Set Variation-aware Shared LLC Management for CPU-GPU Heterogeneous Architecture

Zhaoying Li   Lei Ju*   Hongjun Dai   Xin Li   Mengying Zhao   Zhiping Jia
School of Computer Science and Technology
Shandong University, China
E-mail: julei@sdu.edu.cn

***Abstract*—Heterogeneous CPU-GPU multiprocessor systems-on-chip (HMPSoC) becomes a popular architecture choice for high performance embedded systems, where shared last-level cache (LLC) management becomes a critical design consideration. We observe that within a sampling period, CPU and GPU may have distinct access behaviors over various LLC sets. In this work, we propose a light-weighted and fined-grained cache management policy to cope with the CPU-GPU access behavior variation among cache sets. In particular, CPU and GPU requests are prioritized disparately in each LLC set during cache block insertion and promotion, based on the per-core utility behaviors and a per-set CPU-GPU miss counter. Experimental results show that our LLC management scheme outperforms the two state-of-the-art schemes TAP-RRIP and LSP by 12.6% and 10.01%, respectively.**

I. Introduction

CPU-GPU Heterogeneous multiprocessor systems-on-chip (HMPSoC) becomes a popular trend for high performance embedded systems running computationally-intensive applications including video processing and deep learning. Many off-the-shelf HMPSoCs provide tightly-coupled shared communication and memory resources between CPU and GPU cores, e.g., Intel Skylake, AMD Zen, and NVIDIA Jetson TX1/TX2. Therefore, shared resource management is one of the crucial design considerations for such HMPSoC architectures.

The last-level cache is one of the pivotal shared memory resource which bridges the ever-increasing gap between fast processing units and the slow DRAM-based main memory. The widely used LRU replacement policy provides good performance for general-purpose applications running on uniprocessor platform. However, recent studies suggest that dynamic per-core LLC management policies for chip multiprocessor (CMP) architectures lead to overall system performance improvement, where the shared LLC resources are carefully allocated to different cores based on the runtime memory access behavior of the applications [6], [17], [23].

However, above-mentioned LLC management policies designed for homogeneous CMP may not lead to optimal performance on a CPU-GPU HMPSoC architecture. Literature work have identified two major characteristics for LLC management of CPU-GPU HMPSoC [9], [13], [20]. Firstly, due to the massively parallel execution model of GPU, most of the shared LLC cache lines are occupied by GPU data blocks in a pure recency-based cache management scheme, which results in significant performance degradation for applications running on CPU cores. Secondly, when the data requested by the current executing threads are not available, GPU allows fast context-switches between thread groups to hide the memory access latency. As a result, unlike most CPU applications whose performance is strongly correlated to the cache hit ratio, GPU application can be generally categorized into cache-sensitive or cache-insensitive ([9]), which must be taken into consideration in the LLC management policy. Furthermore, the recent study [20] on LLC management for CPU-GPU HMPSoC shows that the cache miss ratio has impact on the overall system performance even for cache-insensitive GPU benchmarks, since the off-chip main memory accesses resulted from the large number of cache misses may harm the performance of other concurrently running applications sharing the main memory.

There are two typical categories for LLC management policies, i.e., partition-based (i.e., UCP [17], TAP-UCP [9], and LSP [20]) or insertion-based (e.g., PIPP [23], RRIP [6], and TAP-RRIP [9]) approaches. Way partition-based method which associates different number of cache ways in a cache set to each core; while insertion-based method shares cache lines between all CPU or GPU cores, where memory requests from different cores are assigned with different priorities in the cache insertion and promotion policy. In general, insertion-based methods are more flexible when the cache set associativity is limited. For example, in case of a 16-way set associative cache which is typically used in today's LLC organization, a partition-based method has restricted design space even with a 4-core CPU and 1-core GPU.

Existing LLC management policies (either partition- or insertion-based) tend to assign the same number of cache ways or the same priority to each CPU/GPU cores uniformly across all cache sets. However, in this paper, we observe a significant fluctuation on the access frequency across different cache sets for any fixed amount of LLC requests from both CPU and GPU. Therefore, applying the same cache management policy to all cache sets lead to sub-optimal performance. Intuitively, for each cache set, it is preferable to allocate more cache resources to the processing units that currently have more access demand on this particular cache set.

In this work, we propose a set variation-aware insertion-based LLC management policy (SVAP) for CPU-GPU HMPSoC. The proposed policy cooperatively considers the per-core LLC utility and off-chip main memory latency, and prioritizes requests from each CPU/GPU core differently across variant cache sets. The contributions are as follows:

- We show empirically that both CPU and GPU have distinct access behaviors over different LLC sets for any fixed amount of LLC requests. Therefore, traditional approaches that give each core a uniform priority among all cache sets lead to sub-optimal solution.
- We propose a fined-grained and light-weighted cache management scheme SVAP to cope with the CPU-GPU access behavior variation across cache sets. In particular, CPU and GPU requests are prioritized disparately in each LLC set during cache block insertion and promotion, based on the per-core utility behaviors and as well as the per-set CPU/GPU request miss counter.
- Experimental results show that with a reduced logic and storage overhead, the proposed SVAP outperforms state-of-the-art cache management policies TAP-RRIP [9] and LSP [20] by 12.6% and 10.01%, respectively.

II. Related work

For shared LLC management for homogeneous CMP platforms, Qureshi et al. [17] partition cache ways between multiple cores based on the cache utility of an application (UCP), which represents the correlation between cache miss ratio and the allocated cache resources. Xie et al. [23] propose pseudo-partitioning scheme (PIPP), and gives application different promotion/insertion priority based on the partition decision according to the cache utility monitor. Jaleel et al. [6] logically categorize cache lines within a cache set into 4 priority levels, where the insertion or promotion of a cache line is determined by cache utility of the corresponding application as well as its current priority level.

For the cache management of GPU, Mu et al. [14] orchestrate both L2 cache and memory in a unified framework to keep the data that predicted to be mostly used into cache lines. Since massively parallel thread might lead to poor cache locality, [21] proposes a compiler-based framework to make use of the correlation between register allocation and thread-level parallelism. Li et al. Wang et al. [18] propose a divergence-aware cache (Da-Cache) management that can orchestrate L1 data cache management and warp scheduling together for GPUs. Liang et al. [11] develop a compiler framework to analyze the application code and select a set of load operations that bypassing these load operations at L1 cache leads to significant L2 cache traffic reduction. [10] uses an efficient locality monitoring mechanism to favor high reuse and short reuse distances in GPU cache management. Wang et al. [19] investigate the characteristics of both the asymmetric read/write latency of the hybrid main memory architecture for GPUs and give them different priorities in LLC. Xie et al. [22] identify the cache preference of global loads to decide cache or bypass through profiling when compiling and use dynamic cache bypassing to flexibly cache only a fraction of threads .

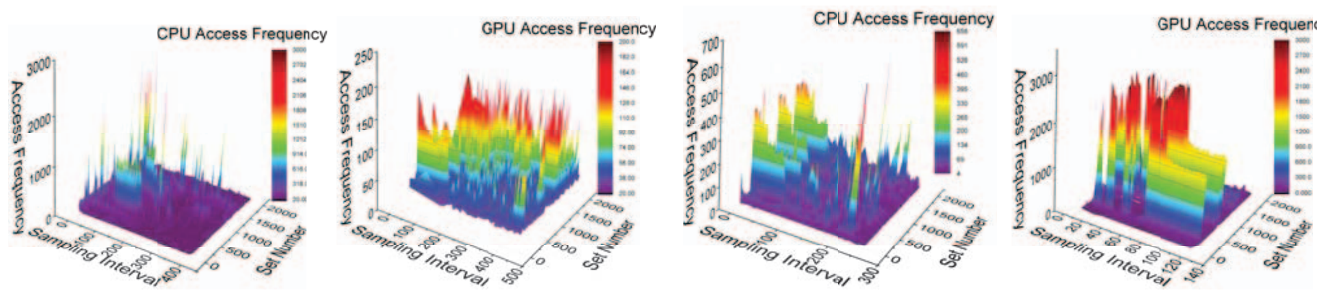
Due to the different characteristic between CPU and GPU, heterogeneous multi-core architecture brings new challenges to the shared resource management. Kim et al. [7] adopts hybrid main memory to allocate fast DRAM to CPU cores, and PRAM with longer write latency to GPU cores. Power et al. [15] propose a cache coherence protocol for CPU-GPU HMPSoC architecture. Forsberg et al. [4] develop a predictable software scheme to arbitrate main memory usage in heterogeneous architecture. On-chip network traffic is also taken into consideration to cope with massive GPU memory requests. Zhan et al. [24] design an STT-RAM based LLC for CPU-GPU HMPSoC, where asynchronous batch scheduling and priority based allocation schemes are proposed to reduce the on-chip communication latency.

For the shared LLC management of CPU-GPU HMPSoC, [9] shows that traditional cache management policies (e.g., [23], [6], and [17]) designed for homogeneous CMP lead to non-optimal performance. Lee et al. [9] first classify GPU into cache-sensitive or cache-insensitive by comparing the application's instruction-per-cycle (IPC) when allocating different cache resources to the GPU cores. Based on the categorization, [9] proposes an insertion-based scheme TAP-RRIP and a partition-based scheme TAP-UCP to allocation shared LLC resources between CPU and GPU cores. In order to further reduce the CPU-GPU cache interferences, Mekkat et al. [13] proposes a heterogeneous LLC management scheme HeLM which uses selective GPU cache bypassing based on the availability of GPU thread-level parallelism. Both TAP and HeLM target on improving the cache hit ratio of cache-sensitive CPU applications, and restrain the LLC resources allocated to the GPU cores. However, a recent work by Wang et al. [20] shows that the overall system performance is sensitive to the off-chip memory latency, where excessive GPU off-chip memory requests occupy the memory bandwidth and affect the performance of CPU applications. [20] proposes a latency sensitivity-based cache partitioning (LSP) framework which jointly considers per-core cache utility and the off-chip memory latency.

III. Motivation

Existing cache management policies (either partition-based or insertion-based) tend to allocate same amount of cache resources to a CPU or GPU core uniformly across all cache sets. In this paper, we show that there is a significant fluctuation on the access frequency across different cache sets for any fixed amount of LLC requests from both CPU and GPU cores.

Fig. 1 shows the per-set access frequency for the LLC under traditional LRU policy, where the HMPSoC consists of 4 CPU cores and 1 GPU cores with 16 SMs. Benchmark set I (Fig. 1(a)) contains 4 CPU benchmarks ("milc", "mcf", "bzip2", and "gcc") from SPEC CPU2006 [5], and 1 GPU benchmark ("kmeans") from Rodinia benchmark [3]. Among these benchmarks, "mcf", "bzip2", and "gcc" are considered as cache-sensitive CPU benchmarks, "milc" is a stream-like CPU benchmark, and "kmeans" is a cache-sensitive GPU benchmark. Similarly, Benchmark set II (Fig. 1(b)) contains 4 CPU benchmarks ("milc", "mcf", "hmmer", and "gcc") from SPEC CPU2006 [5], and 1 GPU benchmark ("pathfinder") from



(a) Benchmark set I

(b) Benchmark set II

Fig. 1. CPU and GPU access frequency per LLC set for two benchmark sets.

Rodinia benchmark [3], where "hammer" is a computation-sensitive benchmark, and "pathfinder" is a cache-insensitive GPU benchmark. We simulate the two benchmark sets with gem5-gpu simulator [16] to collect the per-set LLC access frequency from CPU and GPU, and display the accumulated frequency count for every 50000 LLC accesses (including both CPU and GPU accesses). Other simulation environment details can be found in Section V.

As we can observe from Fig. 1, for various benchmark combinations, both CPU and GPU LLC accesses show distinct temporal (over execution time) and spatial (across cache sets) variations. Traditional shared LLC management policies utilize various sampling mechanisms to capture the temporal variation which is caused by different execution phases (entering loops, procedures, kernels, etc). However, these polices ignores the spatial cache set variation, and tend to allocate same amount of cache resources to a given core uniformly across all cache sets. Intuitively, within a certain execution interval, if one type of core (CPU or GPU) generates less LLC bandwidth pressure on a particular cache set, more LLC resource of this set can be allocated to the other type of core which currently has significantly higher number of LLC requests. Therefore, a set variation-aware approach is crucial for optimal shared LLC management of CPU-GPU HMPSoC.

IV. Set Variation-aware LLC management

A. Framework

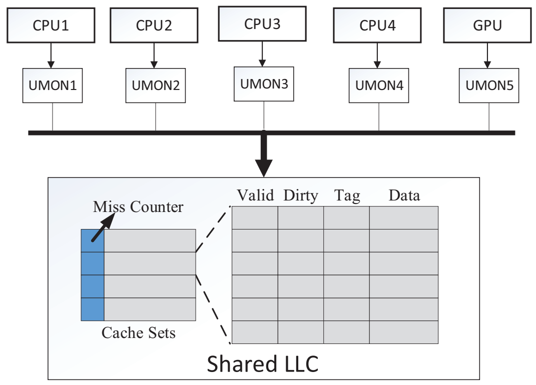


Fig. 2. Framework of proposed SVAP

The overall framework of the proposed set variation-aware LLC management policy (SVAP) is shown in Fig. 2. In SVAP, we monitor two types of information,, i.e., the core utility monitor ([17]) and the per-set miss counter, to dynamically determine the insertion/promotion priority for LLC blocks. In particular, the core utility monitor predicts the benefits of additional cache hits in case more cache resources are allocated to a core; while the miss counter associated to each LLC set denotes whether the resulted off-chip main memory latency due to cache misses from a particular set is dominated by CPU or GPU requests.

Utility monitor. The utility monitoring (UMON) is proposed by the utility-based cache partitioning (UCP) framework [17], and adopted in the TAP-UCP [9] and LSP [20] frameworks for LLC management of CPU-GPU HMPSoC. For each core, a dedicated UMON maintains an LRU stack for each sampling set during the sampling phase, where 1 sampling set is chosen among every 32 consecutive LLC sets. When an LLC hit occurs in a certain position of an LRU stack, the corresponding hit counter is increased by 1. Given the LRU stack property that a hit in an LRU managed cache containing N ways is guaranteed to be a hit in a M-way ($M > N$) cache with the same number of sets [12], a hit counter represents the number of misses saved by each recency position ([17]).

At the end of each sampling period, UCP performs an iterative partitioning algorithm to find a new partition between all cores. In each iteration, the core with the maximum number of hits will be chosen based on the UMON counters, till all ways are assigned to the cores.

Let A be the cache associativity, given a partition result $\{w_0, \ldots, w_{n-1}\}$ where n is total number of cores, w_i is the number of ways allocated to core c_i, and $\sum_{i=0}^{n} w_i = A$. Based on the partition obtained, we perform a pseudo-partitioning scheme to obtain the initial insertion and promotion priority for LLC requests from a CPU or GPU core c_i. The initial priority $InitPos^{c_i}$ for LLC requests from a particular CPU or GPU core c_i is defined as

$$InitPos^{c_i} = w_i$$

Intuitively, applications with higher marginal utility will be allocated with more cache ways in the partition-based policies. Similarly, requests from these applications will have a higher (initial) priority during insertion and promotion in our insertion-based SVAP.

Miss counter. Based on our observation that there is significant fluctuation on the LLC access frequency across different

Algorithm 1: SVAP insertion policy (when an LLC request req misses)

```
1: evictCache(0); /*evict cache block at LRU position*/
2: if $req$ is from CPU $c_i$ then
3:     if $mc > 0$ then
4:         $ipos = InitPos^{c_i} + mc * InitPos^{c_i}/A$;
5:     else
6:         $ipos = InitPos^{c_i}$;
7:     end if
8:     $mc = mc + 2$;
9: else if $req$ is from GPU $c_i$ then
10:    if $mc < 0$ then
11:        $ipos = InitPos^{c_i} - mc * InitPos^{c_i}/A$;
12:    else
13:        $ipos = InitPos^{c_i}$;
14:    end if
15:    $mc = mc - 1$;
16: end if
17: insert($req.block$,$ipos$);
```

Algorithm 2: SVAP promotion policy (when an LLC request req hits on cache line C) at position $C.pos$

```
1: if $req$ is from CPU $c_i$ then
2:     if $mc > 0$ then
3:         $ppos = C.pos + mc * InitPos^{c_i}/A$;
4:     else
5:         $ppos = C.pos + 1$;
6:     end if
7: else if $req$ is from GPU $c_i$ then
8:     if $mc < 0$ then
9:         $ppos = C.pos - mc * InitPos^{c_i}/A$;
10:    else
11:        $ppos = C.pos$;
12:    end if
13: end if
14: promote($C$, $ppos > MRU?MRU : ppos$);
```

cache sets within each sampling interval for both CPU and GPU, we associate a saturating miss counter to each cache set to capture whether the recent off-chip memory accesses are dominated by CPU or GPU requests. We believe the per-set miss counter is a good complement to the per-core utility based cache resource allocation. First, it captures the off-chip memory latency which may has substantial impact on the performance of both CPU and GPU applications ([20]). Moreover, if the number of misses in a particular cache set from CPU is significantly higher than that of GPU, we intuitively tend to allocate more cache resources to CPU cores which is likely to reduce more cache misses (and vice versa). Such intuition becomes especially effective when combined with the utility information, i.e., we should allocate more resources to a CPU/GPU core whose marginal utility is high but currently suffers more cache misses in a particular cache set (due to access frequency variation among cache sets). On the other hand, a streaming application with low marginal utility should

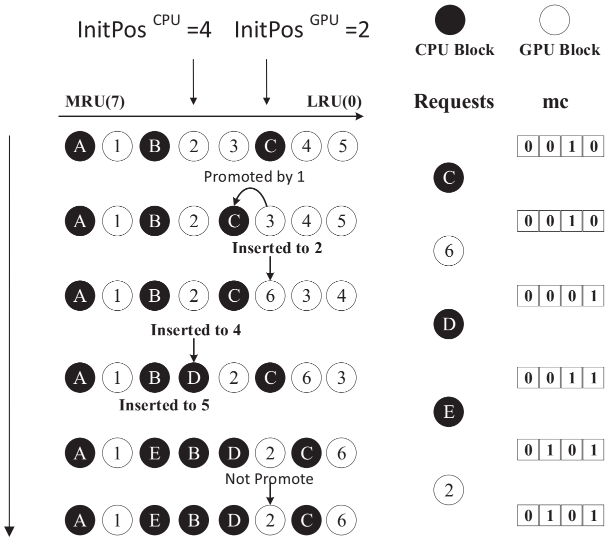


Fig. 3. An illustrative example of SVAP.

not be given a high cache insertion and promotion priority even if it is suffering a lot of cache misses.

As shown in Fig. 2, an n-bit saturating counter mc is associated with each cache set, where $n = (log_2 A + 1)$ for an A-way associative LLC. Since GPU cores with massively parallel execution model typically have significantly higher number of memory accesses compared to CPU cores, we perform a bias update on the miss counter to balance between the impact of CPU and GPU cache misses. The mc is decreased by 1 for each GPU cache miss in the corresponding cache set. On the other hand, the counter is increment by 2 for each CPU cache miss. Therefore, a large positive mc value indicates that majority of recent cache misses from this set are CPU requests; and a negative mc value indicates GPU dominate off-chip memory accesses due to misses from this set.

B. Insertion and Promotion Policy

Algorithm 1 shows the proposed SVAP insertion policy, where req denotes the memory block loaded into LLC due to a cache miss from core c_i. First, the cache block at the LRU position is evicted from the cache (line 1). For a CPU cache miss (line 2-8), if recently we observe more CPU cache misses in this set ($mc > 0$), the insertion position is calculated based on the $InitPos^{c_i}$ (determined by the utility monitor as in Section IV-A) and mc counter value (line 4). Otherwise, if the cache misses of this set is not dominated by CPU requests, we rely on the utility monitor to determine the initial insertion position (line 6). The mc counter is then updated accordingly (line 8). Similar actions are performed in case of a GPU miss (line 9-16). Finally, the corresponding memory block is inserted into the cache set with the calculated position.

Similar to the insertion policy, the proposed promotion scheme for a cache hit on a request req from core c_i is presented in Algorithm 2. Note that the corresponding cache line C is promoted at least by 1 for a CPU cache hit (line 5). While for GPU cache hit in case GPU cache miss is not dominating ($mc > 0$), the hit cache line is not promoted in order to favor the cache-sensitive CPU applications (line 11).

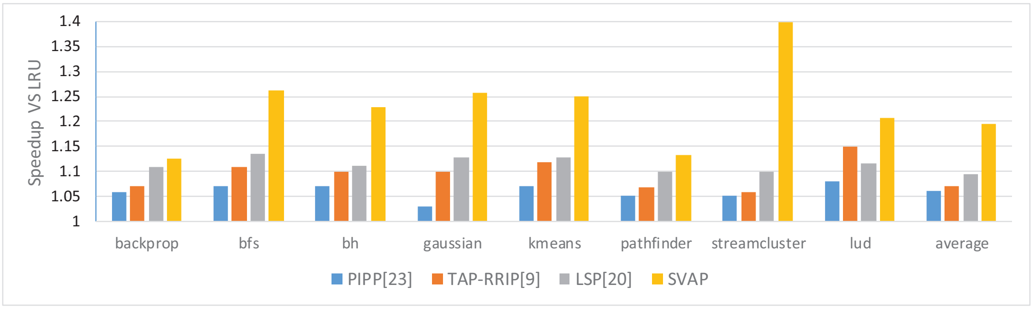


Fig. 4. Experimental results for 16-way associative LLC.

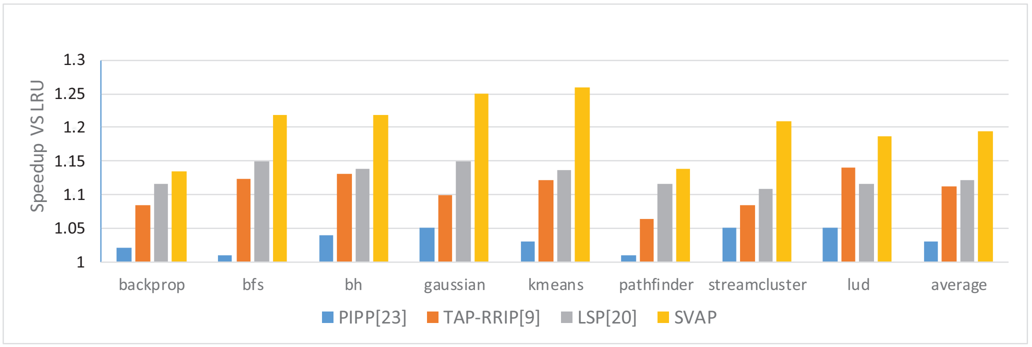


Fig. 5. Experimental results for 32-way associative LLC.

TABLE I
SYSTEM CONFIGURATIONS.

CPU	4-cores, Out-of-Order, 2GHz 32KB private L1 I/D-cache, 8-way, 2 cycles latency 256KB private L2, 8-way, 6 cycle latency
GPU	16 SMs, 4KB private I-cache, 4-way, 32KB private D-cache,8-way,2-cycle latency
LLC(shared)	8MB, 16 or 32-way, 128B/line, 22-cycle latency
DRAM	4GB, DDR3-1333, FR-FCFS, 2 controllers, 16 Banks

TABLE II
CPU AND GPU BENCHMARKS.

Core	Type	Benchmarks
CPU	cache-sensitive	bzip2, gcc, mcf, lbm
	cache-insensitive	hmmer, sjeng, gromacs
	Streaming	milc, zeusmp
GPU	cache-sensitive	bfs, bh, kmeans, gaussian
	cache-insensitive	backprop, pathfinder, lud, streamcluster

C. Illustrative example

Fig. 3 presents an illustrative example of the proposed SVAP framework with an 8-way shared LLC set. We assume that the utility monitor decides the initial priority for the CPU core and GPU core is 4 and 2, respectively (range from 0 to 7). In the given LLC request trace, CPU first makes a request and it hits Block C. According to line 3 of Algorithm 2, we promote the block by $2 * 4/8 = 1$ position. The next request from GPU for Block 6 is a cache miss. According to line 13 and 15 of Algorithm 1, we insert the new cache block at position 2, and decrease the mc by 1. The next two CPU misses evict the cache line at LRU position, insert block D and E into the cache set, and update the mc counter (line 1, 4 and 8 of Algorithm 1). Finally, the GPU request for Block 2 leads to a cache hit, but no promotion is performed given the positive value of mc (line 11 of Algorithm 2).

V. EXPERIMENTAL EVALUATION

A. Experimental Setup

We use gem5-gpu simulator [16] in our experimental evaluation, which integrates GPGPU-Sim [1] with gem5 [2]. Gem5-gpu use Ruby to model both the function and timing of memory accesses. We use built-in three-level MESI cache coherency protocol. We run the experiment for both 16-way and 32-way associative LLC. TableI summarize our simulation configurations.

Table II lists the CPU and GPU benchmarks we have used in the evaluation, where CPU benchmarks are selected from SPEC CPU2006 [5], and GPU benchmarks are selected from Rodinia [3] and Lonestar [8] benchmark sets. We form 8 workloads from these benchmarks, each workload contains 1 GPU benchmark and 4 randomly chosen CPU benchmarks. We use the corresponding GPU benchmark as the name for each workload. For each workload, we first run the workload for 80 million cycles to warmup the cache, and then collect the experimental results until the first benchmark completes.

B. Evaluation Results

We use the standard LRU policy as the baseline in our experimental evaluation. We evaluate the proposed SVAP w.r.t. LRU, PIPP [23], TAP-RRIP [9], and LSP [20], under 16-way and 32-way shared LLC with 8MB capacity.

Fig. 4 shows the evaluation results for the 16-way associative LLC. Compared with LRU, PIPP, TAP-RRIP, and LSP, the proposed SVAP leads to 19.6%, 13.6%, 12.6%, and 10.01% average speedup, respectively. We achieve up-to

15.2% speedup compared to LSP in the workload "bfs". Although we do not perform any GPU cache or off-chip latency sensitivity monitoring as in TAP-RRIP or LSP, experimental results show that the proposed scheme outperform existing approaches for both cache-sensitive and cache-insensitive GPU benchmarks.

The experimental results for 32-way associative LLC is shown in Fig. 5. Compared with LRU, PIPP, TAP-RRIP, and LSP, the proposed SVAP leads to 19.4%, 16.4%, 8.3%, and 7.1% average speedup, respectively. We achieve up-to 12.1% speedup compared to LSP in the workload "kmeans".

C. Hardware Cost

In this work, assume cache line size of 128B and associativity of 32, we require a 6-bit saturating miss counter for each cache set. And the storage overhead of the miss counter is 6/(32*128*8), which is approximately 0.02% of the LLC capacity. TAP-RRIP, LSP, and SVAP all use UMON to monitor the per-core LLC marginal utility. With 4 CPU cores and 1 GPU core, the overhead of UMON is 9.375KB ([17]). For an 8MB shared LLC, the additional overhead of SVAP is 1.5KB, compared to the storage overhead of TAP-RRIP and LSP are 16KB and 2KB, respectively. Compared with existing schemes which usually monitor GPU cache sensitivity, the proposed SVAP requires simplified control logic as well.

VI. Conclusion

In this work, we show our observation that both CPU and GPU accesses frequencies have distinct spatial variations across LLC sets. We propose a light-weighted set variation-aware insertion-based LLC management policy (SVAP) for CPU-GPU HMPSoC. Experimental results show that the proposed framework outperforms state-of-the-art LLC management schemes for CPU-GPU HMPSoC architectures.

VII. Acknowledgments

This research is sponsored by the Natural Science Foundation of China (NSFC) under Grant No. 61533011 and 61602274; Shandong Provincial Natural Science Foundation No. ZR2017MF034, ZR2015FM001, and ZR2014FM014; the Independent Innovation Engineering of Shandong Province under Grant No. 2017CXGC0703; and the Science & Technology Development Program of Shandong Province No. 2015GGX106002.

References

[1] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 163–174. IEEE, 2009.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[3] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. Ieee, 2009.

[4] B. Forsberg, A. Marongiu, and L. Benini. Gpuguard: Towards supporting a predictable execution model for heterogeneous soc. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 318–321. IEEE, 2017.

[5] J. L. Henning. SPEC CPU 2006 benchmark descriptions. *ACM SIGARCH Computer Architecture News*, 34(4):1–17, 2006.

[6] A. Jaleel, K. B. Theobald, S. C. Steely Jr, and J. Emer. High performance cache replacement using re-reference interval prediction (rrip). In *ACM SIGARCH Computer Architecture News*, volume 38, pages 60–71. ACM, 2010.

[7] D. Kim, S. Lee, J. Chung, D. H. Kim, D. H. Woo, S. Yoo, and S. Lee. Hybrid dram/pram-based main memory for single-chip cpu/gpu. In *Proceedings of the 49th Annual Design Automation Conference*, pages 888–896. ACM, 2012.

[8] M. Kulkarni, M. Burtscher, C. Casçaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *IEEE International Symposium on Performance Analysis of Systems and Software*, pages 65–76, 2009.

[9] J. Lee and H. Kim. Tap: A tlp-aware cache management policy for a cpu-gpu heterogeneous architecture. In *IEEE 18th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12. IEEE, 2012.

[10] C. Li, S. L. Song, H. Dai, A. Sidelnik, S. K. S. Hari, and H. Zhou. Locality-driven dynamic gpu cache bypassing. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 67–77. ACM, 2015.

[11] 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*, 34(10):1677–1690, 2015.

[12] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[13] V. Mekkat, A. Holey, P.-C. Yew, and A. Zhai. Managing shared last-level cache in a heterogeneous multicore processor. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pages 225–234. IEEE Press, 2013.

[14] 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*, 22(8):1803–1814, 2014.

[15] J. Power, A. Basu, J. Gu, S. Puthoor, B. M. Beckmann, M. D. Hill, S. K. Reinhardt, and D. A. Wood. Heterogeneous system coherence for integrated cpu-gpu systems. In *46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 457–467. IEEE, 2013.

[16] J. Power, J. Hestness, M. S. Orr, M. D. Hill, and D. A. Wood. gem5-gpu: A heterogeneous cpu-gpu simulator. *IEEE Computer Architecture Letters*, 14(1):34–36, 2015.

[17] M. K. Qureshi and Y. N. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *IEEE/ACM International Symposium on Microarchitecture*, pages 423–432. IEEE, 2006.

[18] B. Wang, W. Yu, X.-H. Sun, and X. Wang. Dacache: Memory divergence-aware gpu cache management. In *29th ACM on International Conference on Supercomputing*, pages 89–98. ACM, 2015.

[19] G. Wang, X. Cai, L. Ju, C. Zang, M. Zhao, and Z. Jia. Shared last-level cache management for gpgpus with hybrid main memory. In *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 25–30. IEEE, 2017.

[20] P.-H. Wang, C.-H. Li, and C.-L. Yang. Latency sensitivity-based cache partitioning for heterogeneous multi-core architecture. In *Design Automation Conference (DAC)*, pages 1–6. IEEE, 2016.

[21] X. Xie, Y. Liang, X. Li, Y. Wu, G. Sun, T. Wang, and D. Fan. Enabling coordinated register allocation and thread-level parallelism optimization for gpus. In *Proceedings of the 48th International Symposium on Microarchitecture*, pages 395–406. ACM, 2015.

[22] X. Xie, Y. Liang, Y. Wang, G. Sun, and T. Wang. Coordinated static and dynamic cache bypassing for gpus. In *International Symposium on High Performance Computer Architecture (HPCA)*, pages 76–88, 2015.

[23] Y. Xie and G. H. Loh. Pipp: promotion/insertion pseudo-partitioning of multi-core shared caches. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 174–183. ACM, 2009.

[24] J. Zhan, O. Kayıran, G. H. Loh, C. R. Das, and Y. Xie. Oscar: Orchestrating stt-ram cache traffic for heterogeneous cpu-gpu architectures. In *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, pages 1–13. IEEE, 2016.