, HAP can adjust the partition size meanwhile try its best to evict the least recently used line. For example, we assume that the NVM Line Counter exceeds T_{high} and a DRAM data miss occurs. In such case, HAP evicts the line in the LRU position even if the line is DRAM, rather than evicts a NVM line that is not in the LRU position.

3.2 2-chances Policy Implementation

Due to the long NVM write latency, the evict cost increases when vast NVM dirty lines are evicted from LLC. As described in Section 2.3, 2-chances policy can achieve good performance through giving NVM dirty lines second chances to reside in LLC.

We design a chance bit for each line. When a new line enters into LLC, its chance bit is set 1. The chance bit only works when the cache line has NVM dirty data. When HAP has to evict an NVM line from LLC (i.e., in *Case 1*), HAP first reads its dirty bit to judge whether the line is dirty. If the evicted NVM line is dirty, then HAP further judges whether its chance bit is 1. If its chance bit is 1, then HAP sets the bit as 0 and keeps the line in LLC to give it a second chance. Then it picks another NVM line that meets the requirement as a victim. When the LRU position is a dirty NVM line under *Case 3*, HAP also gives the line a second chance to reside in LLC by using the chance bit. For NVM data in LLC, those technologies guarantee that HAP only evicts clean ones or dirty ones whose chance bits indicate no chance (i.e., the value is 0). In such doing, HAP reduces much evict cost without increasing the number of misses. Algorithm 1 shows the final replacement mechanism of HAP.

3.3 Dynamically Adjusting the Thresholds

To maintain good performance anytime, HAP must dynamically adjust the two thresholds (T_{low} and T_{high}) based on CPU access behaviors in real time. To accomplish this, HAP uses a mechanism based on the Dynamic Set Dueling (DSS) technique [28]. DSS is widely used to select an appropriate one among some mechanisms in LLC [26, 28, 29, 34]. Here, HAP uses it to adjust the thresholds.

ALGORITHM 1: HAP Final Replacement Mechanism

```

Input:  $\alpha$ , A request for LLC
if ( $\alpha$  hits in cache line  $C$ ) then
    | promote( $C$ , MRU);
end
else
    if ( $L_{NVM} > T_{high}$  and  $isNVM(\alpha)$ ) then                                /* Case 1 */
        for line in  $NVM_{LRU}$  do
            if ( $isdirty(line)$  and  $ischance(line)$ ) then
                | set chance bit as 0;
                | continue;
            end
            else
                | evict line;
                | break;
            end
        end
    end
    else if ( $L_{NVM} < T_{low}$  and  $isDRAM(\alpha)$ ) then                                /* Case 2 */
        | evict the last line in  $DRAM_{LRU}$ 
    end
    else                                /* Case 3 */
        for line in  $LRU$  do
            if ( $isNVM(line)$  and  $isdirty(line)$  and  $ischance(line)$ ) then
                | set chance bit as 0;
                | continue;
            end
            else
                | evict line;
                | break;
            end
        end
    end
end

```

Figure 6 shows the mechanism used to dynamically adjust the thresholds. HAP dedicates a few sets as sample sets. The remaining are as normal sets. HAP divides these sample sets into n parts (e.g., $n = 3$). Different parts statically set different NVM partition sizes. For example, the maximal proportion of NVM lines in part I is $\alpha\%$. During system running, HAP sets sample points every a few instructions (e.g., 100 millions). On a sample point, HAP compares the performance among different parts and uses the NVM partition size ranges in which performance are best as the thresholds of normal sets in next period.

As described in Section 2, HAP uses TMPKI as the performance metric: If TMPKI is smaller, then the performance is better. To calculate TMPKI, HAP adds an Access Counter and a Cost Counter for each sample part. The Access Counter increments every when the sample part is accessed. The Cost Counter records the current value of T_{mc} . Thus, when a miss from DRAM data occurs, the counter increments one. When a miss from NVM data occurs, the counter increments L_{ratio} . The values of Access Counters and Cost Counters are reset to 0 at the end of a sample point.

Fig. 6. The mechanism used to adjust the thresholds.

Table 1. System Configurations [34, 54]

Processor	Eight cores, Out-of-Order, 2GHz (Gem5 and the Real Server have the same configuration)
L1 caches (per core)	32KB I-caches, 64KB D-caches, 2-way associative, 64B cache line, 1-cycle latency
L2 caches (per core)	512KB, 8-way associative, 64B cache line, 6-cycle latency, write-back
LLC (shared)	32MB, 16-way associative, 64B cache line, 30-cycle latency, write-back
Hybrid memory configuration	small size: 512MB DRAM, 1536MB NVM, large size: 2GB DRAM, 6GB NVM, each DIMM: 64 entry write queue; 64 entry read queue each DIMM: 8 banks, 8KB row buffer read prioritize write scheduling policy and scheme4 address mapping row buffer policy: closed
Memory latencies	DRAM: 200-cycle latency NVM: 400-cycle read latency, 1000-cycle write latency
Memory energies(nJ) per 64B	DRAM: 12.06 burst energy, 12.17 read energy, 14.48 write energy NVM: 3.77 burst energy, 10.68 read energy, 20.75 write energy

DSS [28] has shown that sampling a small number of sets (normally 2%–3% of total sets) can indicate the cache behavior with a high probability. Therefore, we select 40 sample sets every 1,024 sets. Considering the tradeoff between performance benefits and storage overhead, we divide the sample sets into five parts in our current implementation. We uniformly set the proportions of NVM lines from 0 to 100%² (i.e., 0, 25%, 50%, 75%, 100%) in the five parts. By doing so, HAP can perform well for a large amount of workloads although their access patterns can be significantly different.

4 EVALUATION METHODOLOGY

4.1 System Configurations and Workloads

We use both micro- and macro-benchmarks to evaluate our work.

Micro experiments. To show the effectiveness and efficiency of our mechanisms, we first use the multiprocessor full-system simulator Gem5 [2] to simulate a hardware architecture that includes processor, cache levels, and an exclusive hybrid memory. Table 1 shows its configurations. We modify the main memory module based on DRAMSim2 [32] to model a hybrid memory. The

²Note that the value, 0, means that the sample part only caches DRAM data. Inversely, 100% means that it only caches NVM data.

Table 2. Micro Workloads' Characteristics

Workloads	Description	MKPI	DRAM Accesses (% of total access)
bzip2	8 copies of bzip2	13.16	65.3
soplex	8 copies of soplex	18.92	10.5
cactusADM	8 copies of cactusADM	6.10	33.3
lbm	8 copies of lbm	44.66	69.8
mcf	8 copies of mcf	58.45	8.9
mix1	(bzip2, cactusADM, lbm, libquantum) $\times 2$	50.9	15.5
mix2	(cactusADM, libquantum, leslie3d, soplex) $\times 2$	43.65	13.5
mix3	(lbm, linkedlist, mcf, gobmk) $\times 2$	104.6	20.8
mix4	(linkedlist, leslie3d, omnetpp, sjeng) $\times 2$	58.75	28.7
mix5	(bzip2, hmmer, libm, sjeng) $\times 2$	51.9	18.2
mix6	(mcf, gcc, libquantum, omnetpp) $\times 2$	55.48	14.6

hybrid memory simulation models two DIMMs, DRAM, and NVM. In each DIMM, there are a write buffer and a read buffer, respectively. Each buffer can buffer 64 requests at most. Especially, if the write requests are buffered, they do not introduce any delay (i.e., write draining policy). We exploits a read prioritizes write [40] scheduling policy, because read requests are more important to the performance. The policy is that writes in the write buffer are to be scheduled for service whenever either of the following two conditions is satisfied: (1) the corresponding rank is idle and the number of writes in write buffer reaches a threshold (80% of the write buffer length) or (2) the write buffer is full. Due to the much long simulation time, we just run some fragments of workloads' instructions. For each workload, we first run 10 million instructions to warm up the caches and then run 500 million instructions for experiments.

We choose 13 applications from SPEC CPU2006 benchmark [5]. Since multiple workloads usually run on the same physical server simultaneously, we also use six mixed workloads to evaluate our work. Table 2 shows the characteristics of all workloads. We choose these workloads because their L2 MPKIs are relatively higher compared to the rest workloads. It means that they need an efficient LLC mechanism to improve performance. To match the memory footprints of these workloads, we set the total memory size to be 2 GB. Specifically, the first 512MB capacity is treated as DRAM module and the rest is treated as NVM module. We also show the percentage of each application's DRAM accesses in Table 2. The rest of accesses from CPU are to NVM.

Macro experiments. Since hybrid memory systems are generally used to provide large capacity memories, we further pick up 10 data-driven workloads from four typical categories (data mining, graph processing, image processing, and HPC). We show the workload characteristics in Table 3. Unlike using the fragments of workloads' instructions in micro experiments, we run these workloads on a real machine and collect their access traces during their whole lifetimes using a binary instrumentation tool Intel Pin [20]. Then we replay these traces in our cache simulator and hybrid memory simulator as described above. To match the memory footprint of these workloads, we set the total memory size to be 8GB. Specifically, the first 2GB capacity is treated as DRAM module and the rest 6GB capacity is treated as NVM module. Table 3 also shows the percentages of DRAM accesses of each application.

4.2 Compared Mechanisms

We evaluate four LLC mechanisms described as follows.

LRU. An implementation of the traditional cache replacement mechanism. Cache lines in a same set compose an LRU list. When a miss occurs, it evicts the cache line in the LRU position and loads the missed data to the MRU position. When a hit occurs, it also promotes the hit data to the

Table 3. Data-driven Workloads' Characteristics

Workloads		Description	MKPI	DRAM Accesses (% of total access)
SPLASH-2x [44]	radiosity	high performance computing	50.7	39.6
	fmm	high performance computing	70.2	7.2
Imagemagick [37] (im)	resize	image processing	100.6	22.2
	noise	image processing	110.6	5.4
NU-MineBench [24]	kmeans	data mining	113.3	49.5
	apriori	data mining	162.2	10.8
HPCC [21]	fft	high performance computing	67.2	62.5
	stream	high performance computing	52.8	37.4
libsvm [4]		data mining	108.6	5.9
graph500 [1]		graph processing	127.3	34.1

MRU position. The mechanism does not distinguish NVM data from DRAM data. We set it as the baseline.

WBAR. An implementation of hybrid-memory-aware cache mechanism. It is proposed in the article [49]. We implement WBAR according to the article. It divides LLC data into four types: Clean DRAM data (CD), Dirty DRAM data (DD), Clean NVM data (CN), and Dirty NVM data (DN). It then promotes the four types of data in different positions of the LRU list. In such doing, the four types of data can be evicted with different priorities (i.e., $CD > CN > DD > DN$). As a result, it keeps a dirty NVM data in LLC for longer time than other types of data. Compared with our mechanisms, WBAR gives NVM dirty data the highest priority to reside in LLC at any time. It cannot dynamically adjust priorities according to applications' behaviors.

HAP. Our cache mechanisms include two implementations, HAP-partition that dynamically partitions NVM line sizes to maintain small TMPKI, and HAP that includes both partition and 2-chances mechanisms. We adjust the two thresholds (T_{low} , T_{high}) every 100 million instructions and use the new ones for next 100 million instructions.

5 RESULTS

5.1 HAP Effectiveness

We first use micro experiments to evaluate the effectiveness of the mechanisms proposed in this article.

TMPKI Effectiveness. We show that TMPKI can effectively reflect cache performance. We take cactusADM as an example. We first run the workload with different proportions of NVM lines in LLC and measure the values of IPCs and TMPKIs. We normalize these values to the ones measured under the mechanism in which none of LLC lines is used to cache NVM data. As shown in Figure 7, IPC is larger if TMPKI is smaller in all mechanisms. We have similar observations for all workloads. Thus, we can conclude that TMPKI is an effective performance metric for LLC in hybrid memory systems.

Effectiveness of adjusting the thresholds. HAP adjusts the thresholds (T_{low} , T_{high}) during the system running. Figure 8 illustrates the impact of adjusting the thresholds. For each workload, we show the proportions of NVM lines at the end of workload running with LRU, WBAR, and HAP, respectively. For all workloads, the proportions of NVM lines are similar under WBAR. The reason is that WBAR gives similar priorities to NVM data and cannot change the proportions of NVM lines according to applications' behaviors. Inversely, the proportions are very different for most of workloads under LRU and HAP. For example, mix3 has 99.5% LLC lines caching NVM data under LRU, while only 12.2% LLC lines cache NVM data under HAP-partition. This indicates that

Fig. 7. TMPKI effectiveness in running cactusADM.

Fig. 8. Effectiveness of dynamically adjusting the thresholds.

HAP uses more LLC lines to cache DRAM data, otherwise many cache misses from DRAM data will result in performance degradation. mix5 has 2.0% NVM lines under LRU but 99.3% NVM lines under HAP-partition. In this case, HAP uses more LLC lines to cache NVM data, since NVM data misses have high cost.

We need a dynamic mechanism to adjust the thresholds, because memory access behaviors could change during the system running. For each bar in Figure 8, error bars present the minimum and maximum proportions of NVM lines during the workload running. We sample the proportions every 100 million instructions. Compared with WBAR, the results show that, for most workloads under HAP-partition, the proportions of NVM lines change in larger ranges during the whole lifetime. Take cactusADM as an example, the proportions of NVM lines change between 57% and 60% under WBAR. However, the changing range under HAP-partition is much larger, from 23% to 64%. The results indicate that HAP is able to dynamically adjust the NVM partition size according to applications' behaviors.

5.2 HAP Efficiency

As described in Section 2.1, each LLC miss introduces a read reference to the hybrid memory while only evicting dirty data introduces write references to the hybrid memory. We thus separately measure read and write references to NVM under the four cache mechanisms. NVM read references is equal to the count of total NVM data misses, which reflects the load cost from NVM data misses. NVM write references reflect the evict cost caused by evicting NVM dirty data. We then separately measure IPCs and total energy consumptions to show the performance improvements and energy efficiencies under those mechanisms.

5.2.1 Memory References.

NVM read references. Figures 9(a) and (b) separately show NVM read references under the four cache mechanisms in micro and macro experiments. LRU achieves most NVM read references

Fig. 9. NVM read references.

for each workload because it is unaware of the memory types of LLC data and thus introduces most NVM misses. Compared with LRU, HAP-partition averagely reduces 13.3% and 10.9% NVM read references in micro and macro experiments as it is aware of the underlying hybrid memory. However, WBAR averagely increases NVM read references by 2.3% in micro experiments and 3.8% in macro experiments, compared with HAP-partition. This indicates that WBAR introduces more NVM misses than HAP-partition does. The reason is that WBAR gives highest priority to NVM dirty lines, which introduces more misses to NVM clean data. Inversely, HAP only gives second chance to NVM dirty lines to reside in LLC, which does not introduce significant NVM misses like WBAR. Therefore, results show that HAP achieves similar NVM read references compared with HAP-partition.

NVM write references. Figure 10(a) shows NVM write references in micro experiments. Compared with LRU, HAP-partition, WBAR, HAP separately reduce NVM write references by 9.8%, 18.24%, and 21.5% on average. HAP-partition performs better than LRU because it reduces more misses. It thus needs less times to evict data. Compared with HAP-partition, WBAR gives NVM dirty lines highest priority to reside in LLC. It thus reduces the count of evicting NVM dirty data. As a result, WBAR reduces more NVM write references. Except partitioning, HAP also gives NVM dirty lines second chance to reside in LLC. In such doing, more writes are hit in LLC and it reduces NVM write references. Moreover, Figure 10(a) shows the reductions of NVM write references in HAP is 1.2x relative to that in WBAR. The reason is that HAP avoids introducing more NVM misses and thus reduces the count of evicting NVM data.

Similar results are observed in macro experiments as shown in Figure 10(b). On average, HAP-partition, WBAR, and HAP separately reduce NVM write references by 9.6%, 20.5%, and 22.5% compared with LRU. These results indicate that the three mechanisms also work well for data-driven workloads during their whole lifetimes.

5.2.2 IPC. We finally show performance improvements under the four cache mechanisms. In micro experiments, Figure 11(a) shows that they separately improve performance by 19.6%

(a) Micro experiments

(b) Macro experiments

Fig. 10. NVM write references.

(a) Micro experiments

(b) Macro experiments

Fig. 11. Performance improvements.

(HAP-partition), 27.7% (WBAR), and 33.6% (HAP) on average, compared with LRU. The results indicate that optimizing cache mechanisms according to the characteristics of hybrid memory systems is necessary for performance. HAP-partition achieves better performance than LRU, because it always maintains the total load cost to be small. However, it does not consider the evict cost due to evicting NVM dirty lines. WBAR reduces much NVM write references (the evict cost), it thus

Fig. 12. Total energy consumption of the hybrid memory system.

averagely achieves better performance by 6.69% compared with HAP-partition. However, WBAR also increases more NVM misses (shown in the above subsection), which counteracts part of benefit from reducing NVM writes. As a result, WBAR performs similar performance to HAP-partition for workloads bzip2, soplex, lbm, mcf, cactusADM et al. Take lbm as an example, compared with HAP-partition, WBAR only improves 1.4% IPC although it reduces 6.6% NVM write references. The reason is that WBAR increases 2.7% NVM read references compared with HAP-partition. Compared with WBAR, HAP further improves performance by 9.3%. There are two reasons. First, due to the 2-chances mechanism, HAP reduces the evict cost (i.e., NVM write references) without dramatically increasing the total load cost (i.e., TMPKI). Second, it dynamically adjusts NVM partition sizes to guarantee the total miss cost to be small all the time. Inversely, WBAR gives fixed priorities to LLC data all the time, which cannot guarantee that the total miss cost is always small.

Similar results are observed in macro experiments as shown in Figure 11(b). Compared with LRU, the three cache mechanisms separately improve performance by 20.1% (HAP-partition), 33.6% (WBAR), and 46.7% (HAP), on average. They all achieve more benefits under these data-driven workloads compared with SEPC workloads in micro experiments. The results indicate that optimizing cache mechanisms according to the characteristics of hybrid memory systems is important for data-driven workloads that are memory intensive.

5.2.3 Energy Consumptions. We also evaluate energy consumptions of the four mechanisms. As shown in Figure 12, HAP-partition, WBAR, and HAP averagely reduce energy consumptions by 8.9%, 16.1%, and 21.2% in micro experiments and 9.0%, 16.6%, and 21.9% in macro experiments, respectively. Since NVM write energy is large, the mechanisms that reduce more NVM write references consume lower energy. Therefore, WBAR and HAP achieve less energy consumption than HAP-partition. WBAR separately reduces 7.8% and 8.3% in micro and macro experiments, on average. Furthermore, HAP reduces 13.5% in micro experiments and 14.2% in macro experiments.

Fig. 13. Performance with different write buffer sizes.

5.3 Storage Overhead

We calculate the storage overhead of HAP here. There are 524,288 cache lines in the LLC with the capacity of 32MB. Thus, the total overhead of Data Type bit and chance bit is 128KB. We select 40 sample sets every 1,024 sets and thus there are total 1,280 sample sets. We divide them into 5 parts and execute sampling every 100 million instructions. Thus, we set each Access Counter and each Cost Counter as 32 bits, because the maximum cannot exceed 200 million. Each sample part also needs an NVM Line Counter to record its own NVM lines. Since the number of NVM lines cannot exceed 4,096,³ we set the NVM Line Counter as 12 bits for each sample part. Likewise, we set the NVM Line Counter as 19 bits⁴ for normal sets. Thus, the total storage overhead is only 0.2% of the total LLC capacity.

5.4 Sensitivity of HAP

Sizes of write buffer in the memory controller. As described in Section 2.1, write requests buffered in the write buffer have no impact on the total performance. Thus, we wonder how 2-chances mechanism performs according to the sizes of write buffer. We change the size of write buffer in NVM DIMM from 32 entries to 256 entries. Then we separately measure the IPCs underlying with only HAP dynamic partition (i.e., HAP-partition) and HAP (including both dynamic partition and 2-chances), when running data-driven workloads. Figure 13 shows the results. For each application, we regard the performance under HAP-partition with 64 entries buffer as baseline.

As shown in the figure, larger write buffer performs better performance. Compared with HAP-partition-64, HAP-partition-128, and HAP-partition-256 averagely improve IPC by 8.8% and 29.9%, respectively. Inversely, HAP-partition-32 averagely reduces 13.3% performance, compared with HAP-partition-64. The reason is that more write requests can be buffered in the larger write buffer and thus it introduces less stall time. Similar to HAP-partition mechanisms, HAP mechanisms also performs better when the size of write buffer is larger. For example, HAP-128 and HAP-256 averagely improve IPC by 7.6% and 9.4%, compared with HAP-64.

Compared with only dynamic partitions, 2-chances further reduces the total NVM write references by giving NVM dirty lines second chances to reside in LLC. As a result, HAP significantly reduces the chances that the write buffer becomes full. Therefore, HAP performs better than HAP-partition underlying with a same size write buffer. For example, HAP-128 averagely improves IPC by 24.2% compared with HAP-partition-128.

We also observe that the performance gap between HAP-128 and HAP-256 is small. Among all workloads, HAP-256 at most improves IPC by 3.8% (in radiosity) and only improves IPC by 0.6% (in fft) at least, compared with HAP-128. The reason is that most of write requests can be buffered

³There are total 4,096 cache lines in each sample part.

⁴There are total 524,288 cache lines.

Fig. 14. Performance with different LLC sizes.

with 128 entry write buffer using HAP mechanisms in these workloads. In such cases, increasing the size of write buffer to 256 entries is unnecessary. However, we observe that changing the size of write buffer from 128 to 256 entries is still important when only using HAP-partition. For example, HAP-partition-256 improves IPC by 26.8% in *fft*, compared with HAP-partition-128. The reason is that HAP without 2-chances introduces more NVM write references. As a result, the write buffer with 128 entries is not enough to buffer most of write requests.

LLC sizes. When the LLC size becomes larger, the count of total LLC miss decreases and the performance increases. We thus wonder whether the cache mechanisms being aware of hybrid memories is necessary when the LLC size becomes larger. We then change the LLC size from 8MB to 64MB. We run data-driven workloads with LRU and HAP again. Figure 14 shows the results. For each application, we regard the performance under LRU-32M as baseline.

As shown in the figure, HAP all performs better than LRU when using a same size LLC. For example, HAP-64MB averagely improves IPC by 47.4% compared with LRU-64MB. We further observe that most of workloads (e.g., *radiosity*, *resize*, *noise*, *libsvm* et al.) are insensitive when the LLC size increases from 32MB to 64MB. Take *noise* as an example. HAP-64MB only increases performance by 2.6% compared with HAP-32MB. However, HAP-64MB achieves 33.9% better performance than LRU-64MB in *noise*. These results indicate that designing cache mechanisms according to the characteristics of hybrid memories is more important than increasing size for the LLCs on the top of hybrid memories.

5.5 Compatibility to Page Migrations

As described in Section 1, many current works [18, 23, 31, 50] migrate pages that are frequently accessed to DRAM and those that are rarely accessed to NVM in the hybrid memory level. In such doing, they directly reduce the NVM references from CPU. In this section, we show how much performance gain can be achieved if we integrate our cache mechanisms with migration mechanisms.

We implement a state-of-the-art page migration mechanism in our hybrid memory simulator according to description in the article [31]. The mechanism uses a multi-LRU-queue to rank memory pages according to their references. It then migrates pages that are frequently accessed to DRAM and those that are rarely accessed to NVM. We then separately run data-driven workloads using LRU and HAP (including both dynamic partition and 2-chances) cache mechanisms underlying with the page migration mechanism. Since a same page address in LLC level can store different types of data after migrations, HAP now sets the data type bit of each cache line according to the information from the memory controller instead of memory addresses.

As shown in Figure 15(a), LRU with migrations averagely reduces the total NVM references by 14.5% compared with LRU without migrations. Similarly, HAP with migrations averagely reduces the total NVM references by 14.9% compared with HAP without migrations. The reason is that migrations directly reduces NVM accesses from CPU. Moreover, HAP with migrations still reduces

Fig. 15. Integrating HAP with page migrations.

NVM references by 16.9% compared with LRU with migrations. The reason is that HAP caches appropriate data through dynamic partition and 2-chances. It thus further reduces NVM accesses from LLC. As a result, HAP with migrations totally reduces NVM references by 28.9%, compared with LRU without migrations.

As shown in Figure 15(b), HAP with migrations performs best among the four configurations. It totally improves performance by 66.4% compared with LRU without migrations. Inversely, LRU with migrations only improves performance by 48.2% compared with LRU without migrations. The results indicate that HAP is compatible well to page migration mechanisms. One can achieve better performance if exploiting the two techniques together.

6 RELATED WORK

6.1 Hybrid Memory Systems

Hybrid memory systems consist of DRAM and NVM, which can exploit the advantages of both types. There are mainly two types of hybrid memory systems, *inclusive* organization and *exclusive* organization.

Inclusive hybrid memory systems. Inclusive organizations use DRAM as a cache for NVM. In such organizations, both DRAM and NVM are organized at page granularity. The inclusive organization results in low utilization of DARM as it treats DRAM as cache. Our work thus targets on the exclusive organization, which uses both DRAM and NVM as main memory.

There are research efforts to study cache replacement for inclusive hybrid memory, which focus on selecting which type of NVM pages into DRAM cache in the hybrid memory level. The details are as following.

Zhou et al. [54] design a PCM-based main memory architecture using DRAM as a write cache. However, no details are provided on cache replacement policies. Qureshi et al. [30] use DRAM as a hardware cache of PCM. To reduce the overhead of writing PCM, they propose lazy-write organization. Pages fetched from the hard disk are written only to the DRAM cache. Only pages

evicted from DRAM are written to PCM. Ferreira et al. [7] further propose a Clean-preferred page replacement. This policy evicts the second oldest victim if the oldest one is dirty while the second oldest is not (and does LRU otherwise). In such doing, they reduce the numbers of writeback to PCM. This mechanism is close to 2-chances mechanism proposed in our work. However, our experiment results show that evicting clean lines as much as possible hurts performance, because it introduces more misses. Thus, 2-chances mechanism only gives each NVM dirty line a second chance to stay in LLC. Lee et al. [15] propose a Threshold-based Pre-Invalidation (TBPI) cache technique to increase memory bus utilization and mask PCM's poor write performance. To hide the steep write-miss penalty in the DRAM cache, TBPI invalidates at least one data block from a particular set in the DRAM cache before the set is full. Yoon et al. [23] observe that accesses that are row buffer hits incur similar latencies (and energy consumption) in DRAM and PCM, whereas accesses that are row buffer misses incur longer latencies (and higher energy consumption) in PCM. They thus selectively cache PCM data with low row buffer locality and high reuse into DRAM to avoid accessing PCM data that frequently cause row buffer misses. Ham et al. [8] extend DRAM cache with dynamic granularity prefetch to improve bandwidth utilization. If a NVM page experiences successive accesses to different cache blocks, then the controller increases block prefetch for this page. On the other hand, if a page experiences few block misses, it decreases prefetch. These works all targets on how to cache one type of data (i.e., NVM) in DRAM cache. Compared with these works, our work addresses how to maintain the proportions of DRAM and NVM data in an appropriate range in the LLC level.

Exclusive hybrid memory systems. Exclusive organizations use both DRAM and NVM as main memory, which fully utilizes the capacities of the two memory types. In such organization, data are either in DRAM or in NVM. Thus, research efforts are made to design efficient data placement.

Most proposals dynamically migrate data into appropriate memory types. These works can be divided into two categories. The first one implement migrations in the OS-level. Zhang et al. [50] use Multi-Queues [55] to rank pages by their hotness, leveraging the OS to dynamically migrate hot pages into DRAM and cold pages into NVM. Salkhordeh et al. [33] also present a data migration scheme at the OS level. They use two Least Recently Used (LRU) queues (one for DRAM and one for NVM) and optimize the LRU queue for NVM to prevent non-beneficial migrations to DRAM. Since most memory accesses bypass OS and are executed through hardware elements (i.e., MMU), some other works implement migrations in the memory controller level to take the accurate access information into account. Ramos et al. [31] modify the memory controller to support Multi-Queues to identify hot PCM pages to migrate to DRAM. Yoon et al. [48] place data according to row buffer locality. They track row buffer miss counts in the memory controller. Data causing frequent row buffer misses are placed in DRAM. Other data are placed in PCM to save energy.

Since migrations cause much overhead [3], some works study on initially placing data into appropriate memory types. Hassan et al. [9] first profile applications to achieve the access patterns of objects. They also provide allocation interfaces for programmers. Programmers can thus allocate spaces from an appropriate memory type for each object according to the profiling results. Wei et al. [42] also profile applications and exploit global access characteristics of heap objects to guide initial data placement. They use an initial placement file to avoid modifying applications. Furthermore, they also monitor the pages which can be initially placed in wrong memory types and migrating them when necessary.

Our work is orthogonal to the above-mentioned works as we study on how to cache the two types of data (i.e., NVM and DRAM data) in the LLC level. Moreover, experimental results (in Section 4) show that our work is compatible with data placement mechanisms and achieves better performance when applied together.

6.2 LLC Mechanisms Towards DRAM-based Memory

Traditional LLC mechanisms target on the DRAM-based memories. There has been a significant amount of work on last-level cache mechanisms. These mechanisms can be broadly divided into two categories. The first category caches data on a total demand basis, mainly studying on how to minimize the total miss cost. The second category takes each application's performance into account. They partition cache sizes among multiple concurrently running applications and reduce cache misses of each application.

Cache mechanisms for improving the whole performance. These works focus on which data should be cached to improve the whole performance. Least-Recently-Used (LRU) is a basic and widely used mechanism to identify which data should be cached. Later, Qureshi et al. [26] propose a Dynamic Insertion Policy (DIP) to insert the incoming line in the LRU/MRU depending on which policy incurs less misses. By doing so, they protect the cache from thrashing behavior for certain applications. Seshadri et al. [36] propose an Evicted-Address Filter (EAF) cache, which mitigates the impact of both cache pollution and thrashing. They use EAF to predict the reuse behavior of cache lines by tracking their recency or reuses after eviction. They then modify the LRU replacement and insertion mechanisms based on the prediction. Xie et al. [45] propose a new cache mechanism (PIPP) that combines both dynamic insertion and promotion policies to utilize the benefits of cache partitioning and adaptive insertion. In such doing, PIPP can handle multiple types of memory behaviors and adapt to large amount of applications. Other works use re-reference interval prediction instead of LRU to statically/dynamically select the victim for replacement [11].

Our work is different from these works as our work must first decide which type of data should be evicted. As our experiment results show, although the cost of missing NVM data is larger than that of missing DRAM data, caching more NVM data does not always achieve higher performance. Our work thus maintains the proportions of DRAM and NVM data in an appropriate range by dynamically selecting DRAM or NVM data to be evicted. Once our work decides the type of data to be evicted, it then uses LRU to select a specific data item (belonging to the previously decided memory type) for eviction. Our work can use any of these works (such as DRRIP, EAF, etc.) to select the specific evicted data or insert the missed data into an appropriate position to overcome the drawbacks of LRU.

Another area of research works propose cost-aware replacement mechanisms. Jeong et al. [12] consider that cache misses have different costs. The cost may be latency, penalty, power consumption, bandwidth consumption, or any other property attached to a miss. They thus select cached data based on its cost instead of total misses. They protect cache lines with high cost. Qureshi et al. [28] show that isolated misses are costlier than parallel misses in MLP architectures. They thus propose Memory-level parallelism- (MLP) aware cache that tries to minimize costly isolated misses. However, our work mainly focuses on the asymmetric cost on top of hybrid memories and thus propose a new metric TMPKI, which takes the asymmetric miss cost into account. We dynamically adjust NVM line sizes to minimize the value of TMPKI, rather than minimizing costly NVM misses.

Another area of research works focus on write- induced interference, namely memory write requests compete with read requests for the available memory resources. Eager writeback [16] is the first proposal that increases the visibility of the write buffer by using the LLC to reduce write-induced interference. It writes back dirty cache blocks in the LRU position of the LLC sets whenever the bus is idle instead of waiting for the block to be evicted to reduce the memory traffic. VWQ technique [38] further uses more positions near LRU in the LLC. It regards a fraction of the LRU positions in the LLC as a "virtual write queue." Dirty cache blocks in the virtual write queue that target the same row buffer when mapping to the memory resource will be written back in a batch,

therefore reducing write-induced interference. Wang et al. [40] uses a last-write predictor based on sampling to predict the last-write cache blocks in LLC. They then can schedule these blocks and evicted dirty blocks together in the memory controller to reduce write-induced interference. Chang et al. [14] aggressively send out writeback requests that are expected to hit in DRAM row buffers before they would normally be evicted by the last-level cache replacement policy. Seshadri et al. [35] proposes the Dirty-Block Index (DBI), that decouples the dirty bit information from the main cache tag store. Using DBI, they can easily find the list of all dirty blocks of the same row and write them back together. Compared with [14], DBI generates a little overhead of tag lookups. The works described above essentially write more blocks in a batch to expand write scheduling space in the memory controller. Being different with these works, our work reduces the total number of NVM writebacks through “second chances” because the write latency of NVM is much longer.

Cache mechanisms for improving each application’s performance. These works mainly study on how to partition cache resources among multiple applications to reduce cache misses for each application.

Suh et al. [39] first propose a low-overhead partition scheme that estimates the miss patterns of each process at runtime, and dynamically partitions the cache among the processes that are executing simultaneously. Qureshi et al. [29] propose utility-based cache partitioning (UCP), which partitions the cache lines among the cores according to the usage of each core. Unlike [39], UCP separates the monitoring circuit from the main cache so it can obtain the application miss information independent of other concurrently running applications. Lee et al. [17] targets the CPU-GPU heterogeneous architectures. In such architecture, GPU cores can tolerate higher memory access latency compared with CPU cores. Therefore, they partition the cache depending on the cache sensitivity of the GPU application. When GPU application is found to be cache insensitive, they reduce GPU partition sizes.

Compared with these works, our work targets on the competitions between NVM and DRAM data as well as the impact of long NVM write latency on LLC performance. Our goal is to minimize the total LLC miss cost instead of reducing misses of each application. We thus do not consider the LLC utility of each application. Moreover, although UCP [29] is a widely used partition mechanism, it is not fit for the cache mechanisms on the top of hybrid memory. UCP implements cache partitions to obtain the maximum value for total cache hits. It does not consider the asymmetric miss cost for the hybrid memory. However, as shown in Section 2.2, the total number of misses cannot reflect the total miss cost well for the hybrid memory, because the load cost from one NVM data miss is much higher than that from one DRAM data miss. Therefore, we propose a new metric TMPKI, which takes the asymmetric miss cost into account, and partition cache spaces to maintain the TMPKI to be a small value.

6.3 LLC Mechanisms Towards NVM-based Memory

Since emerging NVMs are widely studied as potential alternatives for replacing DRAM, LLC mechanisms that target on NVM-based memories are also proposed. Since NVM write latencies are much longer than NVM read latencies, these mechanisms focus on how to avoid evicting dirty lines.

Zhang et al. [51] propose LLC cache mechanisms that set different priorities for clean and dirty lines and prioritize clean lines as victims. Zhou et al. [53] further propose two writeback-aware cache policies to hide the NVM write latency. Writeback-Aware Cache Partitioning (WCP) partitions the cache among applications to reduce the number of misses and writebacks. Write Queue Balancing (WQB) replacement spreads writeback traffic among write queues in the memory controller to further avoid stalling. Wang et al. [41] propose WADE which keeps highly reused dirty

cache blocks in the LLC. It predicts cachelines that are frequently written back and dynamically partitions cache sizes between frequent and non-frequent write-back cachelines.

Compared with these works, our experiment results show that prioritizing evicting clean lines as much as possible hurts the whole performance due to the significant increases of clean lines misses. We thus conduct experiments and observe that giving NVM dirty data only a second chance to reside in LLC is appropriate for most of applications. Moreover, our work targets on the LLC performance on top of hybrid memories. We thus first maintain the proportions of DRAM and NVM data in an appropriate range, which is not considered by these above works.

6.4 LLC Mechanisms Towards Hybrid Memory

The mechanisms described above are all based on the memories that only have a single type (i.e., DRAM or NVM). We take the first step to design a LLC mechanism (i.e., HAP-partition [43]) for hybrid memories. HAP-partition logically divides the cache into two partitions (i.e., DRAM lines and NVM lines) and restricts the NVM partition size in an appropriate range to guarantee TMPKI to be a small value. WBAR [49] then is proposed to address the problem of long NVM write latencies. Similarly to [51], WBAR sets different priorities for the four type data, clean DRAM data, dirty DRAM data, clean NVM data, and dirty NVM data, to reside in LLC. It keeps dirty NVM data in LLC for longer time than other types of data. However, experiment results in this article show that prioritizing evicting clean lines as much as possible hurts the whole performance due to the significant increases of clean lines misses. We conduct experiments and observe that giving NVM dirty data only a second chance to reside in LLC is appropriate for most of applications. We thus propose HAP-2-chance policy based on HAP-partition in this article. Moreover, compared with WBAR, our work can dynamically adjust the proportions of DRAM and NVM data instead of setting fixed priorities.

7 CONCLUSION

Current servers need large capacity main memory to hold a significant amount of data. Hybrid memories that combine DRAM and NVM are good candidates, since they can take advantage of the low latency of DRAM and the good scalability of NVM. However, hybrid memories also bring a new problem to shared LLC. Due to NVMs' longer latencies, cache misses cost from NVM data is more than that from DRAM data. Thus, a large number of NVM data misses greatly decrease LLC performance. Moreover, given the long NVM write latency, the write buffer in hybrid memories easily become fully occupied, and thus the evict cost significantly increases. Unfortunately, current cache mechanisms cannot address these challenges because they do not distinguish NVM from DRAM data in LLC.

In this article, we first propose a new performance metric, TMPKI, to describe LLC performance in hybrid memory systems. By adding a latency weight for each NVM data miss, TMPKI can exactly reflect the miss cost per-kilo instructions. We then propose a HAP mechanism. HAP includes two mechanisms, dynamic partition and 2-chances. Dynamic partition distinguishes between lines that cache NVM data and those that cache DRAM data. Then it logically divides the cache into two partitions and dynamically adjusts the NVM partition size in an appropriate range to guarantee TMPKI to be a small value all the time. Based on dynamic partition, 2-chances further gives NVM dirty data second chances to reside in LLC to reduce the evict cost.

We use two sets of experiments (i.e., micro and macro experiments) to evaluate our mechanisms. Experimental results show that HAP-partition and HAP (including both dynamic partition and 2-chances) results in, on average, 20.1% and 46.7% improvement in IPC, 9.0% and 21.9% reduction in energy against the traditional LRU. Moreover, HAP averagely improves performance by 9.3% and reduces energy consumption by 6.4% against a state-of-the-art cache mechanism (WBAR). We also

integrate HAP with a state-of-the-art page migration mechanism (RaPP). Experiment results show that HAP can be compatible well to page migration mechanisms. Exploiting the two technologies both achieves higher performance.

REFERENCES

- [1] 2014. The Graph 500 list. Retrieved August 2014 from <http://www.graph500.org/>.
- [2] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2 (2011), 1–7.
- [3] Santiago Bock, Bruce R. Childers, Rami Melhem, and Daniel Mossé. 2014. Concurrent page migration for mobile systems with OS-managed hybrid memory. In *Proceedings of ACM Conference on Computing Frontiers*. 31:1–31:10.
- [4] Chih-Chung Chang and Chih-Jen Lin. 2011. LIBSVM: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.* 2 (2011), 27:1–27:27.
- [5] Standard Performance Evaluation Corporation. 2011. SPEC CPU2006. Retrieved September 2011 from <http://www.spec.org/cpu2006/index.html>.
- [6] B. Dieny, R. Sousa, S. Bandiera, M. Souza, S. Auffret, B. Rodmacq, J. P. Nozieres, J. Herault, E. Gapihan, I. L. Prejbeanu, C. Portemont, K. Mackay, and B. Camb. 2011. Extended scalability and functionalities of MRAM based on thermally assisted writing. In *Proceedings of the International Electron Devices Meeting*. 1–3.
- [7] Alexandre P. Ferreira, Miao Zhou, Santiago Bock, Bruce Childers, Rami Melhem, and Daniel Mossé. 2010. Increasing PCM main memory lifetime. In *Proceedings of the Conference on Design, Automation and Test in Europe*. 914–919.
- [8] T. J. Ham, B. K. Chelepalli, N. Xue, and B. C. Lee. 2013. Disintegrated control for energy-efficient and heterogeneous memory systems. In *Proceedings of the International Symposium on High Performance Computer Architecture*. 424–435.
- [9] Ahmad Hassan, Hans Vandierendonck, and Dimitrios S. Nikolopoulos. 2015. Software-managed energy-efficient hybrid DRAM/NVM main memory. In *Proceedings of International Conference on Computing Frontiers*. Article 23, 8 pages.
- [10] Intel. 2016. 3D XPoint Unveiled-The Next Breakthrough in Memory Technology. Retrieve from <http://www.intel.com/content/www/us/en/architecture-and-technology/3d-xpoint-unveiled-video.html>.
- [11] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. 2010. High performance cache replacement using re-reference interval prediction (RRIP). In *Proceedings of the International Symposium on Computer Architecture*. 60–71.
- [12] J. Jeong and M. Dubois. 2003. Cost-sensitive cache replacement algorithms. In *Proceedings of the High-Performance Computer Architecture*. 327–337.
- [13] A. Kawahara, R. Azuma, Y. Ikeda, K. Kawai, Y. Katoh, K. Tanabe, T. Nakamura, Y. Sumimoto, N. Yamada, N. Nakai, S. Sakamoto, Y. Hayakawa, K. Tsuji, S. Yoneda, A. Himeno, K. Origasa, K. Shimakawa, T. Takagi, T. Mikawa, and K. Aono. 2012. An 8Mb multi-layered cross-point ReRAM macro with 443MB/s write throughput. In *Proceedings of the Solid-State Circuits Conference Digest of Technical Papers*. 432–434.
- [14] Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2010. *DRAM-aware Last-level Cache Writeback: Reducing Write-caused Interference in Memory Systems*. Technical Report. CMU.
- [15] Hyung Gyu Lee, Seungcheol Baek, C. Nicopoulos, and Jongman Kim. 2011. An energy- and performance-aware DRAM cache architecture for hybrid DRAM/PCM main memory systems. In *Proceedings of the International Conference on Computer Design*. 381–387.
- [16] H. H. S. Lee, G. S. Tyson, and M. K. Farrens. 2000. Eager writeback-a technique for improving bandwidth utilization. In *Proceedings of the International Symposium on Microarchitecture*. 11–21.
- [17] Jaekyu Lee and Hyesoon Kim. 2012. TAP: A TLP-aware cache management policy for a CPU-GPU heterogeneous architecture. In *Proceedings of the High Performance Computer Architecture*. 1–12.
- [18] Soyoon Lee, Hyokyung Bahn, and S. H. Noh. 2011. Characterizing memory write references for efficient management of hybrid PCM and DRAM memory. In *Proceedings of the Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. 168–175.
- [19] Dong Li, Jeffrey S. Vetter, Gabriel Marin, Collin McCurdy, Cristian Cira, Zhuo Liu, and Weikuan Yu. 2012. Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. In *Proceedings of the International Parallel and Distributed Processing Symposium*. 945–956.
- [20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the Programming Language Design and Implementation*. ACM, New York, NY, 190–200.
- [21] Piotr R. Luszczek, David H. Bailey, Jack J. Dongarra, Jeremy Kepner, Robert F. Lucas, Rolf Rabenseifner, and Daisuke Takahashi. 2006. The HPC Challenge (HPCC) benchmark suite. In *Proceedings of the Conference on Supercomputing*. ACM, Article 213.

- [22] M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero. 2007. MLP-aware dynamic cache partitioning. In *Proceedings of the Parallel Architecture and Compilation Techniques*. 418–418.
- [23] Onur Mutlu, Rachata Ausavarungnirun, Rachael A. Harding, Justin Meza, and HanBin Yoon. 2012. Row buffer locality aware caching policies for hybrid memories. In *Proceedings of the International Conference on Computer Design*. 337–344.
- [24] R. Narayanan, B. Ozisikyilmaz, J. Zambreno, G. Memik, and A Choudhary. 2006. MineBench: A benchmark suite for data mining workloads. In *Proceedings of the IEEE International Symposium on Workload Characterization*. 182–188.
- [25] T. Nirschl, J. B. Phipp, T. D. Happ, G. W. Burr, B. Rajendran, M.-H. Lee, A. Schrott, M. Yang, M. Breitwisch, C.-F. Chen, E. Joseph, M. Lamorey, R. Cheek, S.-H. Chen, S. Zaidi, S. Raoux, Y. C. Chen, Y. Zhu, R. Bergmann, H.-L. Lung, and C. Lam. Dec. 2007. Write strategies for 2 and 4-bit multi-level phase-change memory. In *Proceedings of the Electron Devices Meeting*. 461–464.
- [26] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive insertion policies for high performance caching. In *Proceedings of the International Symposium on Computer Architecture*. 381–391.
- [27] Moinuddin K. Qureshi, John Karidis, Michele Franceschini, Vijayalakshmi Srinivasan, Luis Lastras, and Bulent Abali. 2009. Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling. In *Proceedings of the International Symposium on Microarchitecture*. 14–23.
- [28] Moinuddin K. Qureshi, Daniel N. Lynch, Onur Mutlu, and Yale N. Patt. 2006. A case for MLP-aware cache replacement. In *Proceedings of the International Symposium on Computer Architecture*. 167–178.
- [29] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the International Symposium on Microarchitecture*. 423–432.
- [30] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the International Symposium on Computer Architecture*. 10.
- [31] Luiz E. Ramos, Eugene Gorbato, and Ricardo Bianchini. 2011. Page placement in hybrid memory systems. In *Proceedings of the International Conference on Supercomputing*. 85–95.
- [32] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. DRAMSim2: A cycle accurate memory system simulator. *IEEE Comput. Arch. Lett.* 10, 1 (2011), 16–19.
- [33] Reza Salkhordeh and Hossein Asadi. 2016. An operating system level data migration scheme in hybrid DRAM-NVM memory architecture. In *Proceedings of the Conference on Design, Automation & Test in Europe*. 936–941.
- [34] Chris Wilkerson Samira Khan, Alaa R. Alameldeen. 2014. Improving cache performance by exploiting read-write disparity. In *Proceedings of the High-Performance Computer Architecture*.
- [35] V. Seshadri, A. Bhowmick, O. Mutlu, P. B. Gibbons, M. A. Kozuch, and T. C. Mowry. 2014. The dirty-block index. In *Proceedings of the International Symposium on Computer Architecture (ISCA'14)*. 157–168.
- [36] Vivek Seshadri, Onur Mutlu, Michael A. Kozuch, and Todd C. Mowry. 2012. The evicted-address filter: A unified mechanism to address both cache pollution and thrashing. In *Proceedings of the Parallel Architectures and Compilation Techniques*. 355–366.
- [37] M. Still. 2005. *The Definitive Guide to ImageMagick*. Apress. Retrieved from <http://books.google.com.hk/books?id=6KVZ8Ya6a8cC>.
- [38] Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. 2010. The virtual write queue: Coordinating DRAM and last-level cache policies. In *Proceedings of the International Symposium on Computer Architecture*. ACM, 72–82.
- [39] G. E. Suh, L. Rudolph, and S. Devadas. 2004. Dynamic partitioning of shared cache memory. *J. Supercomput.* 28, 1 (2004), 7–26.
- [40] Z. Wang, S. M. Khan, and D. A. Jimnez. 2012. Improving writeback efficiency with decoupled last-write prediction. In *Proceedings of the International Symposium on Computer Architecture (ISCA'12)*. 309–320.
- [41] Zhe Wang, Shuchang Shan, Ting Cao, Junli Gu, Yi Xu, Shuai Mu, Yuan Xie, and Daniel A. Jiménez. 2013. WADE: Writeback-aware dynamic cache management for NVM-based main memory system. *ACM Trans. Arch. Code Optimiz.* 10 (Dec. 2013), 51:1–51:21.
- [42] W. Wei, D. Jiang, S. A. McKee, J. Xiong, and M. Chen. 2015. Exploiting program semantics to place data in hybrid memory. In *Proceedings of the International Conference on Parallel Architecture and Compilation*. 163–173.
- [43] W. Wei, D. Jiang, J. Xiong, and M. Chen. 2014. HAP: Hybrid-memory-aware partition in shared last-level cache. In *Proceeding of International Conference on Computer Design*. 28–35.

- [44] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. 1995. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*. 24–36.
- [45] Yuejian Xie and Gabriel H. Loh. 2009. PIPP: Promotion/insertion pseudo-partitioning of multi-core shared caches. In *Proceedings of the International Symposium on Computer Architecture*. 174–183.
- [46] Chun Jason Xue, Youtao Zhang, Yiran Chen, Guangyu Sun, J. Jianhua Yang, and Hai Li. 2011. Emerging non-volatile memories: Opportunities and challenges. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis*. 325–334.
- [47] Dong Ye, A. Pavuluri, C. A. Waldspurger, B. Tsang, B. Rychlik, and S. Woo. 2008. Prototyping a hybrid main memory using a virtual machine monitor. In *Proceedings of the International Conference on Computer Design*. 272–279.
- [48] Hanbin Yoon, Justin Meza, Rachata Ausavarungnirun, Rachael Harding, and Onur Mutlu. 2011. *Row Buffer Locality-Aware Data Placement in Hybrid Memories*. Technical Report No. 2011-005. Dept. of Electrical and Computer Engineering, Carnegie Mellon University.
- [49] Deshan Zhang, Lei Ju, Mengying Zhao, Xiang Gao, and Zhiping Jia. 2016. Write-back aware shared last-level cache management for hybrid main memory. In *Proceedings of the Annual Design Automation Conference*. 172:1–172:6.
- [50] Wangyuan Zhang and Tao Li. 2009. Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proceedings of the Parallel Architectures and Compilation Techniques*. 101–112.
- [51] X. Zhang, Q. Hu, D. Wang, C. Li, and H. Wang. 2011. A read-write aware replacement policy for phase change memory.
- [52] Y. Zhang and S. Swanson. 2015. A study of application performance with non-volatile main memory. In *Proceedings of the Symposium on Mass Storage Systems and Technologies*. 1–10.
- [53] Miao Zhou, Yu Du, Bruce Childers, Rami Melhem, and Daniel Mossé. 2012. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *ACM Trans. Arch. Code Optimiz.* 8, 4 (Jan. 2012), 53:1–53:21.
- [54] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the International Symposium on Computer Architecture*. 14–23.
- [55] Yuanyuan Zhou, James Philbin, and Kai Li. 2001. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the USENIX Annual Technical Conference*. 91–104.

Received December 2016; revised May 2017; accepted June 2017