JCSE Journal of Computing Science and Engineering

Regular Paper
Journal of Computing Science and Engineering,
Vol. 14, No. 3, September 2020, pp. 89-101

Exploring Time-Predictable and High-Performance Last-Level Caches for Hard Real-Time Integrated CPU-GPU Processors

Xin Wang
Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, VA, USA
wangx44@vcu.edu

Wei Zhang*
Department of Computer Science and Engineering, University of Louisville, Louisville, KY, USA
wei.zhang@louisville.edu

Abstract
Time predictability is crucial for hard real-time and safety-critical systems. In an integrated CPU-GPU (graphic processing units) architecture, the shared last-level cache (LLC) can cause a large number of interferences between CPU and GPU LLC accesses with diverse patterns and characteristics, which can significantly impact the performance and time predictability of both CPUs and GPUs. In this paper, we explore cache partitioning, locking, and a combination of them to make the LLC time-predictable for integrated CPU-GPUs while achieving high performance. By evaluating these LLC management approaches, we can provide real-time system developers recommendations on the most effective time-predictable LLC designs for heterogeneous CPU-GPU multicore processors.

Category: Cloud Computing / High Performance Computing

Keywords: GPU; Cache partitioning; Cache locking; Real-time systems; Integrated CPU-GPU

I. INTRODUCTION

Graphic processing units (GPUs), originally designed for graphic computations, have become a popular computing platform to accelerate high-performance and data-parallel applications. The massively parallel processing capability and enhanced energy efficiency of GPUs can potentially benefit parallel real-time applications such as autonomous navigation and medical data processing. All these applications have stringent deadlines and require high system throughput, thereby rendering GPUs as ideal computing engines. Prior work [1] has shown that GPUs can benefit real-time applications by reducing the response time by three times or more.

GPUs for desktops and servers are typically on different dies from the CPU. In such a discrete architecture, the memory hierarchies of the CPU and the GPU are private and separated, and communication between them is orchestrated by the application by copying the data between the CPU and the GPU via the PCIe bus. To remove the performance and energy overheads of copying data back and forth, a trend towards the heterogeneous computing architecture that integrates the CPU and the GPU on the same die has emerged. Recent examples include Intel's Sandy Bridge, AMD's accelerated processing unit (APU), NVIDIA's Denver, etc. The integrated CPU-

Open Access http://dx.doi.org/10.5626/JCSE.2020.14.3.89 http://jcse.kiise.org

Received 12 July 2020; **Revised** 26 July 2020; **Accepted** 07 August 2020
*Corresponding Author

GPU systems are particularly attractive for real-time systems as they can provide high throughput and efficient data sharing, and have stringent constraints on energy consumption, form factor, and cost.

To safely accelerate hard real-time applications by using the integrated CPU-GPUs, designers must be able to estimate the worst-case execution time (WCET) of the tasks running on CPU and GPU cores. The WCET provides a basis for schedulability analysis to ensure that the hard deadlines are met. However, many performance-oriented architectural features of modern microprocessors make it very hard, if not impossible, to accurately and safely derive the WCET, especially for the heterogeneous multicore processors consisting of both CPU and GPU cores. While the WCET analysis for caches in the context of single-core processors has been thoroughly studied in the recent decades, the architectural design shifting to multicores and heterogeneous CPU-GPU multicores makes it much harder and more complicated to compute the WCET tightly due to the tremendous amount of interferences between the homogeneous and/or heterogeneous cores.

In the past two decades, researchers have studied time-predictable architectural designs [2, 3] to mitigate the complexity of WCET analysis and enable hard real-time computing on high-performance processors. For example, Suhendra and Mitra [4] and Paolieri et al. [5] studied time-predictable shared caches on homogeneous multicores with only CPUs. However, varying from homogeneous multicores, the CPU and GPU cores on the heterogeneous CPU-GPU processors have significant different cache access patterns, demands, and performance characteristics. In general, GPUs tend to consume more LLC (last-level cache) resources than CPUs due to their massive threads and higher data accessing activity, leading to significant degradation in the performance CPU. In addition, in contrast to CPU data accesses, GPU data accesses may typically exhibit much less temporal and/or spatial locality, yet the GPU cache access latency may be largely hidden, taking advantage of the high thread-level parallelism of GPU kernels. Consequently, the traditional time-predictable caches designed for CPUs may not be suitable for the heterogeneous CPU-GPU processors.

The integrated CPU-GPU architecture that was considered in this paper is depicted in Fig. 1. In the integrated CPU-GPU architecture, both CPU and GPU cores have their own private L1 (and L2 caches for CPUs) and share the LLC, memory controllers, and on-chip interconnection network. While there can be interferences in all these three shared resources, this paper focuses on exploring time-predictable LLC designs, which is the first step towards building a fully time-predictable integrated CPU-GPU processor for hard real-time computing. To reduce the complexity of WCET analysis and support compositional timing analysis, it is crucial to eliminate

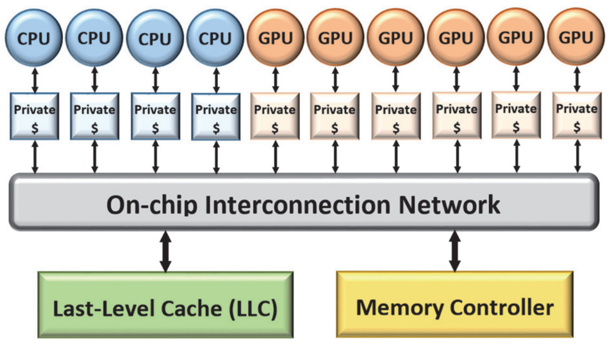


Fig. 1. Heterogeneous CPU-GPU multicore architecture.

the inter-core interferences in the shared LLC on heterogeneous multicores. Considering the diverse characteristics of CPU and GPU architectures and programs, we explore several LLC management techniques to improve time predictability while achieving as high performance as possible, including cache partitioning, locking, and a combination of them to achieve caching time predictability for both CPUs and GPUs. Cache partitioning is done to separate CPU and GPU cache blocks to minimize the interferences. The LLC space is partitioned into two parts to serve CPU and GPU threads separately, thus both CPU and GPU cores have their private LLC logically, thereby making it possible to avoid the interferences between CPU and GPU data accesses. Consequently, the cache partitioning enables the stable availability of shared LLC space for CPU and GPU cores guaranteeing that the WCET analysis can be done on the CPU or GPU cache partition independently, which can leverage existing cache timing analysis techniques [5-7]. Cache locking is another method to achieve time predictability for cache memories [8]. With cache locking, selected data are allowed to be locked into the cache, which cannot be evicted unless they are unlocked. While both cache partitioning and cache locking have been studied extensively for CPUs, their effectiveness on GPUs and performance impact on the integrated CPU-GPU remain unknown. Our study aims at addressing these issues.

In this paper, we explore possible design choices for a time-predictable and high-performance LLC on the integrated CPU-GPU architecture. While all these design options can improve the time predictability of the LLC, we also quantitatively compare their impact on the performance for both CPU and GPU programs. This study can provide interesting LLC design options and guidelines for real-time application programmers to accelerate their programs on integrated CPU-GPU processors.

The rest of this paper is organized as follows. Section II reviews previous work and Section III shows the motivation behind the design of this paper. Section IV introduces shared LLC management mechanisms that were implemented and describes cache partitioning and locking algorithms for heterogeneous CPU-GPU multicore processors. Section V presents the methodology and hardware details and Section VI provides the experimental results. Finally, we conclude this work in Section VII.

II. RELATED WORK

The shared LLC in heterogeneous CPU-GPU multicore processors leads to a conflict between CPU and GPU applications. GPU cores are more powerful to capture LLC space than CPU cores due to the high frequency of data accesses, thus CPU blocks are always evicted by GPU blocks thereby resulting in unfair LLC sharing and significant performance loss of CPU application. Cache partition and locking are two solutions to overcome this problem.

Researchers have studied the resource sharing problems for multicore processors. Partitioning of the algorithms has been used to monitor the cache accesses to identify proper partitions for maximizing performance [9] or fairness [10] for multicore processors. Qureshi and Patt [11] extended the work in [9, 10] to develop a utility-based cache partitioning (UCP) that obtains the information about the utility of cache resource by runtime monitoring. These proposals predict the number of cache misses of each application with various partitioning percentages, which then choose the best partition that results in the least amount of cache misses. Moreto et al. [12] considered the memory level parallelism (MLP) of each cache access to improve throughput. Kedar et al. [13] presented a novel cache architecture termed SPACE (semi-partitioned cache) that makes it possible to maintain the predictability of the execution time of the parallel threads while reducing the overall energy consumption of the system.

For heterogeneous multicore processors with GPUs, Lee and Kim [14] introduced a core-sampling mechanism called TLP-aware cache management policy to detect the manner in which caching affects the performance of general-purpose GPU (GPGPU) applications. Mekkat et al. [15] proposed heterogeneous LLC management (HeLM) that considered the advantage of the GPU's tolerance for memory access latency. Woo and Lee [16] and Yang et al. [17] proposed to exploit the GPU or CPU respectively to prefetch data for boosting the performance of the integrated CPU-GPU architectures. Wang et al. [18] demonstrated a latency sensitivity-based cache partitioning (LSP) framework, which leverages a lightweight runtime mechanism to quantify the latency-sensitivity and a navel cost function to guide the LLC partitioning. However, all these prior studies focused on improving the average-case performance and not time predictability. These methods typically rely on runtime profiling information to improve the efficiency of cache partitioning or adapt the partitioning space based on the runtime program behavior, which generally is harmful to time predictability and make WCET analysis even harder.

There exist studies on static and dynamic cache locking to improve time predictability or performance [8, 19-23]. However, to the best of our knowledge, none of these studies accounted for the unique features of heterogeneous multicore processors. Thus, their impacts on GPUs or the integrated CPU-GPUs are still unknown. Also, cache partition and locking mechanisms have never been used together in the context of the heterogeneous CPU-GPU multicore processors, which will be explored in this paper.

Researchers recently have also studied real-time scheduling to exploit GPUs for real-time systems [24-28]. All these efforts, however, assume that the WCET of the GPU kernel is known. Therefore, our work on improving time predictability (and WCET analyzability) is complementary to existing efforts and can benefit these studies.

This paper is based on an extension of a conference paper [29]. In this paper, we consider the benchmark behavior to investigate various impacts of CPU and GPU cache locking on cache miss rates and performance. Moreover, we further study the hardware-based static way partitioning to understand how different partitioning affects the performance of both CPU and GPU.

III. MOTIVATION

While the integrated CPU-GPU architecture can benefit the performance by enabling efficient data sharing on-chip and reducing the overhead of communicating data between CPU and GPU cores, the highly shared resources may have a negative impact on the overall performance in case there are too many interferences on the shared resources. An important shared on-chip resource is the LLC, most of which is likely to be used by GPU core due to the diverse cache access demands, leading to limited LLC space for CPU cores and thus more CPU cache misses and worse CPU performance. On the other hand, GPU applications can generally exploit thread-level parallelism to hide cache miss latency, making them less sensitive to the reduction in the LLC space. Moreover, since the performance of the CPU application is now dependent on the GPU application and vice versa due to the shared LLC, the complexity of WCET analysis for both CPU and GPU will be increased enormously.

To develop a better understanding of the impact of the concurrent running of CPU and GPU applications on the performance of the CPU and the GPU, respectively, we utilize the MacSim simulator [30] to measure their performance in terms of the total number of execution cycles on both CPU and GPU (more details of our experiments can be seen in Section V). Fig. 2 shows the performance degradation of both CPU and GPU appli-

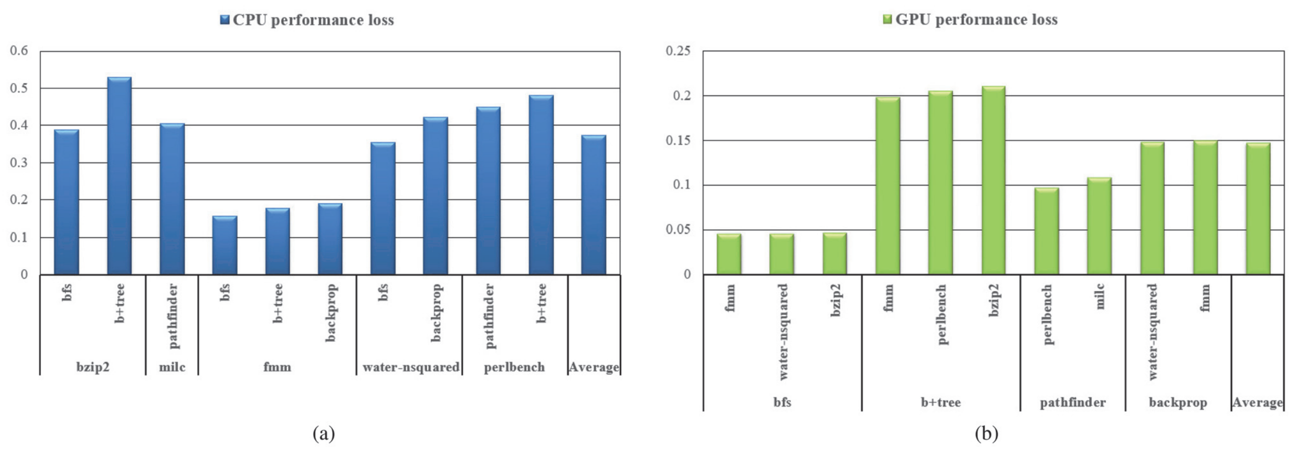


Fig. 2. Performance loss for co-running CPU application (a) and GPU application (b).

cations, respectively, when they are co-running simultaneously, which is normalized to the performance of running the CPU or GPU application alone (i.e., the CPU or the GPU uses the LLC privately without sharing). As we can see in Fig. 2, when running CPU and GPU applications simultaneously, CPU applications result in larger performance degradation. On average, the CPU performance decreases significantly by 37.4% when co-running with GPU applications. However, for GPU, the average performance degradation is only 14.7%. This is because CPU applications are often more sensitive to the cache size, and the variety of data access densities of GPU applications can lead to different amounts of LLC space available for CPU applications. However, it should be noted that the performance impact of LLC sharing on GPU applications is smaller owing to their ability to hide memory latency through high thread-level parallelism (more details can be seen in Section VI).

Although both cache partitioning and locking can improve time predictability, they may also affect the performance of both CPU and GPU applications. In cache partitioning, both CPU and GPU applications will be assigned with a fixed portion of cache space, which may be quite different from the actual cache space they can occupy in a shared LLC. In particular, since GPU applications typically have many more threads and access much more data during a given time interval, cache partitioning may guarantee the CPU applications with a fixed amount of cache space, while GPU applications may actually result in less LLC space to use considering the fixed partition. For cache locking, since the locked cache space cannot be reused, the effective reusable cache space for both CPU and GPU applications may be greatly reduced unless the cache data are unlocked or the locked data are frequently reused to decrease the pressure on the cache space. In summary, both cache partitioning and locking may have an impact on the performance of CPU and GPU applications, though CPU performance is likely to be affected more. Combining both partitioning and locking can result in more complex interactions between the reusable cache space and the reuse of locked data for both CPU and GPU applications, whose performance implication will be examined in this work.

IV. TIME-PREDICTABLE LLC DESIGN OPTIONS

To manage the LLC for better time predictability, we implement and evaluate five time-predictable LLC designs including partitioning, locking CPU blocks, locking both CPU and GPU blocks, and two combinations of partitioning and locking schemes. For cache partitioning, we use hardware-based static way partitioning that allocates a specified number of ways in a single cache set to CPU and GPU applications, respectively. For cache locking, we use a reuse counter-based mechanism that dynamically selects cache blocks to be locked according to the reuse frequency.

Cache partitioning. We use the hardware-based static way partitioning to partition the LLC space for CPU and GPU. The static partitioning chooses a preset percentage (50% by default, but it can be varied). The 50% value means that half of the ways in a cache set are exclusively reserved for CPU applications whereas the remaining ways are for GPU use. The allocation of a cache way follows the logic specified in Algorithm 1. The ownership of cache ways is decided at runtime. Specifically, if an unused cache block is first used to load a CPU block, then only CPU application can access and replace this block afterward. Similarly, a cache block which is first used by a GPU core is then exclusive for the GPU application. Within each partitioning, the least recently used (LRU) cache replacement policy is applied, which will sacrifice the LRU cache block to accommodate the new data.

Algorithm 1 Cache Partitioning: allocate a cache block from the partition.

```
Require: $CPU\_Static\_Partition$+$GPU\_Static\_Partition$ =$num\_cache\_way$
 1: Find_Replacement_From_Same_Block_Type($Set\_num$, $Block\_type$)
 2: if $Block\_type$ is CPU block then
 3:     $Current\_type$=CPU block
 4:     $Current\_count$=$CPU\_block\_num$ of $LLC[Set\_num]$
 5:     $Current\_limit$=$CPU\_Static\_Partition$
 6: else
 7:     $Current\_type$=GPU block
 8:     $Current\_count$=$GPU\_block\_num$ of $LLC[Set\_num]$
 9:     $Current\_limit$=$GPU\_Static\_Partition$
10: end if
11: for $index = 1$ to $num\_cache\_way$ do
12:     if $LLC[Set\_num][index]$ is unused and $Current\_count < Current\_limit$ then
13:         $LRU\_index$=$index$
14:         $BlockType$ of $LLC[Set\_num][index]$=$Current\_type$
15:         if $Current\_type$ is CPU block then
16:             ++$CPU\_block\_num$ of $LLC[Set\_num]$
17:         else
18:             ++$GPU\_block\_num$ of $LLC[Set\_num]$
19:         end if
20:         BREAK
21:     end if
22:     if $LLC[Set\_num][index]$ is used and $Type$ of $LLC[Set\_num][index]$=$Current\_type$ then
23:         if $LLC[Set\_num][index]$ is less recently used then
24:             $LRU\_index$=$index$
25:         end if
26:     end if
27: end for
28: return $LLC[Set\_num][LRU\_index]$
```

Locking CPU and GPU blocks. To support the cache locking mechanism evaluated in this work, a counter is associated with each cache way to record the frequency of accesses to the current way. A cache block is locked once the re-access time is greater than a threshold which is predetermined before the execution of applications. Since the cache block is inevitable after being locked, the LRU policy is tailored to search the LRU cache block only among the unlocked ones. In case that all the cache blocks in a set are locked, the new data bypasses the LLC and heads to the upper-level cache directly. The pseudo-code of cache locking is detailed in Algorithm 2.

Algorithm 2 Cache Locking: find a replacement cache block from the unlocked blocks.

```
 1: Find_Replacement_From_Unlocked_Block($Set\_num$)
 2: $LRU\_index$=$-1$
 3: for $index = 1$ to $num\_cache\_way$ do
 4:     if $LLC[Set\_num][index]$ is unused then
 5:         $LRU\_index$=$index$
 6:         BREAK
 7:     end if
 8:     if $LLC[Set\_num][index]$ is used and $Reuse\_frequency$ of $LLC[Set\_num][index] < Locking\_threshold$ then
 9:         if $LLC[Set\_num][index]$ is less recently used then
10:             $LRU\_index$=$index$
11:         end if
12:     end if
13: end for
14: if $LRU\_index = -1$ then
15:     return NULL {Bypassing LLC, If all blocks in a set are locked}
16: else
17:     return $LLC[Set\_num][LRU\_index]$
18: end if
```

Locking CPU blocks only. Instead of locking both CPU and GPU blocks, in this design, only cache space holding the data of CPU application can be locked according to the reused frequency while cache space owned by GPU application is free to be reused. Locking a cache block can negatively influence the performance of the remaining memory blocks that are mapped to the same set, since the reusable cache capacity is reduced. Therefore, the cache locking algorithm should ensure that the overhead and benefit of locking are balanced. Locking both CPU and GPU gets a high probability of a few unlocked cache space for other memory blocks. As a result, the benefit of locking is limited and the overhead rises. Since CPU applications are more sensitive to cache misses and GPU applications tend to access the LLC more intensively, we choose to lock CPU blocks only to prevent frequently used CPU blocks being replaced and also leave more cache space for CPU data as no GPU blocks are locked in this scheme. This is not expected to significantly impact GPU performance because of its capability to tolerate the cache miss latency through thread-level parallelism. It should be noted that we did not study a GPU only locking scheme because this will essentially leave few, if any, space for CPU applications, which can greatly affect the CPU performance.

Partitioning and locking CPU and GPU blocks. While cache locking guarantees that the locked CPU cache blocks cannot be evicted by GPU blocks, they can still be replaced by GPU blocks before being locked. To address this issue, we propose to combine cache partitioning and locking by partitioning the LLC into two equal parts and locking highly reused cache blocks within each partition. While partitioning guarantees a deterministic cache space for both CPU and GPU, cache locking can avoid early evicting of frequently used CPU and/or GPU data and thus can potentially enhance performance further. On the other hand, compared to pure cache partitioning, partitioning and locking will reduce the cache space that can be reused within both CPU and GPU partitions, which may result in a negative impact on the performance. Therefore, the overall performance of this LLC management method will depend on the interaction of both factors.

Partitioning and locking CPU blocks only. Similar to the locking CPU cache block only, in this method, we partition the LLC for CPU and GPU, and employ cache locking scheme in the CPU partition only. Thus, in the GPU partition, the data are not locked, thereby potentially leaving more flexibility for GPU data.

V. EVALUATION METHODOLOGY

We have implemented all the above-mentioned time-predictable LLC designs and evaluated them using the

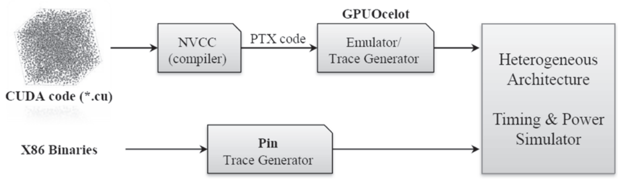


Fig. 3. The simulation framework. Adapted from [31].

MacSim [30]. MacSim is a heterogeneous architectural timing model simulator, which conducts trace-driven cycle-level simulation. MacSim thoroughly models micro-architectural behaviors, including detailed pipeline stages, multi-threading, and memory systems [31], and the traces of GPU applications are generated by GPUOcelot [32]. The overview of the MacSim simulator is shown in Fig. 3.

The heterogeneous processor we studied consists of 4 CPU cores and 6 GPU cores. The CPU core modeled is a highly pipelined 4 wide superscalar out-of-order processor with a 256 entry ROB and a gshare branch predictor, loosely based on Intel's Sandy Bridge [33]. The GPU cores are similar to the cores used in NVIDIA's Fermi [34]. In this architecture, the GPU consists of a scalable number of streaming multiprocessors (SMs), each of which contains several scalar processors (SP) and special function units (SFUs), a multi-threaded instruction fetch and issue unit, a read-only constant cache, and a read/write scratch pad memory called Shared Memory [31]. The important parameters for our heterogeneous architecture are listed in Table 1. By default, the LLC is configured as 256 kB and 16-way set-associative.

Table 2 describes the CPU and GPU benchmarks used in our evaluation. The CPU benchmarks are selected from SPEC CPU 2006 [35] and SPLASH-2 benchmarks, and GPU benchmarks are selected from Rodinia [36]. We generate 10 groups of benchmarks to run simultaneously, each of which consists of one CPU and one GPU application, as shown in Table 3. Because GPU application always terminates earlier than CPU application, we repeat GPU application until CPU application finishes so that both CPU and GPU applications keep running during the entire simulation. Since cache partitioning and locking can eliminate the inter-core cache interferences between CPU and GPU, it makes WCET analysis less complicated and possible. Thus, in this paper, we focus on evaluating the simulated performance in terms of the number of execution cycles for both CPU and GPU. Developing a WCET analyzer for the integrated CPU-GPU processor needs to deal with challenges other than the LLC such as branch divergence on GPUs, the interferences in shared DRAM and on-chip network, etc., which are out of the scope of this paper.

We comparatively evaluate the following six LLC designs:

- No partitioning and locking (i.e., shared LLC);
- Locking CPU and GPU blocks (i.e., Locking Both);
- Locking CPU blocks only (i.e., Locking CPU);

Table 2. CPU and GPU benchmarks

CPU benchmarks	SPEC CPU 2006: bzip2, perlbench, milc SPLASH-2: water-nsquared, fmm
GPU benchmarks	Rodinia: bfs, b+tree, pathfinder, backprop

Table 3. Groups of co-running CPU and GPU benchmarks

Group No.	CPU benchmarks	GPU benchmarks
1	bzip2	bfs
2	bzip2	b+tree
3	milc	pathfinder
4	fmm	bfs
5	fmm	b+tree
6	fmm	backprop
7	water-nsquared	bfs
8	water-nsquared	backprop
9	perlbench	pathfinder
10	perlbench	b+tree

Table 1. Simulated heterogeneous processor configuration

	Description
CPU	32 kB L1I cache, 8-way set-associative, line size 64 byte 16 kB L1D cache, 8-way set-associative, line size 64 byte, 3 cycles latency 256 kB L2 cache, 8-way set-associative, line size 64 byte, 8 cycles latency 4 cores, 3 GHz
GPU	6 cores, 1.5 GHz 4 kB L1I cache, 8-way set-associative, line size 64 byte, 2 cycles latency 32 kB L1D cache, 8-way set-associative, line size 64 byte no L2 cache
LLC	1 GHz, 30 cycles latency, 4 banks, 1 cycle latency 256 kB, 16-way set-associative, line size 64 byte
Memory	dual channels, 1.6 GHz

• LLC partitioning;
• LLC partitioning and locking CPU blocks (i.e., Partitioning&Locking CPU); and
• LLC partitioning and locking CPU and GPU blocks (i.e., Partitioning&Locking).

VI. EXPERIMENTAL RESULTS

We first study the cache sensitivity of different CPU and GPU applications. We vary the size of LLC from 64 kB to 1024 kB with fixed 16-way associativity and measure LLC miss rates as well as the performance for CPU and GPU applications. The evaluation involves 10 combinations of CPU and GPU applications.

A. Cache Sensitivity

As can be seen in Fig. 4, for CPU applications, `bzip2`, `perlbench`, and `fmm` are sensitive to the cache size while `milc` and `water-nsquared` are not cache-sensitive. For GPU applications as depicted in Fig. 5, `b+tree`, `pathfinder`, and `backprop` are cache-sensitive while `bfs` is not.

B. Evaluation of CPU Applications

Fig. 6 depicts the cache miss rates and performance of CPU applications with six different LLC designs, including

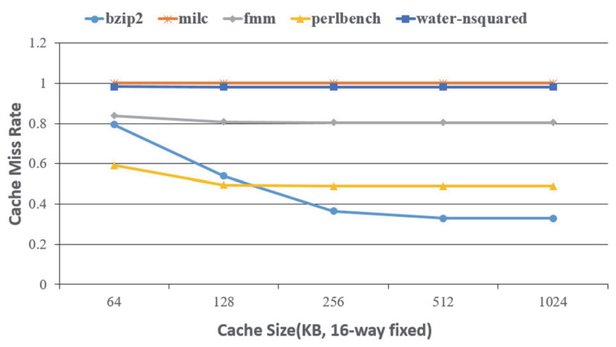


Fig. 4. The cache-sensitivity of different CPU applications.

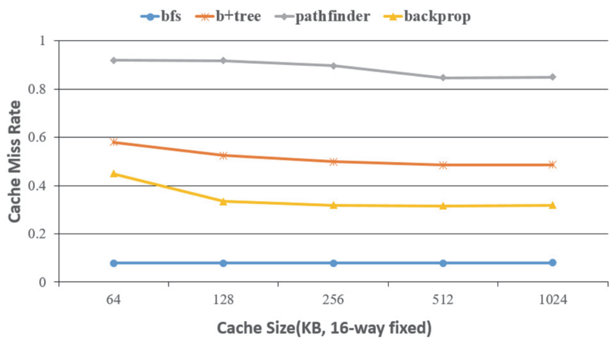
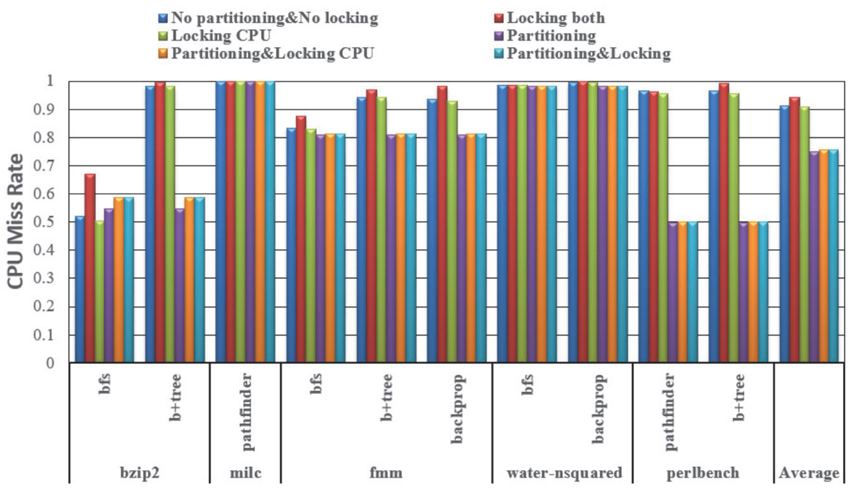


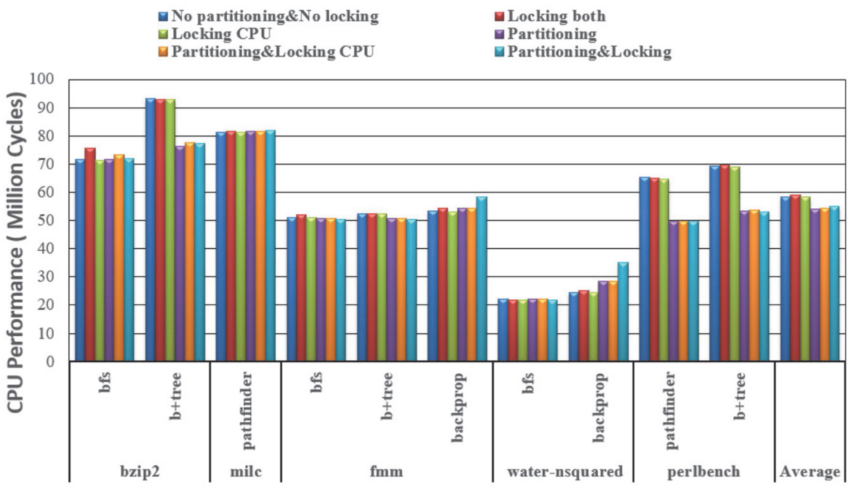
Fig. 5. The cache-sensitivity of different GPU applications.

the default one without any locking or partitioning. For most of the CPU applications, partitioning outperforms the other five LLC schemes. In the shared LLC (i.e., no partitioning&no locking), the availability of LLC space is less than 50% for most of the CPU applications, since GPU applications consume more LLC space than CPU applications due to the higher density of data accesses, leading to higher CPU cache miss rates and worse performance for CPU applications. The partitioning scheme splits the LLC equally for CPU and GPU applications. This has the advantage of performance isolation, which signifies that the performance of a CPU application is not affected by the concurrent GPU application. For example, in the benchmark group `perlbench&b+tree`, CPU application `perlbench` can consume 50% cache space when the partitioning scheme is applied. However, it can only effectively use lower than 10% cache space when using other schemes without partitioning. The portion of the cache space that is actually used by CPU and GPU applications is depicted in Fig. 7. As we can see, while partitioning can guarantee 50% of cache space for CPU applications, all other approaches result in a significantly unequal amount of cache space allocation. In such LLC designs, as expected, GPU applications use more than 90% of cache space, while CPU applications can only use less than 10% of cache space, potentially resulting in higher CPU miss rates and worse performance. Therefore, the cache partitioning scheme is the most effective approach for improving the performance of cache-sensitive CPU applications such as `bzip2`, `fmm`, and `perlbench`. On average, cache partitioning improves CPU performance by 7.76% over the baseline shared LLC (no partitioning&no locking) and reduces CPU LLC miss rate by 16.37%. Cache partitioning is applied in the other two schemes, partitioning with locking CPU blocks and partitioning with locking CPU and GPU blocks, which also improve CPU performance and decrease CPU LLC miss rate. However, these two approaches perform a little worse than cache partitioning. In these two approaches, the LLC space is reduced by both partitioning and locking and the remaining LLC space is not enough for new memory blocks thereby resulting in lower cache miss rate and worse performance of CPU applications.

Compared to the baseline scheme, we find that locking CPU blocks results in slightly better performance. For CPU applications in all the benchmark groups, locking CPU blocks outperforms the method of locking both CPU and GPU blocks. On average, locking CPU blocks improves performance by only 0.35% over the baseline scheme, thereby decreasing the CPU LLC miss rate by 0.48%. Although locking only CPU blocks can slightly improve CPU performance, locking both CPU and GPU blocks increases the LLC miss rate and degrades CPU performance as aggressively locking cache blocks of both CPU and GPU data reduces the reusable cache capacity sharply, resulting in more cache misses for the remaining



(a)



(b)

Fig. 6. Comparison of cache miss rates and performance of CPU applications with six LLC designs (LLC configuration: 256 kB, 16-way set-associative). (a) Cache miss rate. (b) Execution cycles.

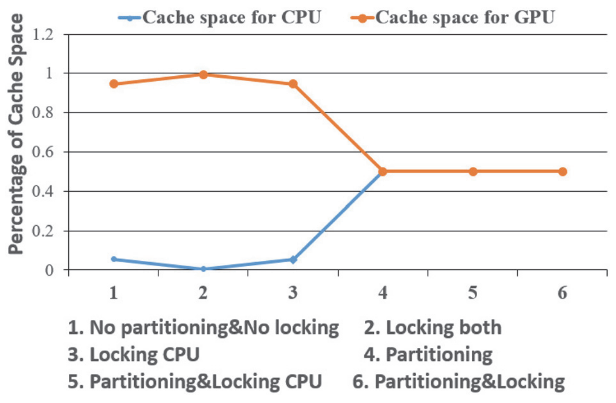


Fig. 7. Percentage of actual LLC space used by CPU and GPU applications at runtime `perlbench&b+tree`.

memory blocks that are not locked. On average, locking both CPU and GPU blocks increases the LLC miss rate by 3.04% and decreases the CPU performance by 1.02% as compared to the baseline shared LLC.

Cache partitioning has also been combined with locking in the other two schemes, i.e., partitioning with locking CPU blocks, and partitioning with locking CPU and GPU blocks, both of which improve CPU performance and decrease CPU LLC miss rate. However, these two approaches perform a little worse than cache partitioning alone. The reason is that when locking is used in addition to partitioning, the locked data reduces the reusable LLC space within CPU and/or GPU LLC partition. This leads to higher CPU cache miss rates and worse performance for CPU applications.

While partitioning performs better than other LLC designs for most of the CPU applications, one exception we observe is with the group of the CPU application bzip2 and the GPU application bfs. Locking CPU actually is the best option for improving the performance of bzip2 when running with bfs. The reason is that bfs only needs a small share of LLC, thereby leaving most of the LLC to bzip2. Therefore, bzip2 can get more LLC space to either lock more data that are frequently used or serve upcoming memory blocks. Since the spare LLC space for CPU application is more than half of the whole LLC when using CPU cache locking, partitioning the LLC into two equal parts reduces the availability of cache space for bzip2. As a result, the locking CPU scheme outperforms the cache partitioning scheme in this case.

However, the benefit of locking CPU blocks can be highly affected by not only the CPU benchmark behavior, but also the co-running GPU benchmark behavior. When replacing bfs with b+tree as the concurrent GPU application, we observe that the LLC miss rate of bzip2 increases significantly and its performance also degrades dramatically. This happens because bzip2 is sensitive to cache space and it is desirable to allocate more LLC space to it. Unfortunately, b+tree captures a much larger portion of the LLC, leaving less LLC space for bzip2. Even with CPU cache locking, most of the CPU data is still evicted by GPU data before they can be reused and locked. Fig. 8 demonstrates that nearly 67% of LLC space belongs to bzip2 when executing concurrently with bfs, which unfortunately is reduced to less than 5% when running with b+tree.

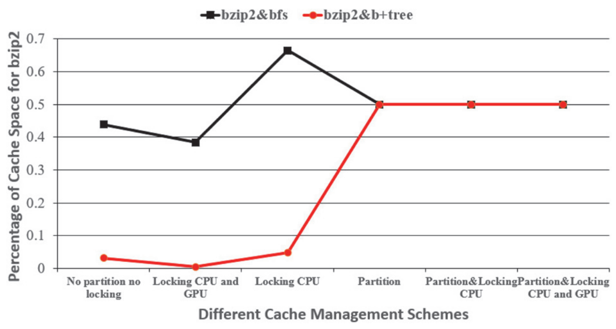


Fig. 8. Comparison of actual LLC space used by bzip2 in bzip2&bfs and bzip2&b+tree.

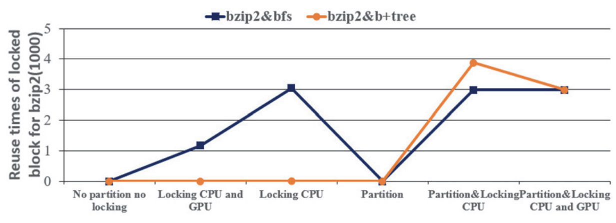


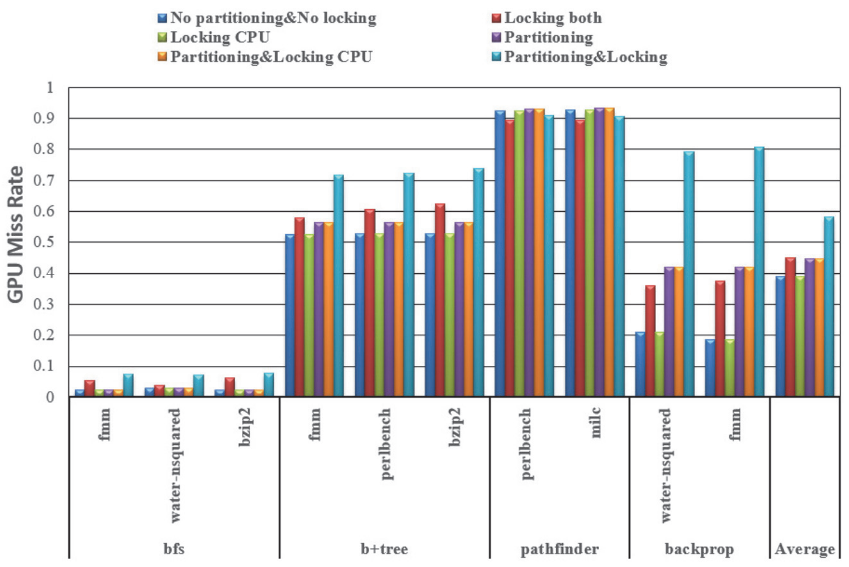
Fig. 9. Comparison of the reuse times of locked blocks for bzip2 in bzip2&bfs and bzip2&b+tree.

Fig. 9 shows the number of reuses for the locked data when bzip2 is co-running with bzip2&bfs or bzip2&b+tree. We observe that the locked cache blocks in bzip2 are reused over 3,000 times when bzip2 is running with bfs, while this number becomes only 7 when bzip2 is running with b+tree. These results explain why bzip2 achieves better performance with bfs but becomes worse with b+tree with CPU locking.

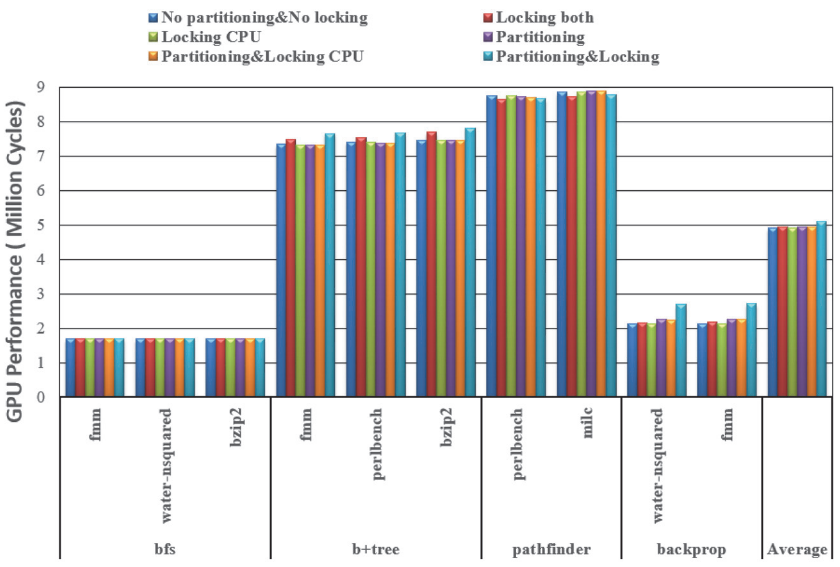
In Fig. 6, we also notice that for the benchmark groups fmm&backprop and water-nsquared&backprop, the partition and locking scheme has lower cache miss rates but worst performance. Although the LLC is partitioned into two parts for CPU and GPU application, the performance of CPU applications can still be negatively impacted by GPU applications due to the shared DRAM and NoC, especially if too many GPU requests flood those shared resources. For both fmm&backprop and water-nsquared&backprop, the cache miss rates of GPU benchmark backprop are very high, leading to a significant number of requests sent to the DRAM and NoC. As a result, the CPU miss latency is increased due to the prolonged waiting time, resulting in worse CPU performance. In our future work, we plan to address the time predictability issue for the shared DRAM and NoC in an integrated manner, which is out of the scope of this paper though.

C. Evaluation of GPU Applications

Fig. 10 provides the cache miss rates and performance of GPU applications for the six LLC designs. As we can see, neither the cache partitioning nor the locking scheme performs better than the baseline design for the majority of GPU applications. This is because in the default shared LLC, GPU applications can always have much more than half of the LLC due to their overwhelming data accesses, and cache partitioning actually reduces the available LLC space for the GPU applications. When locking GPU blocks in the LLC, the remaining reusable cache space becomes very limited because the GPU blocks tend to be easily locked owning to their high data access density. In fact, both partitioning and locking result in smaller cache space for GPU applications compared to the baseline shared LLC. On average, the GPU LLC miss rates in partitioning and locking are increased by 5.65% and 5.89%, respectively, over the baseline. Combining partitioning and locking aggravates this problem to a greater extent, thereby raising the GPU cache miss rate by 19.14% on average. Nevertheless, the increase of miss rates only has very limited impact on the performance of GPU applications due to GPU's massive threading capability to tolerate the cache miss latencies. On average, cache partitioning reduces performance by only 0.43% compared to the baseline, while partitioning and locking reduces performance by 4.00%. The average GPU performance degradation of locking CPU and GPU blocks



(a)



(b)

Fig. 10. Comparison of cache miss rates and performance of GPU application with six different LLC designs (LLC configuration: 256 kB, 16-way set-associative). (a) Cache miss rate. (b) Execution cycles.

(without partitioning) is 0.75%. Locking CPU only achieves performance very close to that of the baseline, because much less CPU data can be successfully reused and locked by GPU applications due to the higher cache access rates.

Although locking CPU and GPU blocks degrades the performance for most of the GPU applications, it does reduce the GPU miss rate and improve the performance for the GPU benchmark `pathfinder` in the groups of `pathfinder&milc` and `pathfinder&perlbench`. The reason is that more frequently reused cache blocks are properly locked for `pathfinder`, and after locking, the remaining LLC space for GPU application is still sufficient to serve the upcoming GPU data cache accesses. For example, Table 4 lists the cache usage of `pathfinder` when co-running with `perlbench`. We find that 95.6% LLC space is used by `pathfinder` and 10.4% GPU blocks are locked, leading to 7,289 reuses of the locked GPU blocks and better performance. On the contrary, Table 5 lists the cache usage of `b+tree` when co-running with `perlbench`. Typically, 99.4% of the LLC space is used by `b+tree` and 88.6% GPU blocks are locked, leading to 108,113 reused locked GPU blocks. Although almost the entire LLC can be used by `b+tree`, most of the GPU blocks are locked and according to the cache sensitivity of `b+tree` reported in Fig. 5, the cache

Table 4. Locking cache blocks of GPU application `pathfinder` in the benchmark group `perlbench&pathfinder`

LLC management	Percentage of LLC for pathfinder	Locking rate of GPU blocks (%)	Reuse times of locked GPU blocks
No partitioning&no locking	95.9	0	0
Locking Both	95.6	10.4	7,289
Locking CPU	96.8	0	0
Partitioning	50.0	0	0
Partitioning&Locking CPU	50.0	0	0
Partitioning&Locking	50.0	8.0	3,009

Table 5. Locking cache blocks of GPU application `b+tree` in the benchmark group `perlbench&b+tree`

LLC management	Percentage of LLC for b+tree	Locking rate of GPU blocks (%)	Reuse times of locked GPU blocks
No partitioning&no locking	94.5	0	0
Locking Both	99.4	88.6	108,113
Locking CPU	94.6	0	0
Partitioning	50.0	0	0
Partitioning&Locking CPU	50.0	0	0
Partitioning&Locking	50.0	93.2	65,461

miss rate of `b+tree` increases remarkably on the remaining reusable cache space.

VII. CONCLUSION

While achieving high throughput is important for GPGPU computing, it is crucial to ensure time predictability for real-time GPU computing. In this paper, we have explored five different time-predictable cache management schemes for the LLC in the integrated CPU-GPU architecture. In particular, we have studied cache partitioning, locking on both CPU and GPU, locking CPU only, and their combinations with partitioning. Our experimental results indicate that cache locking, either locking the CPU alone or both CPU and GPU, does not benefit CPU or GPU performance for most of the benchmarks we have studied. By comparison, cache partitioning is more effective in boosting the CPU's performance while improving time predictability at the same time. The GPU performance for most of the applications is not very sensitive to these time-predictable LLC designs, though locking both CPU and GPU data may degrade the GPU performance noticeably. On average, cache partitioning can improve the performance of CPU applications by 7.76% with only 0.43% performance degradation for GPU applications. Moreover, we study the hardware-based static way partitioning to understand how different partitioning affects the performance of both CPU and GPU. We find that the partitioning percentage that can achieve the most effective overall performance must be appropriately chosen according to the cache-sensitivity of both GPU and CPU applications.

In our future work, we also plan to exploit the data/instruction access patterns of both CPU and GPU applications to make cache partition and locking more effective for real-time integrated CPU-GPU architectures. Exploring time-predictable LLC architecture with high performance is our first step towards building a fully time-predictable and high-performance heterogeneous CPU-GPU architecture for hard real-time data-parallel applications. Our future work will investigate the time-predictable design of other shared architectural components such as DRAM and NoC, as well as their interactions with the time-predictable LLCs, based on which a full WCET analyzer will be developed.

ACKNOWLEDGMENTS

This work was funded in part by the NSF grant CNS-1421577.

REFERENCES

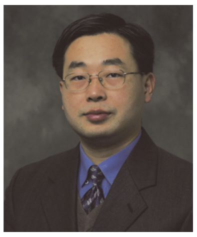
1. G. A. Elliott, B. C. Ward, and J. H. Anderson, "GPUSync: a framework for real-time GPU management," in *Proceedings of 2013 IEEE 34th Real-Time Systems Symposium*, Vancouver, Canada, 2013, pp. 33-44.
2. L. Thiele and R. Wilhelm, "Design for timing predictability,"

Real-Time Systems, vol. 28, no. 2-3, pp. 157-177, 2004.
3. C. Berg, J. Engblom, and R. Wilhelm, "Requirements for and design of a processor with predictable timing," in *Perspectives Workshop: Design of Systems with Predictable Behaviour*. Dagstuhl, Germany: Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), 2004.
4. V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," in *Proceedings of the 45th Annual Design Automation Conference*, Anaheim, CA, 2008, pp. 300-303.
5. M. Paolieri, E. Quinones, F. J. Cazorla, G. Bernat, and M. Valero, "Hardware support for WCET analysis of hard real-time multicore systems," *ACM SIGARCH Computer Architecture News*, vol. 37, no. 3, pp. 57-68, 2009.
6. C. A. Healy, R. D. Arnold, F. Mueller, D. B. Whalley, and M. G. Harmon, "Bounding pipeline and instruction cache performance," *IEEE Transactions on Computers*, vol. 48, no. 1, pp. 53-70, 1999.
7. Y. T. Li, S. Malik, and A. Wolfe, "Cache modeling for real-time software: beyond direct mapped instruction caches," in *Proceedings of the 17th IEEE Real-Time Systems Symposium*, Washington, DC, 1996, pp. 254-263.
8. Y. Liang and T. Mitra, "Instruction cache locking using temporal reuse profile," in *Proceedings of the 47th Design Automation Conference*, Anaheim, CA, 2010, pp. 344-349.
9. G. E. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *The Journal of Supercomputing*, vol. 28, no. 1, pp. 7-26, 2004.
10. S. Kim, D. Chandra, and Y. Solihin, "Fair cache sharing and partitioning in a chip multiprocessor architecture," in *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Juan-les-Pins, France, 2004, pp. 111-122.
11. M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: a low-overhead, high-performance, runtime mechanism to partition shared caches," in *Proceedings of 2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Orlando, FL, 2006, pp. 423-432.
12. M. Moreto, F. J. Cazorla, A. Ramirez, and M. Valero, "MLP-aware dynamic cache partitioning," in *High-Performance Embedded Architectures and Compilers*. Heidelberg: Springer, 2008, pp. 337-352.
13. G. Kedar, A. Mendelson, and I. Cidon, "SPACE: semi-partitioned cache for energy efficient, hard real-time systems," *IEEE Transactions on Computers*, vol. 66, no. 4, pp. 717-730, 2017.
14. J. Lee and H. Kim, "TAP: a TLP-aware cache management policy for a CPU-GPU heterogeneous architecture," in *Proceedings of IEEE International Symposium on High-Performance Comp Architecture*, New Orleans, LA, 2012, pp. 1-12.
15. 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*, Edinburgh, UK, 2013, pp. 225-234.
16. D. H. Woo and H. H. S. Lee, "COMPASS: a programmable data prefetcher using idle GPU shaders," *ACM SIGPLAN Notices*, vol. 45, no. 3, pp. 297-310, 2010.
17. Y. Yang, P. Xiang, M. Mantor, and H. Zhou, "CPU-assisted GPGPU on fused CPU-GPU architectures," in *Proceedings of IEEE International Symposium on High-Performance Comp Architecture*, New Orleans, LA, 2012, pp. 1-12.
18. P. H. Wang, C. H. Li, and C. L. Yang, "Latency sensitivity-based cache partitioning for heterogeneous multi-core architecture," in *Proceedings of the 53rd Annual Design Automation Conference*, Austin, TX, 2016, pp. 1-6.
19. K. Qiu, M. Zhao, C. J. Xue, and A. Orailoglu, "Branch prediction-directed dynamic instruction cache locking for embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 13, no. 5s, article no. 156, 2014.
20. T. Adegbija and A. Gordon-Ross, "Phase-based cache locking for embedded systems," in *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI*, Pittsburg, PA, 2015, pp. 115-120.
21. K. Anand and R. Barua, "Instruction-cache locking for improving embedded systems performance," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 3, article no. 53, 2015.
22. B. Cilku, D. Prokesch, and P. Puschner, "A time-predictable instruction-cache architecture that uses prefetching and cache locking," in *Proceedings of 2015 IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops*, Auckland, New Zealand, 2015, pp. 74-79.
23. W. Zheng, H. Wu, and C. Nie, "Integrating task scheduling and cache locking for multicore real-time embedded systems," *ACM SIGPLAN Notices*, vol. 52, no. 5, pp. 71-80, 2017.
24. G. Raravi, B. Andersson, and K. Bletsas, "Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors," *Real-Time Systems*, vol. 49, no. 1, pp. 29-72, 2013.
25. S. Kato, K. Lakshmanan, R. Rajkumar, and Y. Ishikawa, "TimeGraph: GPU scheduling for real-time multi-tasking environments," in *Proceedings of USENIX Annual Technical Conference*, Portland, OR, 2011, pp. 17-30.
26. G. A. Elliott and J. H. Anderson, "Globally scheduled real-time multiprocessor systems with GPUs," *Real-Time Systems*, vol. 48, no. 1, pp. 34-74, 2012.
27. G. A. Elliott and J. H. Anderson, "Robust real-time multiprocessor interrupt handling motivated by GPUs," in *Proceedings of 2012 24th Euromicro Conference on Real-Time Systems*, Pisa, Italy, 2012, pp. 267-276.
28. Y. Xu, R. Wang, T. Li, M. Song, L. Gao, Z. Luan, and D. Qian, "Scheduling tasks with mixed timing constraints in GPU-powered real-time systems," in *Proceedings of the 2016 International Conference on Supercomputing*, Istanbul, Turkey, 2016, pp. 1-13.
29. X. Wang and W. Zhang, "Cache locking vs. partitioning for real-time computing on integrated CPU-GPU processors," in *Proceedings of 2016 IEEE 35th International Performance Computing and Communications Conference (IPCCC)*, Las Vegas, NV, 2016, pp. 1-8.
30. macsim: a simulator for heterogeneous architecture, https://code.google.com/archive/p/macsim/.
31. H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "MacSim: a CPU-GPU heterogeneous simulation

framework user guide," Georgia Institute of Technology, Atlanta, GA, 2012.
32. G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proceedings of 2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Vienna, Austria, 2010, pp. 353-364.
33. D. Kanter, "Intel's Sandy Bridge microarchitecture," 2010; https://www.realworldtech.com/sandy-bridge/2/.
34. NVIDIA "NVIDIA's next generation CUDA compute architecture: Fermi," 2009; https://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.
35. C. D. Spradling, "SPEC CPU2006 benchmark tools," *ACM SIGARCH Computer Architecture News*, vol. 35, no. 1, pp. 130-134, 2007.
36. 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 *Proceedings of 2009 IEEE International Symposium on Workload Characterization (IISWC)*, Austin, TX, 2009, pp. 44-54.

Xin Wang

Xin Wang received his B.S. degree in electronic engineering from Peking University, China, in 2008. He is currently pursuing for his Ph.D. degree in computer engineering at Virginia Commonwealth University, USA. His research interests include GPU and heterogeneous CPU-GPU architectures.



Wei Zhang https://orcid.org/0000-0003-1343-2817

Dr. Wei Zhang is a professor and Chair of the Department of Computer Engineering and Computer Science at the University of Louisville. He received his Ph.D. in Computer Science and Engineering from the Pennsylvania State University in 2003. Dr. Zhang served as an assistant/associate professor in Electrical and Computer Engineering at Southern Illinois University Carbondale (SIUC) from 2003 to 2010 and as an associate and full professor at Virginia Commonwealth University from 2010 to 2019. His research interests are in computer architecture, compiler, real-time computing, and hardware security. Dr. Zhang has led 8 NSF projects as the PI and has published 160+ papers in refereed journals and conference proceedings. He received the 2016 Engineer of the Year Award from the Richmond Joint Engineer Council, the 2009 SIUC Excellence through Commitment Outstanding Scholar Award for the College of Engineering, and the 2007 IBM Real-time Innovation Award.