Shared Last-level Cache Management for GPGPUs with Hybrid Main Memory

Guan Wang, Xiaojun Cai, Lei Ju, Chuanqi Zang, Mengying Zhao, Zhiping Jia
School of Computer Science and Technology, Shandong University, China
{xj_cai, julei, zhaomengying, jzp}@sdu.edu.cn

Abstract—**Memory intensive workloads become increasingly popular on general purpose graphics processing units (GPGPUs), and impose great challenges on the GPGPU memory subsystem design. On the other hand, with the recent development of non-volatile memory (NVM) technologies, hybrid memory combining both DRAM and NVM achieves high performance, low power and high density simultaneously, which provides a promising main memory design for GPGPUs. In this work, we explore the shared last-level cache management for GPGPUs with consideration of the underlying hybrid main memory. In order to improve the overall memory subsystem performance, we exploit the characteristics of both the asymmetric read/write latency of the hybrid main memory architecture, as well as the memory coalescing feature of GPGPU. In particular, to reduce the average cost of L2 cache misses, we prioritize cache blocks from DRAM or NVM based on observation that operations to NVM part of main memory have large impact on the system performance. Furthermore, the cache management scheme also integrates the GPU memory coalescing and cache bypassing techniques to improve the overall cache hit ratio. Experimental results show that in the context of a hybrid main memory system, our proposed L2 cache management policy improves performance against the traditional LRU policy and a state-of-the-art GPU cache strategy EABP [20] by up to 27.76% and 14%, respectively.**

I. Introduction

Graphics processing units (GPUs) were originally designed for graphics applications. Large data caches were not imported into these systems because graphics applications demonstrate relatively smaller opportunity of data reuse [1] and the massively parallel execution model was effective in hiding the long latency of accessing off-chip memory. However, this advantage becomes less appealing when the concept of general purpose graphics processing units (GPGPUs) has been widely accepted. Memory accesses in GPGPU programs can be very irregular [4], making the massively parallel execution model alone cannot effectively hide memory latency. So data caches have become a major concern for GPU architecture design. L2 cache is shared by all the processors on a single GPU. The shared nature and the massively parallel execution model significantly complicate the management of L2 cache on GPUs. Recently, significant research efforts have been devoted to designing effective cache management policies on multicore CPUs [7] [9] [12]. However, these techniques cannot be scaled to GPUs, because the number of threads on GPUs is more than that of multicore CPUs by 1–2 orders of magnitude and the memory pressure of GPGPU applications is also much higher. Such a nature of massively parallel execution poses great challenges for the last-level cache (LLC) design.

The extensive number of concurrently executing threads in GPUs necessitates a large main memory. However, the conventional DRAM exhibits limited scalability and high leakage power, making DRAM-only solution unattractive as GPU memory. The emerging non-volatile memory (NVM) technologies, such as phase-change memory (PCM) and spin-torque-transfer memory (STT-RAM), have better scalability and lower power consumption [2] while are usually characterized with longer write access latency, higher write energy, and limited endurance. Hybrid main memory can leverage the best characteristics of both DRAM and NVM to build a large-capacity, high-performance and energy-efficient main memory for GPUs. In this work, our hybrid main memory system is to consider both DRAM and NVM as the overall main memory address space [8].

Hybrid main memory brings a new challenge to the shared LLC in GPU architecture. Due to NVMs' longer latencies, the cache miss cost of NVM data is higher than that of DRAM data. The competition for LLC space between DRAM and NVM data significantly affects the cache performance. For example, NVM data are expected to have privilege of staying in cache for a longer time due to the miss cost consideration; however, if most cache space is used to cache NVM data, it will reduce many NVM data misses but dramatically increase DRAM data misses. As a result, cache performance may decrease. Therefore, the LLC management should carefully take the asymmetric cache miss cost into account to achieve globally optimal cache performance when GPGPUs adopt the hybrid main memory. The major contributions of this paper can be summarized as follows:

- We consider the asymmetric cache miss cost of L2 cache in hybrid memory system. As a result, we assign DRAM cache lines and NVM cache lines different insertion and promotion priorities to gain superior performance.
- We propose HAC, a hybrid main memory aware LLC management mechanism for GPGPUs, which includes an insertion strategy based on the effective addresses of memory coalescing combined with cache bypassing and a promotion method based on types of cache lines.
- We use different workloads to compare HAC with LRU and EABP [20]. The experimental results show that HAC improves performance by up to 27.76% and 12.78% on average. Furthermore, HAC's storage overhead is just 0.42% of the total L2 cache capacity.

II. Related Work

A. Cache Management

Targeting multithreaded processors, researchers have proposed a large amount of works about LLC management policies either by logically partitioning cache into multiple ways and then allocating a fixed way to each thread [7] or identifying the application-specific accessing patterns for prioritizing memory requests to be cached [9] [12]. Qureshi et al. [7] propose utility-based cache partitioning (UCP), which divides the cache depending on the reduction in cache misses that each application is likely to obtain. These works focus on multithreaded or multicore platforms in which the number of threads is significantly less than that of GPUs and are inappropriate to be implemented on GPUs.

In case of GPU, the cache is shared by tens of thousands of threads. Such an application-based coarse-granularity partition cannot be applied on fine-grained threads directly. Lee et al. [22] propose a prefetching mechanism to exploit the existence of common memory access behaviors among fine-grained threads. This technique is only suitable for regular access patterns. [20] develops a priority scheme to keep data that are the most likely to be reused in the cache lines and reorder the memory request from different warps according to the divergence behaviors to reduce the average warp stalled time. [13] proposes to tightly couple the thread scheduling mechanism with the cache management algorithms such that GPU cache pollution is minimized. Some work proposes cache bypassing in GPU to improve application performance [6] [15]. These policies are based on the traditional memory architecture without considering the influence of hybrid memory architecture.

B. Hybrid Main Memory

Most existing work on hybrid memory focuses on main memory of CPUs [8] [14] [16]. Ramos et al. [8] rely on memory controller to monitor popularity and write intensity of memory pages, based on which to migrate pages between DRAM and PCM. Lee et al. [14] propose a page-placement policy CLOCK-DWF to place frequently written data into DRAM, leaving rarely written ones into PCM, in order to reduce the number of write operations on PCM. Some studies have considered data placement for the hybrid memory system on GPU. [19] explores the use of hybrid memory as the GPU global memory, and investigates how to match memory design with massive parallelism of GPU devices and improve energy efficiency. [18] presents an adaptive data migration mechanism that exploits various memory access patterns of GPGPU applications to improve the memory bandwidth and reduce the power consumption.

These studies are almost based on data migration and bring additional overhead to cache management. [3] proposes a hybrid memory aware cache partitioning technique (HAP) to dynamically adjust the cache spaces for DRAM and NVM data. [5] proposes a write-back aware last-level cache management scheme for the hybrid main memory. However, these policies are based on the CPU architecture and can't take full advantage of the characteristics of the GPU. As far as we know that we are the first group to explore shared LLC replacement policy on the hybrid memory GPU architecture.

III. Background and Motivation

A. GPU Architecture

Fig. 1 shows an overview of GPU microarchitecture with shared L2 cache and hybrid memory. In each Streaming Multiprocessor (SM), one hardware warp scheduler manages all active warps. L2 cache is logically shared by all the SMs and physically implemented as several banks. Each L2 cache bank communicates with L1 cache of different cores through an interconnection network. Each L2 bank is connected to an off-chip memory channel through a dedicated memory controller using write-back policy. In the hybrid memory, we replace the half DRAM capacity with NVM per channel. The hardware modification is limited to the memory interface and controllers and no modification is required to the internal structures of the GPU processors and the memory arrays.

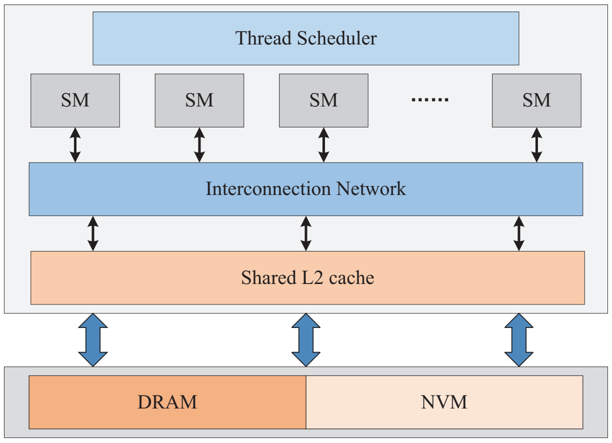

Fig. 1 GPU architecture with hybrid memory

B. Memory Coalescing

During GPU execution, 32 threads in each warp execute the same instruction on different data [21]. Many warps on the same SM execute in an interleaved manner to hide the memory latency. Given a memory instruction in a warp, the load/store unit (LD/ST) generates separate memory references to different memory addresses for each thread. Then, these memory requests will be merged into a certain number of transactions based on the distribution of memory addresses across the warp. This mechanism is called memory coalescing [21] and is strongly correlated with the spatial locality among threads in the same warp. A key observation of [20] is that cache lines which are accessed by the memory requests with more effective addresses have a higher chance to be reused. So we can design a cache management mechanism by assigning a higher priority to cache lines which are accessed by the memory requests with more effective addresses for a longer time.

C. Motivation

We select a few applications (shown in TABLE II) and run them in a full system simulator (described in Section V). For each application, we assign DRAM cache lines and NVM cache lines different insertion priorities when a cache access misses. We take an 8-way shared L2 cache as an example, where insertion positions from 0 to 7 can be selected upon a cache miss. As shown in Fig. 2, for MUM, the IPC dramatically increases while we give higher insertion priority to NVM cache lines than DRAM cache lines. However, for LIB and DCT, given the higher priority to both DRAM and NVM cache lines cannot consistently achieve better performance. We have similar observations for all applications. Consequently, with hybrid main memory, the cache mechanism can be improved by dynamically adjusting the insertion priorities of DRAM and NVM cache lines, and the mechanism should be adaptable to various workloads.

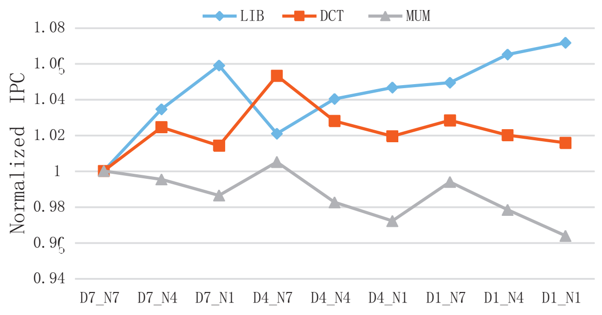

Fig. 2 Performance comparison of different insertion priorities between NVM and DRAM cache lines. (Di_Nj means DRAM and NVM cache lines insert to the ith and jth position of LRU list separately when a cache access misses.)

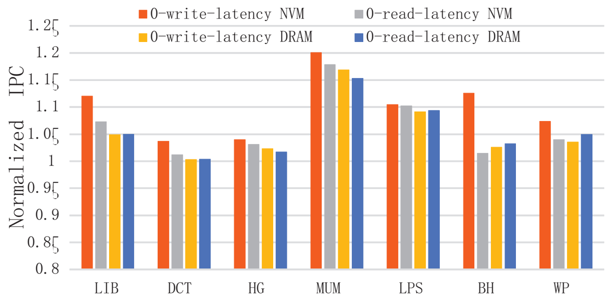

Fig. 3 Performance improvement with 0 memory access latency

Fig. 3 shows the performance for a set of workloads on a hybrid main memory with DRAM and NVM. In order to evaluate the impact of memory accesses on system performance, we separately set the NVM write latency, NVM read latency, DRAM write latency and DRAM read latency to 0, and then compare the IPCs. The results show that compared with the unmodified system, the IPC can be improved by 10%, 6.4%, 5.6% and 5.7%, respectively, indicating that the NVM write in main memory has the biggest impact on system performance. As a result, it is crucial to reduce the writeback of dirty cache lines to the NVM part of hybrid main memory. In GPU system, it is not enough to consider the writeback of dirty cache lines, because the IPC improvement is not obvious compared with the observation of [5] (described in Section V).

IV. Cache Management Policy

A. *Cache Classification*

A memory request has different numbers of effective addresses because of memory coalescing. The number of effective addresses range from 1 to 32. As we mentioned before, cache lines which are accessed by the memory requests with more effective addresses are more likely to be revisited. So we should assign a higher priority to cache lines which are accessed by the memory requests with more effective addresses. There will be 32 different priorities since it can have at most 32 effective addresses in total. For simplicity, we divide cache lines into three groups. Cache lines which are accessed by the memory requests with the number of effective addresses from 1 to 8 belong to low priority group. Cache lines which are accessed by the memory requests with the number of effective addresses from 24 to 32 belong to high priority group. The rest of cache lines belong to middle priority group.

In addition, we distinguish cache lines mapped from NVM and DRAM. Each cache line has a data type bit to identify whether the data is from DRAM or NVM. As shown in Fig. 4, in our implementation, when a new line enters into L2 cache, the cache controller checks its memory address. If the address belongs to DRAM space, then the data type bit is set as 0. Otherwise, it is set as 1. As we have analyzed before, the cost of one NVM data miss is much higher than that of one DRAM data miss. We thus distinguish DRAM and NVM cache lines with different priorities.

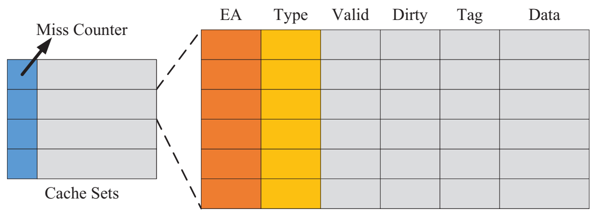

Fig. 4 GPU L2 cache organization

In this work, we consider the two characteristics to design an efficient LLC management policy. When a cache miss occurs, we categorize cache lines in L2 cache into 6 types: High-NVM(HN), High-DRAM(HD), Middle-NVM(MN), Middle-DRAM(MD), Low-NVM(LN) and Low-DRAM(LD). HN represents cache lines belonging to high priority group from NVM. Given the above-mentioned observations, we set the L2 cache lines priorities for different types of cache lines as follows: *HNP>HDP>MNP>MDP>LNP>LDP*. *HNP* means the insertion priority of HN cache lines. Similarly, when a cache hit occurs, we categorize cache lines in L2 cache into NVM and DRAM, respectively. We set the 2 types cache lines into different promotion priorities, such as *NP>DP*. *NP* means the promotion priority of NVM cache lines.

B. *Cache Management Policy*

For simplicity and low overhead, we allocate each type line a constant priority which is a static hybrid main memory aware LLC management policy. When a cache access misses, lines will be inserted into MRU, MRU-1, central, central-1, LRU+1 and LRU positions, if they are HN, HD, MN, MD, LN and LD, respectively. Otherwise, when a cache access hits, lines will be promoted by associativity/2 and associativity/4 (at most) over its current position, if they are NVM and DRAM, respectively. Fig. 5 illustrates an example of our static insertion and promotion strategy for an 8-way L2 cache, where we denote each cache line type with different symbols. Whenever a cache line needs to be evicted, the cache line at the LRU position is always selected.

Given the fact that an application may have distinct memory behaviors at different execution phases, it is better to modify the priority of each cache line type dynamically to cope with such changes. Compared with the coarse granularity dynamic cache management mechanism such as Set Dueling [9], we propose a fine-grained dynamic LLC management policy. As shown in Fig. 4, we add a *n*-bit saturating miss counter named *mc* to each cache set, where $n = (log_2A + 1)$ and *A* is the associativity of L2 cache. The initial value of *mc* is set to be 2^{n-1}. We add a *m*-bit flag called *EA* to each cache line, where $m = log_2(A/2)$ and $EA = A*(ea-1)/64$. We use *EA* to distinguish different priority groups. The number of effective address of each memory request is called *ea* for short. *EA* is updated at each memory request. We assume the associativity of the shared L2 cache is at least 8. The MRU position is at *A*-1, and an L2 cache eviction always chooses the LRU line at position 0.

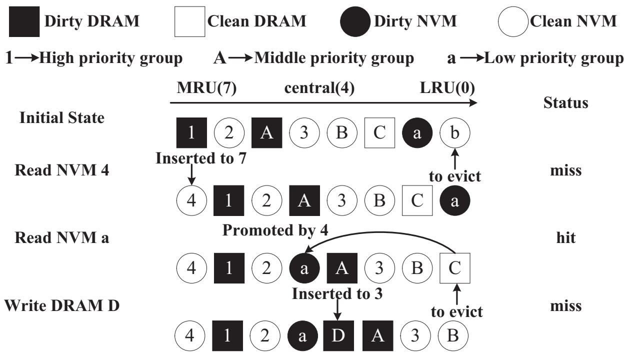

Fig. 5 Illustrative example of static insertion and promotion strategy

The hybrid main memory aware dynamic insertion strategy based on the effective addresses of memory coalescing combined with cache bypassing is illustrated in Algorithm 1. *req* denotes a memory request. *DNP* means the insertion priority of dirty NVM cache lines. If a memory request results in a cache miss, an incoming cache line needs to be inserted (according to write-back and write-allocation mechanism). If *req* is a writeback miss, we evict the cache line at the LRU position (Line 3), and the insertion location depends on which cache line type it belongs to (Line 4-8). Because the memory request is a writeback miss, we do not consider the effective addresses of memory coalescing to cache insertion priority. Otherwise, if evicted cache line is dirty NVM and its *EA* is greater than current request *EA*, we bypass current memory request (Line 10-11). Otherwise we evict the cache line at the LRU position (Line 13), and the insertion location is calculated according to both its priority group and cache line type (Line 15-21). Note that the saturating miss counter *mc* is decreased by 2 for a memory request miss on NVM (Line 16), and increased by 1 for a memory request miss on DRAM (Line 19), because the cost from one NVM data miss is much higher than that from one DRAM data miss. Intuitively, a larger counter value indicates more DRAM misses in the recent history, so that the priorities of DRAM lines during insertion should be raised.

The hybrid main memory aware dynamic promotion method based on types of cache lines is illustrated in Algorithm 2. If a memory request leads to a writeback hit for cache line *C* in L2 cache, *C* is set to be a dirty line, and the data is updated accordingly (Line 2-3). For the promotion sub-policy, if the hitting cache line *C* is a DRAM line, it is promoted by (at most) *A*/2+*mc*/4 over its current position *C.pos* (Line 5-7). Similarly, if *C* is a NVM line, its promotion position is calculated according to its memory device type (Line 8-10). According to the previous observations, we assign a higher promotion priority to NVM cache lines than DRAM cache lines.

Algorithm 1: HAC insertion policy

```
1: if req misses in cache
2:   if req is writeback then
3:     evict cache line C;
4:     if req is NVM cache line then
5:       DNP = A-1-mc/8; insert(req.line, DNP);
6:     else
7:       DDP = A/2+mc/4; insert(req.line, DDP);
8:     end if
9:   else   /*req is demand*/
10:    if evicted cache line C is dirty NVM and C.EA > req.EA then
11:      bypass req; break;
12:    else
13:      evict cache line C;
14:    end if
15:    if req is NVM cache line then
16:      mc = mc-2;
17:      CNP = A/2-mc/8+A*(ea-1)/64; insert(req.line, CNP);
18:    else
19:      mc = mc+1;
20:      CDP = A/8+mc/4+A*(ea-1)/64-1; insert(req.line, CDP);
21:    end if
22:  end if
23:end if
```

Algorithm 2: HAC promotion policy

```
1: if req hits in cache lines C then
2:   if req is writeback then
3:     C.dirty = 1; updateContent(C);
4:   end if
5:   if C is DRAM cache line then
6:     DP = C.pos+A/2+mc/4;
7:     promote(C, DP>MRU?MRU:DP);
8:   else
9:     NP = C.pos+A-mc/8-1;
10:    promote(C, NP>MRU?MRU:NP);
11:  end if
12:end if
```

V. Experiments

A. System Configuration

In the evaluation, we modify GPGPU-sim [17], a cycle accurate PTX-ISA simulator, to implement the hybrid memory. TABLE I specifies the configurations and parameters of the GPU processor and hybrid memory. The SMs, caches, and interconnection network are configured based on NVIDIA's GTX480. The SM is configured based on the NVIDIA's Fermi Architecture. Main characteristics of the simulated NVM are set according to [5]. In the hybrid memory, we reduce the DRAM capacity to 128MB per channel and replace the rest 128MB with PCM. In our dynamic hybrid main memory aware LLC management policy, we initialize the 5-bit saturating miss counter of each cache set to be 16.

B. Workloads

As shown in [20], the benchmarks can be classified into four classes. This classification clearly suggests that the performance-limiting factor varies for different applications. It has been clear that not all the GPGPU applications benefit from a larger L2 cache with traditional cache management policies. So we evaluate memory-intensive GPU applications from the NVIDIA CUDA SDK [11], Rodinia Benchmarks [10], LonestarGPU [4] and applications distributed with GPGPU-sim [17]. TABLE II lists the characteristics of these applications. For each workload, we first run 10 million instructions to warm up the caches and then run 2 billion instructions for the experiment.

TABLE I. GPU AND HYBRID MEMORY CONFIGURATION

Streaming Multiprocessors	15
Warp Size	32
Core Clock Frequency	700MHz
Threads per Multiprocessor	1024
Registers per Multiprocessor	32768
Shared Memory per Multiprocessor	48KB
Memory Channels	12
L2 Cache	768KB, 16-way, 128B lines, LRU
Bandwidth per Channel	4byte/cycle
DRAM Scheduling Policy	FR-FCFS
Hybrid Memory Timing (cycles) [5]	DRAM-tRCD:12, tRP:12, tRRDact: 6, tRRDpre: 6 NVM-tRCD:55, tRP:150, tRRDact: 6, tRRDpre: 22

TABLE II. BENCHMARK CHARACTERISTICS

Abbrev.	Benchmarks	Classification [20]
LIB [17]	libor	potentially cache-sensitive
DCT [11]	dct8x8	potentially cache-sensitive
HG [11]	histogram	potentially cache-sensitive
RAY [10]	Ray Tracing	potentially cache-sensitive
BFS [10]	Breadth First Search	potentially cache-sensitive
NN [10]	neuralnet	potentially cache-sensitive
MUM [10]	mummergpu	cache-sensitive
LPS [17]	laplace	cache-sensitive
BH [4]	BarnesHut	cache-sensitive
WP [17]	Weather Prediction	cache-sensitive

C. *Evaluation Results*

Fig. 6 shows performance comparisons among different policies for all workloads. The baseline is typical LRU cache replacement policy. The cache management policy in [20] and [5] is called EABP and WBAR, respectively. The following observations can be obtained from the results. On average, HAC improves performance by 12.78% compared with LRU. Compared with EABP, HAC improves performance by 5.25% on average (up to 14%). The best performance improvement comes from MUM, where HAC improves the IPC by 27.76% compared with LRU. The reason is that we distinguish DRAM and NVM in insertion priority and promotion priority, and we consider that the cache line which are accessed by the memory requests with more effective addresses has a higher priority in insertion policy. Compared with WBAR, HAC improves performance by 8.3% on average. Because WBAR can't take full advantage of the characteristics of the GPU. However, we also find that HAC improves IPCs by less than 5% for RAY and NN. The reason is that the two workloads are insensitive to the different latencies between DRAM and NVM. This insensitivity is due to the effectiveness of the massively parallel execution model in hiding the long memory latency.

We summarize the normalized total miss rate in Fig. 7. We observe that the miss rate declines dramatically with our hybrid main memory aware LLC management policy. Compared with LRU, HAC reduce total miss rate by 8.67% on average. HAC reduces total cache miss rate by 2.27% on average compared with EABP. Generally, the performance of benchmarks increases with a reduced miss rate. Specifically, cache-sensitive benchmarks (such as MUM, WP, and BH) exhibit good performance improvement, i.e., over 10%. For some other applications (such as RAY and NN), we achieve smaller performance enhancement by improving the cache performance.

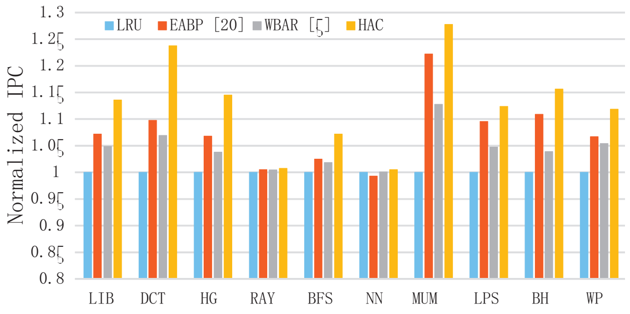

Fig. 6 IPC improvement over baseline LRU

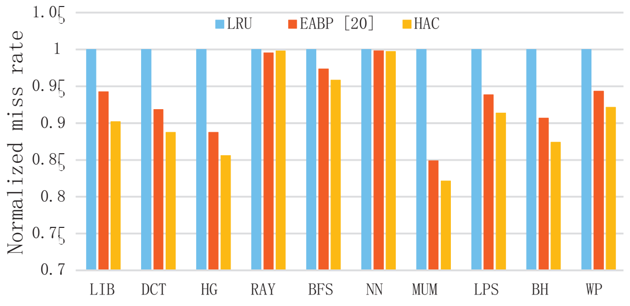

Fig. 7 Normalized total miss rate

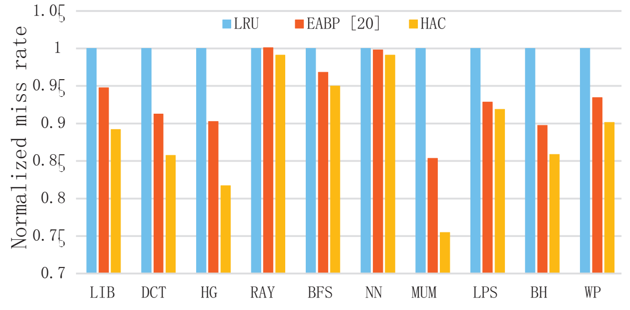

Fig. 8 Normalized NVM miss rate

Fig. 8 shows the normalized NVM miss rate. The observations are as follows. Compared with LRU, HAC reduces total and NVM miss rate by 8.67% and 10.68% on average, respectively. Compared with EABP, HAC reduces NVM miss rate by 4.12% on average by distinguishing DRAM and NVM cache lines. The results indicate that HAC reduces the cache misses from NVM data without dramatically increasing the cache misses from DRAM data. By doing so, raising the priority of NVM in L2 cache does not degrade the cache behavior in general and achieves performance improvement. We can also observe that HAC achieves more performance improvement in the workloads where HAC reduces more L2 cache misses. For example, for MUM, HAC reduces NVM miss rate and total miss rate by 24.52% and 17.87%, respectively.

Fig. 9 and Fig. 10 show the normalized DRAM and NVM writeback operations for all workloads. From the figures, we observe that HAC separately reduces DRAM writeback and NVM writeback by 43.05% and 50.85% against LRU, on average. Compared with EABP, HAC reduces DRAM writeback and NVM writeback by 21.34% and 29.64% on average, respectively. We observe that HAC reduces more NVM writeback than DRAM writeback compared with EABP. The reason is that we set relatively higher priority for NVM cache line than DRAM cache line, especially retaining NVM dirty data in cache for a much longer time. It benefits system performance as well as NVM lifetime to reduce NVM writeback.

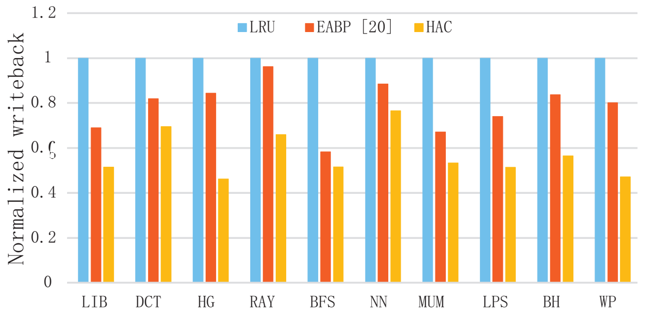

Fig. 9 Normalized DRAM writeback

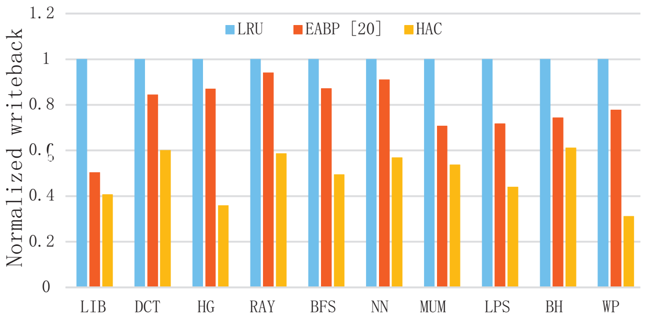

Fig. 10 Normalized NVM writeback

D. Hardware Overhead

For given cache configuration in our experimental setting as shown in TABLE I, with cache line size of 128B and associativity of 16, we add a 5-bit saturating miss counter for each cache set, and 1-bit memory type flag and 3-bit priority group flag for each cache line. There are total 384 cache sets and 6144 cache lines in L2 cache with the capacity of 768KB. Thus the total overhead is 3312B, which is approximately 0.42% of the L2 cache capacity.

VI. Conclusion

In this paper, we propose our light-weighted dynamic hybrid main memory aware LLC management policy (HAC), which includes a dynamic insertion strategy based on the effective addresses of memory coalescing combined with cache bypassing, and a dynamic promotion method based on types of cache lines. We compare HAC with LRU policy and related work EABP [20] on different workloads. Experiment results show that a 12.78% performance improvement can be achieved on average, whereas the maximum gain can be up to 27.76% with little storage overhead (0.42%).

Acknowledgment

This research is sponsored by the State Key Program of National Natural Science Foundation of China No. 61533011, Natural Science Foundation of China (NSFC) No. 61202015, National High-tech R&D Program of China(863 Program) No. 2015AA015304, Shandong Provincial Natural Science Foundation under Grant No. ZR2015FM001, the Fundamental Research Funds of Shandong University No.2015JC030.

References

[1] D. B. Kirk and W. M. Hwu, Programming Massively Parallel Processors:A Handson Approach. San Mateo, CA, USA: Morgan Kaufmann, 2010.

[2] C. J. Xue, Y. Zhang, Y. Chen, G. Sun, J. J. Yang, and H. Li, "Emerging non-volatile memories: opportunities and challenges," IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis, Oct 2011, pp. 325-334.

[3] W. Wei, D. Jiang, J. Xiong and M. Chen, "HAP: Hybrid-memory-Aware Partition in shared Last-Level Cache," IEEE International Conference on Computer Design, Oct 2014, pp. 28-35.

[4] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on GPUs," IEEE International Symposium on Workload Characterization, Nov 2012, pp. 141-151.

[5] D. S. Zhang, L. Ju, M. Y. Zhao, X. Gao, and Z. P. Jia, "Write-back Aware Shared Last-level Cache Management for Hybrid Main Memory," Design Automation Conference, June 2016, pp. 1-6.

[6] Y. Liang, X. Xie, G. Sun, and D. Chen, "An Efficient Compiler Framework for Cache Bypassing on GPUs," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 34(10), pp. 1677-1690, 2015.

[7] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," International Symposium on Microarchitecture, Jun 2006, pp. 423-432.

[8] L. Ramos, E. Gorbatov, and R. Bianchini, "Page placement in hybrid memory systems," International Conference on Supercomputing, May 2011, pp. 85-95.

[9] A. Jaleel, K. B. Theobald, S. C. Steely Jr., and J. Emer, "High performance cache replacement using Re-Reference Interval Prediction (RRIP)," International Symposium on Computer Architecture, Jun 2010, pp. 60-71.

[10] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, "Rodinia: a benchmark suite for heterogeneous computing," IEEE International Symposium on Workload Characterization, Oct 2009, pp. 44–54.

[11] NVIDIA, CUDA SDK, http://www.nvidia.com/object/cudasdks.html.

[12] Y. Xie and G. H. Loh, "PIPP: Promotion/Insertion Pseudo-Partitioning of multi-core shared caches," International Symposium on Computer Architecture, Jun 2009, pp. 174-183.

[13] D. Li, M. Rhu, D. R. Johnson, M. O'Connor, M. Erez, D. Burger, D. S. Fussell and S. W. Redder, "Priority-based cache allocation in throughput processors," International Symposium on High Performance Computer Architecture, Feb 2015, pp. 89-100.

[14] S. Lee, H. Bahn, and S. H. Noh, "CLOCK-DWF: A write-history-aware page replacement algorithm for hybrid pcm and dram memory architectures," IEEE Transactions on Computers, vol. 63(9), pp. 2187-2200, Sep. 2014.

[15] X. Xie, Y. Liang, Y. Wang, G. Sun and T. Wang, "Coordinated static and dynamic cache bypassing for GPUs," International Symposium on High Performance Computer Architecture, Feb 2015, pp. 76-88.

[16] Z. Wang, Z. Gu, and Z. L. Shao, "Optimizated Allocation of Data Variables to PCM/DRAM-based Hybrid Main Memory for Real-Time Embedded Systems," IEEE Embedded Systems Letters, vol. 6(3), pp. 61-64, Sept 2014.

[17] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," IEEE International Symposium on Performance Analysis of Systems and Software , 2009..

[18] J. Zhao and Y. Xie, "Optimizing bandwidth and power of graphics memory with hybrid memory technologies and adaptive data migration," International Conference on Computer-Aided Design, Nov 2012, pp. 81-87.

[19] B. Wang, B. Wu, D. Li, X. Shen, W. Yu, Y. Jiao and J. S. Vetter, "Exploring hybrid memory for GPU energy efficiency through software-hardware co-design," International Conference on Parallel Architectures and Compilation Techniques, Oct 2013, pp. 93-102.

[20] S. Mu, Y. Deng, Y. Chen, H. Li, J. Pan, W. Zhang and Z. Wang, "Orchestrating cache management and memory scheduling for GPGPU applications," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 22(8), pp. 1803-1814, Aug 2014.

[21] CUDA Programming Guide 5.0, NVIDIA, Santa Clara, CA, USA, 2012.

[22] J. Lee, N. B. Lakshmiarayana, H. Kim, and R. Vuduc, "Many-thread aware prefetching mechanisms for GPGPU application," International Symposium on Microarchitecture, Dec 2010, pp. 213-224.