JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 14, No. 4, December 2020, pp. 131-145

Reducing CPU-GPU Interferences to Improve CPU Performance in Heterogeneous Architectures

Hao Wen and Wei Zhang*
Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, VA, USA
wenh2@vcu.edu, wzhang4@vcu.edu

Abstract
Current heterogeneous CPU-GPU architectures integrate general-purpose CPUs and highly thread-level parallelized GPUs (graphic processing units) in the same die. The contention in shared resources between CPU and GPU, such as the last level cache (LLC), interconnection network, and DRAM, may degrade both CPU and GPU performance. Our experimental results show that GPU applications tend to have much more power than CPU applications to compete for the shared resources in the LLC and on-chip network, and therefore make CPU suffer from more performance loss. To reduce the GPU's negative impact on CPU performance, we propose a simple yet effective method based on probability to control the LLC replacement policy for reducing the CPU's inter-core conflict misses caused by GPU without significantly impacting GPU performance. In addition, we develop two strategies to combine the probability-based method for the LLC and an existing technique called virtual channel partition (VCP) for the interconnection network to further improve the CPU performance. The first strategy statically uses an empirically pre-determined probability value associated with VCP, which can improve the CPU performance by 26% on average, but degrades GPU performance by 5%. The second strategy uses a sampling method to monitor the network congestion and dynamically adjust the probability value used, which can improve the CPU performance by 24%, and only have 1% or 2% performance overhead on GPU applications.

Category: Compilers / Programming Languages

Keywords: Heterogeneous architectures; Last level cache; Inter-application interferences

I. INTRODUCTION

Following their popularity in graphics processing and high-performance computing (HPC), graphics processing units (GPUs) have been increasingly used for general-purpose computing on GPUs (GPGPUs). Modern GPUs consist of thousands of simple cores (shader cores) with extremely high-bandwidth external memory systems, which offer tremendous advantages in terms of high throughput and energy efficiency. Each core of the GPU is based on a single instruction multiple data (SIMD) architecture, which can efficiently support the parallel processing of multiple data.

Traditionally, a GPU works with a CPU in a host-device mode, where the CPU executes the host code and launches the GPU kernels to execute the parallel parts of the computation. OpenCL [1] and CUDA [2] are programming models that work on such a host-device mode architecture. However, this architecture has some limitations because it has high communication overheads due to the separate

Open Access http://dx.doi.org/10.5626/JCSE.2020.14.4.131 http://jcse.kiise.org

Received 12 July 2020; **Revised** 26 July 2020; **Accepted** 04 September 2020
*Corresponding Author

 pISSN: 1976-4677 eISSN: 2093-8020

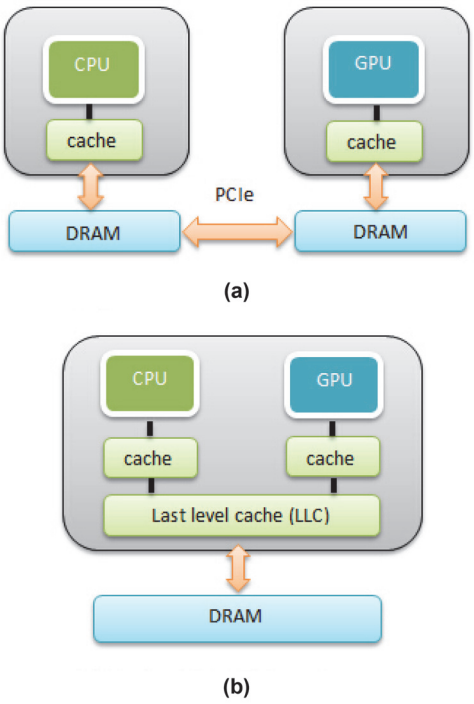


Fig. 1. CPU-GPU heterogeneous architectures: (a) a discrete architecture and (b) an integrated architecture.

address space that requires explicit data movement between the CPU and the GPU [3]. Fig. 1(a) shows a discrete CPU-GPU system with separate DRAM. Before CUDA 6, the memory spaces are distinct both logically (separate memory address spaces) and physically (both the CPU and the GPU have their private DRAM). With CUDA 6, NVIDIA introduced unified virtual memory (UVM), in which the system automatically migrates data between the host and the device so that it looks like they share the memory logically. In contrast, the CPU and the GPU can also be placed on the same die and share the same DRAM, and/or the last level cache (LLC), as well as the interconnection network. As shown in Fig. 1(b), the integrated CPU-GPU architecture (also called unified or fused architecture) can avoid the data transfer between the distinct DRAM addresses. Some of the recent real-life examples of integrated CPU-GPU architectures include Intel's Sandy Bridge and Ivy Bridge, AMD's accelerated processing units (APU), and NVIDIA's Denver [4].

In the integrated CPU-GPU heterogeneous architecture, CPU and GPU applications compete for shared resources in LLC, interconnection network (or Network-on-Chip [NoC]), and DRAM. In the shared LLC, conflict misses between CPU and GPU applications (also called inter-core conflict misses) may slow down both CPU and GPU applications, with a particularly large impact on latency-sensitive CPU applications. In the interconnection network, virtual channels in the router are shared between CPU and GPU, which may block each other when the CPU and GPU packets flow in the virtual channels between different ports of the router. Similarly, the memory channels in the DRAM controller are shared, making the CPU and GPU memory requests interleaved in the request buffer. The DRAM scheduling policy may unfavorably prioritize GPU requests over CPU requests, as indicated in the prior study [5]. These shared resources contentions across LLC, NoC, and DRAM may degrade both CPU and GPU performance of the integrated CPU-GPU architecture.

To assess how CPU and GPU applications interfere with each other and influence the performance in the integrated architecture, we run different combinations of CPU and GPU applications concurrently to get the performance results and compare them to the base performance when they execute alone respectively. We break down the performance results into network latency, DRAM latency, and others, which are shown in Fig. 2 (see Section VII for the evaluation methodology). We observe that the GPU performance degradation is small, while the CPU suffers significant performance loss. This observation is consistent with the results of previous work [3]. One reason is that CPU applications are more latency-sensitive, while GPU

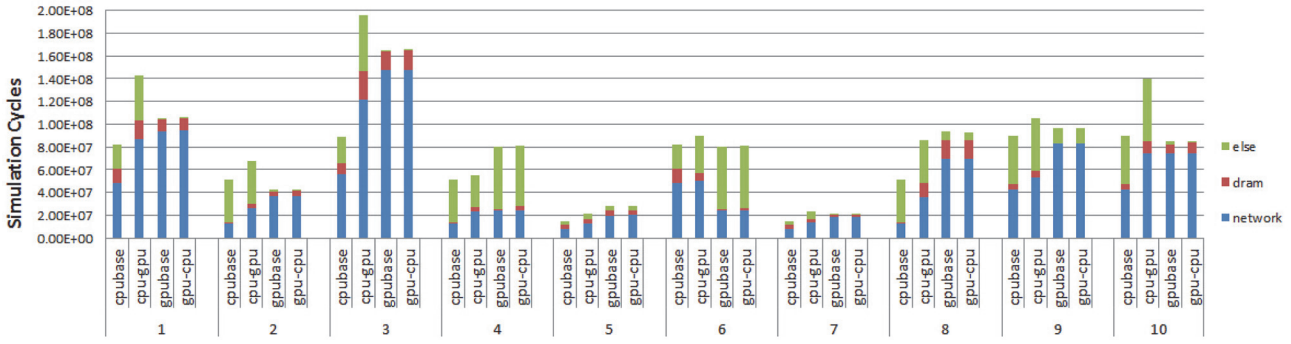


Fig. 2. CPU and GPU application performance breakdown (X-axis; workloads). CPUbase and GPUbase are the performance when they run alone. CPU-GPU is the CPU performance running with GPU and GPU-CPU is the GPU performance running with CPU. The corresponding benchmarks can be seen in Table 1.

Table 1. LLC miss rate (%) increases when CPU and GPU applications run concurrently

No.	Benchmark (CPU : GPU)	CPU	CPU-GPU	GPU	GPU-CPU
1	bzip2 : b+tree	43.16	71.55	32.99	34.37
2	fmm : bfs	30.14	56.64	22.20	23.40
3	milc : b+tree	44.16	77.32	32.87	33.70
4	fmm : lud	30.14	47.92	4.46	7.29
5	water-nsquared : pathfinder	94.90	99.54	93.41	93.25
6	bzip2 : lud	43.16	47.65	4.45	4.54
7	water-nsquared : bfs	94.90	99.37	22.43	23.34
8	fmm : pathfinder	30.14	88.85	95.00	94.88
9	perlbench : gaussian	21.88	28.78	0.22	0.29

applications can well-tolerate the memory latency by exploiting massive thread-level parallelism (TLP). Another important reason is GPU applications have much more memory requests and tend to monopolize the resources in the network and LLC. From Fig. 2 we can see that in general, the CPU network latency increases dramatically, which takes a large portion of the overall performance. In the LLC, GPU applications are also more likely to evict out the data of CPU applications. Table 1 illustrates that the CPU LLC miss rates are affected much more than the GPU LLC miss rates when CPU and GPU applications run concurrently, as compared to the baseline miss rates when they run alone.

The above results and analysis show the importance of managing the shared LLC and network to reduce the inter-application interferences on CPU applications from GPU in the integrated architecture. Some researchers have proposed different methods to solve this inter-application interference problem. The reported studies include GPU core management such as controlling GPU concurrency to reduce shared resource contention [3] or shared resource management such as managing shared LLC to allocate more cache ways to the applications that are more sensitive to the cache [6, 7], and partitioning the virtual channel for CPU and GPU to reduce interferences in the interconnection network [4]. For the management of the shared resources, previous efforts focus on either the LLC or the network only. To the best of our knowledge, none of them has considered addressing the contention in both LLC and interconnection network together. Our study indicates focusing only on the LLC or the network management alone may be suboptimal for the CPU performance because the performance degradation may be caused by the inter-application interferences from the LLC, the network, or both.

In this paper, we explore a simple yet effective method to reduce CPU-GPU interferences in the LLC, and combine the LLC and interconnection network management in an attempt to achieve the best CPU performance (as compared to the baseline performance as CPU applications run alone) with insignificant impact on GPU performance (e.g., only 1% or 2% or less). The main contributions are summarized as follows:

- We propose a new simple LLC replacement policy for the CPU-GPU heterogeneous architecture based on probability. It effectively reduces the CPU conflict misses caused by GPU in the LLC by limiting the GPU's evicting power on CPU cache lines.
- We apply an existing technique of virtual channel partition (VCP) [4, 8] for the on-chip network, and study its interaction with the LLC management technique. Our results show that reserving one virtual channel for the CPU is sufficient to benefit CPU performance with insignificant GPU performance overhead.
- We propose to statically combine the probability method on the LLC (with probability 0.001) with the VCP, which can lead to the best CPU performance (26% improvement). However, the GPU performance is degraded by 5% on average.
- To reduce the GPU performance overhead, we can statically choose a higher probability of 0.01 (the higher the probability, the less GPU miss rate in LLC, thus less bandwidth pressure on the network). However, the optimal probability (either 0.001 or 0.01, determined empirically) is different for different applications. We thus propose a sampling method to monitor network congestion and dynamically determine the best application-specific probability value. The dynamic method used in conjunction with VCP can improve CPU performance by 24%, with only 1% or 2% GPU performance overhead.
- We change the DRAM scheduling policy to prioritize CPU requests in the DRAM controller. The results show that it has a limited improvement on CPU performance. This finding is consistent with the results shown in Fig. 2, which indicates that the DRAM latency is not a large portion of the overall performance.

The rest of the paper is organized as follows. Section II gives the background of the integrated CPU-GPU heterogeneous architecture and its programming model. Section III introduces the probability-based inter-core LLC replacement control. Section IV describes the VCP, and Section V presents two approaches to statically or dynamically combine probability-based LLC control and VCP. Section VI discusses the CPU request prioritization in DRAM. The evaluation methodology is described in Section VII and the experimental results are given in Section VIII. We discuss the related work in Section IX and finally, we make conclusions in Section X.

II. BACKGROUPD

We first briefly describe the architecture evaluated in this work, which is shown in Fig. 3. It has 4 CPU cores and 6 GPU cores. Both CPUs and GPUs have their private L1 caches, and each core of CPUs also has a private L2 cache (not shown in the figure). CPUs and GPUs share the LLC (divided into 4 slices, each has 8 banks) and DRAM (with 2 channels), which are connected through a ring network. This architecture is similar to Intel's Sandy Bridge [9], in which each slice of LLC, the CPU cores, the on-die GPU, and the system agent (fancy word for North Bridge) have a stop on the ring bus. The LLC is shared among all the CPU cores and graphics. Each LLC slice is associated with each CPU core although each CPU core can address the entire LLC.

Since all the cores, LLC banks, and memory channels are connected by the simple ring topology network, all CPU memory requests missed in the CPU L2 cache and GPU memory requests missed in the GPU L1 cache are routed to the corresponding LLC banks via the ring network. All memory requests missed in the LLC also need to go through the network to the DRAM. It is not hard to imagine that the network can have high congestion, which is confirmed by the experimental results demonstrated in Fig. 2.

The memory requests coming from the different nodes in the ring are injected into the local router first, which then go through all the intermediate nodes to arrive at the destination nodes. From a router's point of view, an outgoing memory request can be either routed to the left or to the right in the T-shaped crossroads to reach the destination. The routing algorithm will typically choose a shorter path in terms of the number of intermediate nodes along the path. As we can see in the ring topology, the placement of the connected components can influence the performance, which is discussed in the previous work [8]. In this paper, we do not consider the impact of different placements. We evaluate our work based on the placement shown in Fig. 3.

When a GPU kernel is launched, the runtime creates massive concurrent GPU threads that are organized hierarchically. Warps are the minimum scheduling unit inside a shader core. Execution of warps can be performed in any order. All the threads in a warp execute the same instructions but different threads in the warp operate on a different portion of data. If an executing warp is waiting for memory, the warp scheduler will schedule another warp that is ready for execution. This helps to hide the memory latency and maximize the available parallelism to boost the performance.

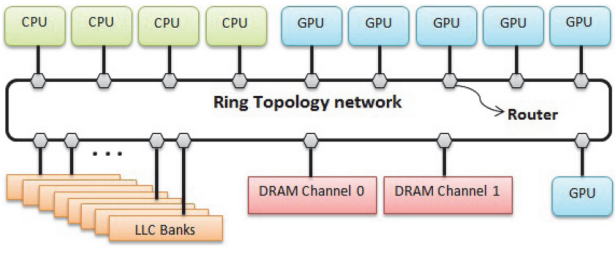


Fig. 3. The integrated CPU-GPU heterogeneous architecture that is evaluated in this work.

III. PROBABILITY BASED METHOD FOR LLC REPLACEMENT

When the LLC is shared between CPU and GPU, because of the limited space in the LLC, both CPU performance and GPU performance may suffer due to inter-core conflict misses. Particularly, GPU applications tend to have more power to evict CPU cache lines, which is demonstrated in Table 1. The CPU and GPU columns give the LLC miss rates when CPU and GPU applications run alone. The CPU-GPU column lists the miss rates of CPU applications running concurrently with GPU applications, whereas the GPU-CPU column shows the miss rates of GPU applications running concurrently with CPU applications. Although both CPU and GPU miss rates increase, the CPU applications suffer much more LLC misses than GPU applications. Therefore, it is important to regulate the evictions of CPU data caused by GPU data, because CPU performance is usually much more sensitive to data access latency.

One way to reduce CPU-evicted-by-GPU conflict misses is to partition the cache ways in every set for different applications. This is what has been done in the previous works [7, 10]. However, determining optimal partitioning for different applications is not trivial. In the context of the integrated CPU-GPU architecture, it is worthy to explore simpler yet effective methods to reduce the CPU conflict misses caused by GPU applications.

In this paper, we propose to use a probability-based method to limit the evicting power of GPU on CPU cache lines. The key idea is to mark every cache line whether it contains CPU data or GPU data. Whenever a cache line that stores CPU data is about to be evicted by a GPU miss, it is only allowed to happen with a pre-set probability. Making an analogy is similar to a coin-flipping. For example, if it is a head, the replacement happens. If it is a tail, instead of replacing the CPU line that is found by the LRU policy, the least recently used GPU line will be replaced instead. In case there is no GPU line in the set, the LRU line is still replaced as usual.

The LRU cache replacement algorithm needs to be modified to enforce the probability-based inter-core cache replacement. The detailed procedure can be seen in Algorithm 1. In this method, the probability can be adjusted to regulate the GPU's evicting power in the LLC. The higher the probability is, the more likely a CPU cache line can be evicted by a GPU miss. If the probability is 1, the LLC becomes a normal LLC that is shared by CPU

and GPU without any restriction on inter-core cache replacements. On the other hand, the smaller the probability is, the less likely CPU data will be evicted by GPU data. Thus, we can use a small probability value to reduce the CPU miss rates; however, it may increase the GPU miss rates. Since GPU applications are often not very sensitive to the miss rates due to GPU's capability to tolerate the miss latency [6, 7], it is possible to aggressively lower CPU miss rates without significantly impacting GPU performance. However, if the GPU LLC miss rate is increased dramatically, it can increase pressure on the shared interconnection network, which can influence performance for both CPU and GPU (see Section IV for more discussion).

Algorithm 1. Probability method for LLC replacement

```
1: begin
2: Pre-set a probability_level (for example, 100);
3: Mark every cache line in a set either a CPU line or a
   GPU line when it is used;
4: One cache line has to be replaced caused by a miss;
5: Call the find_replacement_line_with_probability function
   to determine which line will be evicted out;
6: Function find_replacement_line_with_probability()
7: while (for each cache line in a cache set) do
8:     if (This line is an invalid line) then
9:         Replace this line.
10:    else
11: Check the lru_time to keep track of the least recently
    used line (eviction_line) and least recently used GPU
    line (eviction_line_candidate) if it exists in a set;
12:    end if
13: end while
14: if (The eviction_line is a CPU line and it is caused by
        a GPU access) then
15:    Randomly generate a number between 1 to 1000;
16:    if (The number > probability_level) then
17:        Evict the eviction_line_candidate;
18:        return;
19:    end if
20: end if
21: Evict the eviction_line;
22: EndFunction
23: end
```

The number of GPU accesses in the LLC is generally more than 10 times higher than the CPU accesses. The number of times that a CPU cache line is evicted by GPU is expected to be about one magnitude higher than the CPU misses. To reduce the number of CPU line evictions caused by GPU, one can set a very small probability to begin with for the probability-based LLC control method. The number of such evictions is expected to be reduced by 1,000 times. In other words, the probability 0.001 is small enough to prevent most of the CPU conflict misses caused by GPU.

The probability-based method has a small hardware overhead. An extra bit for every cache line is needed to identify whether it is the cache line that stores CPU data or GPU data. In addition, a global random number generator is needed to produce the probability of inter-core cache replacement control. To hide the latency of random number generation, the random numbers can be generated and stored in advance. The size of the random number store buffer depends on the bandwidth of the cache.

IV. VIRTUAL CHANNEL PARTITIONING

As Fig. 3 shows, each on-chip component is connected to the ring network by the router. The microarchitecture of a router is depicted in Fig. 4. A router has multiple ports. For the router in the ring topology, each router has three ports. The local ports connect to the on-chip components such as CPU and GPU cores, LLC banks, and DRAM controller. The left and right ports link to the other ports of neighboring routers. Each port has multiple incoming and outgoing virtual channels as can be seen in Fig. 4. Packets received from the on-chip components are injected into the injection buffer first (not shown in the figure). Then, packets follow multiple stages [4] to get to the next router.

Since the injection buffer and virtual channels are shared between CPU and GPU applications in heterogeneous architecture, the contention on these resources will slow down each other's progress in the network. However, the flow of CPU packets can be more severely interrupted by GPU packets since GPU applications typically have much more network traffic.

The CPU packets are blocked mainly in two different locations. The first one is at the injection buffer of the local port due to the buffer full, which is above the local port depicted in Fig. 4. Another one is within the virtual channels. Those channels may be exhausted by the GPU packets so that the CPU packets cannot be injected, or the CPU packets are interleaved with GPU packets. In both cases, the CPU packets have to wait for the GPU packets to be drained first.

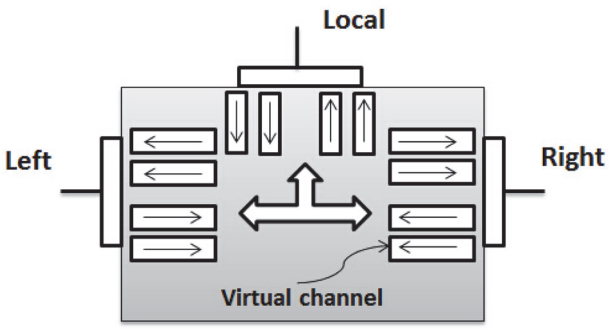


Fig. 4. Microarchitecture of the router.

To reduce the contention in the network, the straightforward way is to partition the virtual channels for CPU and GPU. Also, the injection buffer of the local port is separated to minimize the interferences. In our work, there are 4 input and 4 output virtual channels for each port. We partition them for three configurations (3:1, 2:2, 1:3 for GPU:CPU). This strategy is similar to the previous work [4, 8].

Since there is only one link connecting different routers, an arbitration policy has to be applied to select a packet from the different output virtual channels in the LT stage. In our work, we use first-come first-service (FCFS) policy within CPU and GPU virtual channels. Between the CPU and GPU channels, we always prioritize the CPU packets since the number of the CPU packets is much smaller than that of the GPU packets.

V. COMBINING VCP AND PROBABILITY-BASED LLC CONTROL

To minimize the inter-application interferences, the LLC and interconnection network management should be considered together. However, the probability-based LLC control and VCP are not independent of each other. For example, if the probability-based LLC control is used too aggressively, the number of GPU cache misses may be increased more significantly, which can increase the pressure to NoC and VCP although the number of CPU cache misses can be reduced. This is especially problematic for latency-sensitive CPU applications and GPU applications that are bandwidth-limited, for which the probability-based LLC control should be carefully adapted to avoid large GPU performance degradation. On the other hand, the VCP without considering the probability-based LLC control may be suboptimal, as the numbers of DRAM accesses from CPU and GPU may have been changed by the probability-based method. In this paper, we consider two approaches, a static one and a dynamic one, to combine the VCP and the probability-based LLC control.

A. Statically Combining VCP and the Probability Based Method

We can statically use a small probability value (for example, 0.001) for the LLC replacement by combining with the VCP to achieve better CPU performance. However, as discussed before, a small probability will increase the GPU miss rate, which can lead to more contention in the network and result in relatively high GPU performance overhead. To reduce network traffic, we can use a higher probability (for example, 0.01), which will sacrifice some CPU performance to trade for the GPU performance. The problem is that different applications have different data access behaviors. For some GPU applications, a smaller probability may not lead to large GPU performance degradation. In the static approach, a small probability can be adopted for the maximal benefit of the CPU performance. However, to be adaptive to different application behaviors, a dynamic strategy is needed to make better tradeoffs of CPU and GPU performance.

B. Dynamically Combining VCP and the Probability-Based Method

To guide the dynamic strategy to determine whether a smaller or a larger probability should be used, we define a metric as follows:

$$Pending_{Buffull} = \frac{GPU\ Packets}{Stalled\ Cycles} \quad (1)$$

In this equation, *GPU Packets* are the total number of GPU network packets, and *Stalled Cycles* are the total number of stalled cycles due to network full that has a limited capacity (It can become full if there are too many packets). For the network that has a higher congestion due to the increased GPU packets, the number of GPU packets should increase at higher rate than the increased stall cycles, because for the packets that are already waiting in the line, the number of stalled cycles during a certain period will not increase immediately. We use different fixed probabilities to run the same benchmark multiple times and get the metric that is shown in Table 2. We observed that increase in the probability led to a decrease in the metric as expected. But for some benchmarks, the metric leads to only small changes in the values. This is because when we increase the probability, network congestion does not drop very much. Apparently, we propose a sampling method to predict whether a GPU application has relatively high-performance overhead with a small probability. Two sampling periods are used. After the initiation of the application, we calculate the metric in the first period of time using a small probability 0.001. After that, in the second consecutive period, we calculate it with a larger probability 0.5. If the difference

Table 2. Metric results with a different probability

Benchmark (CPU : GPU)	Probability = 0.001	Probability = 0.01	Probability = 0.1
bzip2 : b+tree	1.05	0.96	0.84
fmm : bfs	1.11	0.93	0.76
milc : b+tree	1.05	0.93	0.81
fmm : lud	0.91	0.89	0.83
water-nsquared : pathfinder	1.26	1.26	1.25
bzip2 : lud	0.84	0.82	0.75
water-nsquared : bfs	1.16	1.03	0.79
fmm : pathfinder	1.07	1.07	1.06
perlbench : gaussian	1.71	1.71	1.63

between the calculated metric in the second period and the predicted value based on the metric of the first period is larger than a threshold (i.e., 0.07, which is determined empirically), we predict that the GPU application is network-sensitive and a larger probability 0.01 is used for the following simulation. Otherwise, we predict the GPU application as not network-sensitive and the probability value of 0.001 is used.

The precondition for the sampling method is that the metric does not change significantly between consecutive periods if the probability does not change; otherwise, we cannot state whether the change in metric is caused by the probability changes or the application itself. The GPU application works in a SIMD way. Although each thread may have different behavior at a time, the phase behavior becomes blurred by other massive numbers of concurrently running threads [4]. Our evaluation shows that the metric behavior is very predictable. Fig. 5 shows that the metric $Pending_{Buffull}$ is relatively stable during consecutive time units, in which each time unit in the X-axis is 2 million cycles. This figure only shows the first 11-time units. In our sampling method, the first 4-time units are used for the first period, and the following 4-time units are used for the second period. The SIMD architecture of the GPU makes all the threads in a warp execute the same instruction but operate on a different portion of the data, and it is unlikely to generate drastically different memory traffic in a short period, even for the irregular application like *bfs*.

If the metric changes significantly by its behavior (although it is very unlikely), the sampling approach will misidentify it and use a higher probability to give GPU applications more space in LLC, which may lead to nonoptimal CPU performance. The CPU performance improvement will be less but still better than pure shared LLC.

VI. DRAM SCHEDULING

The DRAM controllers are also shared between CPU and GPU and hence can potentially affect the system performance. DRAM has a two-dimensional bit array structure. When the DRAM is accessed, a row of bit cells is read into a row buffer first, which is called activation. Read and write operations from the row buffer are called row buffer hits, and it has relatively low latency. If the requested data are not in the row buffer, a precharge operation has to be applied to close the current row and a new row will open. This process needs much longer latency. The first-ready first-come first-service (FRFCFS) [11] scheduling policy is used to prioritize the memory requests that result in row buffer hits, which can increase DRAM throughput. However, in the integrated CPU-GPU heterogeneous architecture, this policy may unfairly prioritize GPU requests against CPU requests since GPUs typically access memory much more frequently and have higher spatial locality due to the SIMD-style execution.

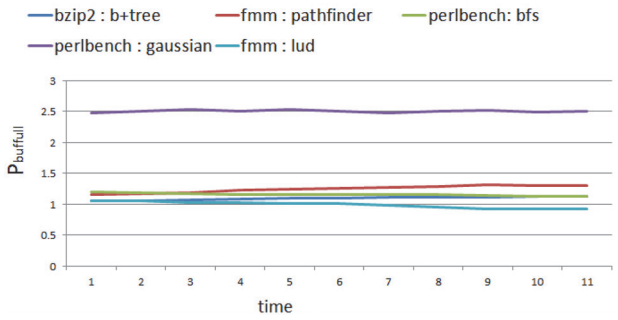


Fig. 5. The metric is relatively stable if the probability does not change.

In this work, DRAM scheduling is not our primary focus, as our evaluation indicates that the inter-applications in the LLC and NoC have a much larger impact on the CPU performance. To further boost CPU performance, we study a DRAM scheduling to prioritize CPU requests in the DRAM controller buffer. Due to the unequal amount of DRAM accesses from CPU and GPU, especially due to the further increase in GPU DRAM accesses after the probability-based LLC control, it becomes important to prioritize the CPU requests to minimize its DRAM access latency. Due to a relatively smaller amount of CPU DRAM accesses, such prioritization is not expected to have large impact on GPU performance.

VII. EXPERIMENTAL METHODOLOGY

A. Simulation Environment

We use MacSim [12], a CPU-GPU heterogeneous simulation framework, to evaluate our work. The architecture

Table 3. Architecture configuration

Component	Specification
CPU	4 cores, 3 GHz
GPU	6 cores, 1.5 GHz
CPU L1 icache per core	32 kB, assoc 8
CPU L1 dcache per core	32 kB, assoc 8
CPU L2 cache per core	256 kB, assoc 8
GPU L1 icache per core	4 kB, assoc 8
GPU L1 dcache per core	32 kB, assoc 8
LLC	4 MB, assoc 16, 1 GHz
Network	Ring topology, 1 GHz
DRAM controller	2 channels, 1.6 GHz
DRAM scheduling policy	FRFCFS

Table 4. Benchmark information

Benchmark	Chosen from	Type	NoC traffic	NoC packets/cycle
bzip2	SPEC2006	CPU	High	0.011
fmm	Splash-2	CPU	Middle	0.005
milc	SPEC2006	CPU	High	0.013
perlbench	SPEC2006	CPU	Middle	0.006
water-nsquared	Splash-2	CPU	Low	0.003
b+tree	Rodinia	GPU	High	0.064
lud	Rodinia	GPU	Low	0.013
pathfinder	Rodinia	GPU	Middle	0.052
gaussian	Rodinia	GPU	Middle	0.040
bfs	Rodinia	GPU	Middle	0.055

simulated is depicted in Fig. 3 in Section II. Each CPU core is a 3 issue, x86 core with out-of-order scheduling. A GPU core contains 8-wide SIMD lanes equipped with texture and constant caches, as well as shared memory. More configurations are shown in Table 3.

The CPU benchmarks are chosen from SPEC2006 [13] and Splash-2 [14], and the GPU benchmarks are from Rodinia [15], all of which are listed in Table 4. Based on the amount of network traffic, we classify each benchmark into high, middle, and low categories as shown in the last column of Table 4. The CPU and GPU applications run concurrently on the simulator. The CPU applications run on the CPU and the GPU applications run on the GPU. The GPU applications are repeated until the completion of CPU applications (since GPU applications are shorter in our simulation) to simulate the on-chip resources contention. In all experiments, the CPU (GPU) performance is measured by the total number of execution cycles. Thus, fewer execution cycles imply better performance.

B. Network and DRAM Latency Breakdown

To figure out how much time is spent on the inter-connection network and DRAM during the simulation cycles, we break down the network and DRAM latencies, which are shown in Fig. 2 in Section I. It is worthy to note that these latency results are not simply to add up all the individual memory request latencies. Instead, we only count the non-overlapped portion of latencies as shown in Fig. 6.

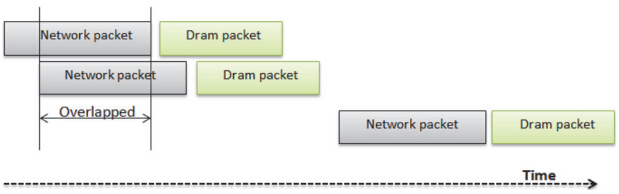


Fig. 6. Calculation of network and DRAM latency.

VIII. EXPERIMENTAL RESULTS

A. The Effectiveness of the Probability-Based Method

We first analyze the effectiveness of the probability-based LLC management method. Fig. 7 shows that as the probability decreases from 0.3 to 0.001, the CPU performance improves in general. This indicates that the probability-based method can effectively limit the evicting power of GPU applications on CPU cache lines as expected. Fig. 8 shows how the CPU miss rates in the LLC change with different probabilities. A smaller probability means more restrictions on the replacement of a CPU cache line caused by a GPU miss thereby leading to a lower CPU miss rate. The higher the probability is, the closer the CPU miss rate is to the case when the LLC is shared without any restriction. For most of the benchmarks, we notice an adequate reduction in CPU LLC miss rates with the probability of 0.001, with up to 47.7% reduction for *fmm* running concurrently with *pathfinder*, as compared to the miss rate of the default shared LLC.

We also find that the CPU performance results and the CPU miss rate results are positively co-related. In

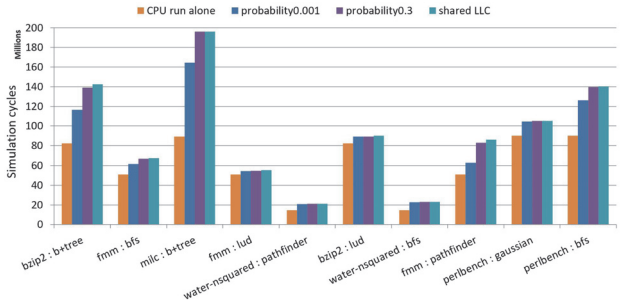


Fig. 7. CPU performance with different probabilities.

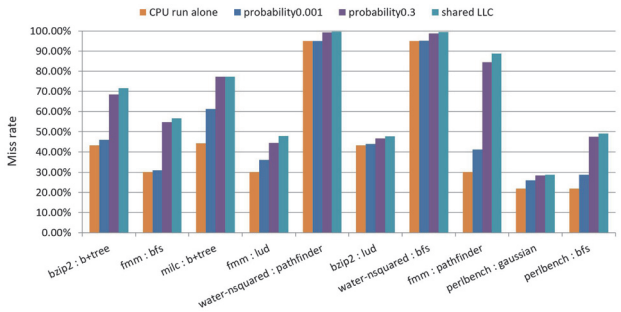


Fig. 8. CPU LLC miss rate with different probabilities.

general, the lower LLC miss rates translate to better CPU performance. However, for the CPU benchmarks that already suffer bad locality and high miss rates even after running alone, for example, *water-nsquared*, their LLC miss rates are only reduced modestly, which results in an insignificant impact on CPU performance. Also, we find that the CPU performance of some benchmarks can be greatly affected by the network the traffic of the concurrent GPU benchmark. For example, since the *b+tree* is classified as a GPU benchmark with high NoC traffic (see Table 4), for the CPU benchmarks *bzip2* and *milc*, there is still a large CPU performance gap between the probability of 0.001 and the CPU running alone. This is likely caused by the intense NoC traffic from the *b+tree*. This also motivates us to study both LLC control and NoC management in an integrated manner to best benefit the CPU performance without significantly affecting the GPU performance. On average, we observe that the CPU performance can be improved by 9% with the probability of 0.001, and up to 27% performance increase is achieved for the CPU benchmark *fmm* that is running with GPU benchmark *pathfinder*.

For GPU, Fig. 9 shows that a smaller probability generally increases the GPU miss rates, because GPU data are forced (based on probability) to replace other GPU data, not CPU data. We also notice that for the GPU benchmark pathfinder, which already has very low locality and high miss rate in the LLC (see in Table 1), the probability-based method has very a limited impact on changing its miss rate.

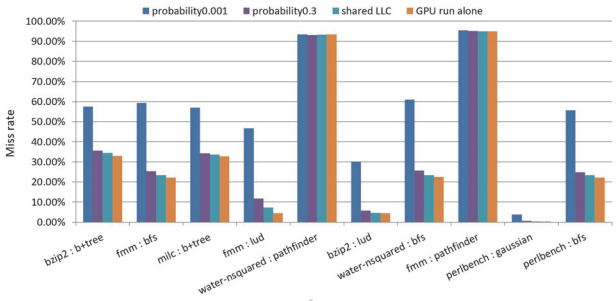


Fig. 9. GPU LLC miss rate with different probabilities.

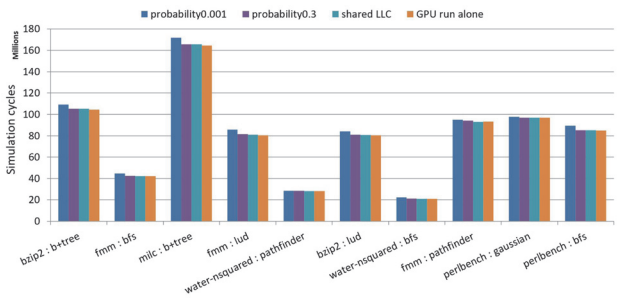


Fig. 10. GPU performance with different probabilities.

However, different from CPU applications that are more sensitive to the LLC miss rates, GPU performance, shown in Fig. 10, does not degrade abundantly with an increase in the GPU LLC miss rate. Again, this is because GPU applications can more effectively tolerate the cache miss latency, which is consistent with the finding of previous studies [6, 7]. With the probability 0.001, the GPU application *lud* has the highest performance overhead (6.3%) when it runs with the CPU application *fmm*. On average, the GPU performance is degraded by about 3.8%.

B. Virtual Channel Partitioning

We evaluate the CPU and GPU performance by partitioning virtual channels in the interconnection network. Figs. 11 and 12 represent the CPU and GPU performance results, respectively, by using different partitioning configurations. These results are obtained with virtual channel partitioning alone, without using the probability-based method for the LLC. As shown in Fig. 11, the CPU performance benefits from separated virtual channels (18% improvement on average), because the partitioned virtual channels prevent GPU packets from blocking CPU packets in the network. Our results reveal that reserving one channel is sufficient for CPU applications to receive most benefits, and allocating more channels for CPU does not result in significant improvement. In contrast, GPU applications provide the best performance when

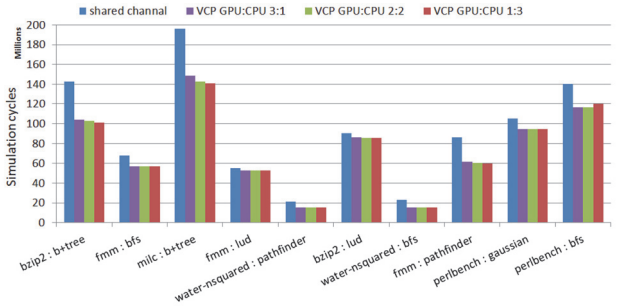


Fig. 11. CPU performance with different partitioning configurations (with the default LLC).

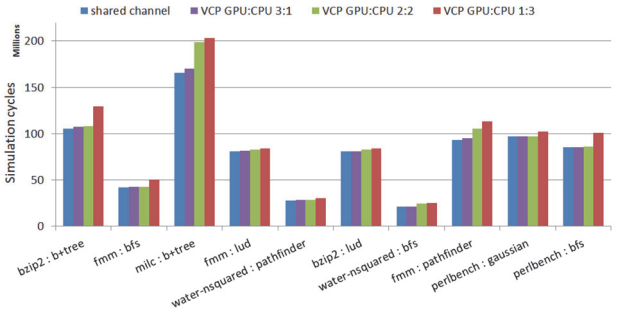


Fig. 12. GPU performance with different partitioning configurations (with the default LLC).

virtual channels are shared. Because GPU applications have much more memory requests than CPU applications, they are more bandwidth sensitive. When the number of GPU virtual channels is reduced to 2 or 1, we observe significant performance reduction in most of the GPU benchmarks. However, for VCP 3:1 (GPU:CPU), the GPU performance is very close to that of the shared channels. This means that reserving one channel for CPU can improve CPU performance significantly with a negligible impact on the GPU performance for the benchmarks we have studied.

C. Statically Combining the Probability and VCP

Since contention exists in both LLC and interconnection networks, focusing on the LLC or network alone may be suboptimal for the improvement of the CPU performance. Combining LLC and network management effectively can further improve the CPU performance. Figs. 13 and 14 show the CPU and GPU performance results by statically combining VCP with probability-based control (with probability 0.001), which is also compared to the performance of VCP alone and the probability of 0.001 alone, respectively. As we can see, statically combining VCP with the probability-based LLC control provides better CPU performance than either VCP alone or the probability-based LLC control alone.

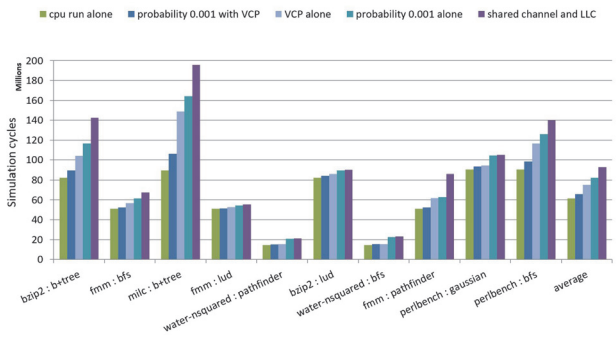


Fig. 13. CPU performance by statically combining the probability based LLC control and VCP.

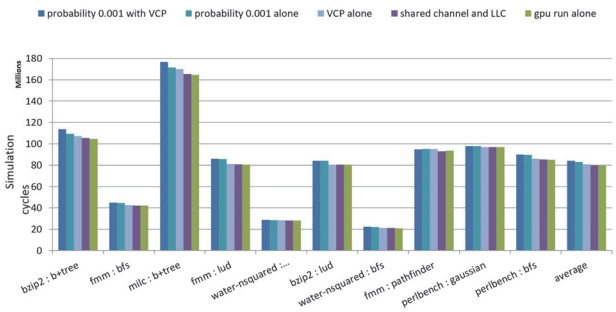


Fig. 14. GPU performance by statically combining the probability based LLC control and VCP.

As can be seen in Fig. 14, although the GPU performance becomes worse, it is not as sensitive as the CPU applications. Combining probability 0.001 and VCP is the closest one to the CPU baseline performance (i.e., when the CPU applications run alone). On average, it is observed that the CPU performance can be improved by 26% and the GPU performance overheads are about 5% by combining with probability 0.001 and VCP.

D. Dynamically Combining the Probability and VCP

Fig. 15 shows the normalized CPU performance results of dynamically combining the probability-based LLC control and VCP, which are normalized to the total number of CPU execution cycles in the default integrated CPU-GPU with a shared LLC and NoC. For CPU benchmarks, the dynamic strategy significantly improves the performance over the baseline for all benchmarks, which is only slightly less than the static strategy with the probability of 0.001.

Fig. 16 represents the normalized GPU performance results of dynamically combining the probability-based LLC control and VCP, which are normalized to the total number of GPU execution cycles in the default integrated CPU-GPU with a shared LLC and NoC. We observe that

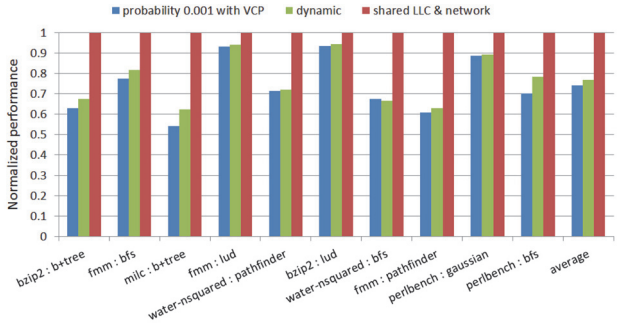


Fig. 15. Normalized CPU performance by dynamically combining the probability based LLC control and VCP.

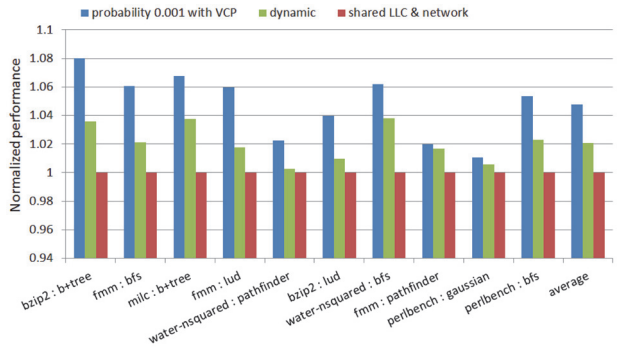


Fig. 16. Normalized GPU performance by dynamically combining the probability based LLC control and VCP.

the dynamic strategy has less GPU performance overhead than the static strategy. For some application pairs such as *water-nsquared:pathfinder*, *bzip2:lud*, *fmm:pathfinder*, and *perlbench:gaussian*, the GPU performance is not very network-sensitive. For example, the miss rate of *pathfinder* is already very high even after running alone (refer to Table 1) and remains stable regardless of the probability change. The *gaussian* has very good locality and high hit rate, thus its performance is only improved slightly with the dynamic strategy. Although the miss rate of *lud* increases dramatically, the total number of network packets is relatively small, resulting in limited performance improvement. For these benchmarks, the dynamic strategy can effectively choose the probability of 0.001 to maximize CPU performance with reduced GPU performance overheads. For other benchmarks, the dynamic strategy chooses to sacrifice some CPU performance to improve GPU performance with a larger probability of 0.01. On average, the GPU performance overhead is reduced to 2% with 24% CPU performance improvement. Thus, if GPU performance degradation has tight constraints, the dynamic approach is more desirable than the static strategy.

E. More Discussion about Interrelation between Network and LLC

In our implementation, the initial probability is set to 0.001. It is assumed is that GPU applications are generally cache insensitive. We use small probability to provide more LLC space to CPU applications. There are two cases for the cache insensitive GPU applications when a small probability is applied. The first case is that the GPU LLC miss rate does not change much (i.e., streaming applications). A small probability will reduce the CPU conflict misses in the LLC without increasing the network traffic. In the second case, the GPU LLC miss rate increases significantly and the network has higher pressure. GPU performance degradation mainly comes from the higher network congestion. In this case, the network feedback will suggest using a higher probability to yield more space to the GPU application to reduce the network traffic. The more memory-intensive GPU applications are, the more important the network feedback is. In Fig. 16, the *b+tree* suffers an 8% performance loss without the network feedback when it runs with *bzip2*. This overhead is reduced to 3.6% by applying the dynamic approach that considers the network feedback. We would expect more performance overhead on the more memory intensive, but cache insensitive GPU applications if we blindly use a small probability without considering the network traffic.

For the cache sensitive GPU applications, a small probability also increases the GPU LLC miss rate, which is similar to the second case that we have just discussed. The network feedback prevents the GPU performance from significant degradation.

The network virtual channel partition is also important. Without the VCP, the possible increase in network congestion will have more interferences to the CPU requests and limit the benefits of reduced conflict CPU misses in the LLC. However, the VCP alone is not enough to achieve superior CPU performance. Fig. 13 shows that on average, combining the VCP and our LLC replacement policy can lead to extra 8% improvement compared to the VCP alone.

F. CPU DRAM Requests Prioritization

We also evaluate whether prioritizing CPU requests in the DRAM controller will have large performance impact on both CPU and GPU applications. Figs. 17 and 18 demonstrate the normalized CPU and GPU performance with and without the CPU DRAM request prioritization, both of which combine the VCP and probability 0.001 together and are normalized to the CPU and GPU execution cycles, respectively, without the DRAM request prioritization. As we can see, the CPU performance is enhanced but the improvement is rather limited (on average, it is only about 1%). The GPU performance degradation is still negligible. These results are consistent with the previous results shown in Fig. 2, i.e., the DRAM latency only counts for a limited portion of the overall performance, and the interconnection network is the main performance bottleneck.

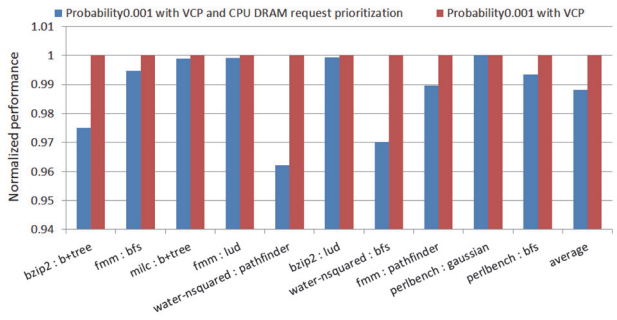


Fig. 17. CPU performance improvement with CPU DRAM requests prioritization.

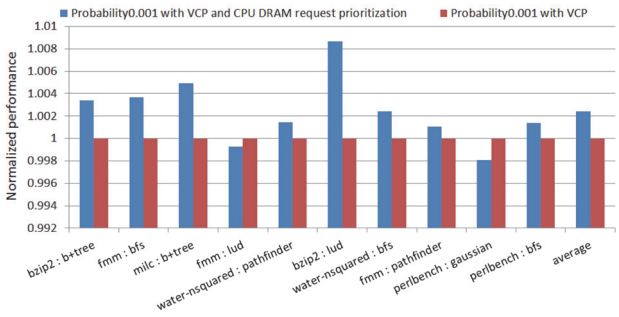


Fig. 18. GPU performance degradation with CPU DRAM requests prioritization.

G. Comparing TAP-UCP with Probability Based LLC Replacement

TAP-UCP is proposed in [7], which is a TLP-Aware cache management policy for a CPU-GPU heterogeneous architecture. This method is based on UCP [10], a dynamic cache partitioning mechanism for only CPU workloads. The UCP tries to find optimal last level cache partitioning between different CPU applications. However, UCP tends to favor GPU applications in heterogeneous workloads since GPU applications have much more memory requests than CPU applications. Therefore, in the TAP-UCP, the GPU accesses are reduced by a normalization factor that is determined by the access ratio between CPU and GPU periodically.

We implemented the TAP-UCP and compared the performance results with our probability-based LLC replacement policy. Figs. 19 and 20 represent the CPU and GPU performance comparison between TAP-UCP and our LLC replacement policy with probability 0.001, respectively. Both TAP-UCP and our method are working on a shared network. So, the performance difference comes from LLC management only. The performance is normalized to the results when the LLC is shared (without any management).

The TAP-UCP dynamically partitions the LLC between CPU and GPU. We also statically partition the cache ways between CPU and GPU for comparison. Considering that CPU applications are more cache sensitive, 75% of the cache ways of each set are allocated to CPU, and the rest is allocated to GPU. Both TAP-UCP and static LLC partition are not optimal for CPU performance.

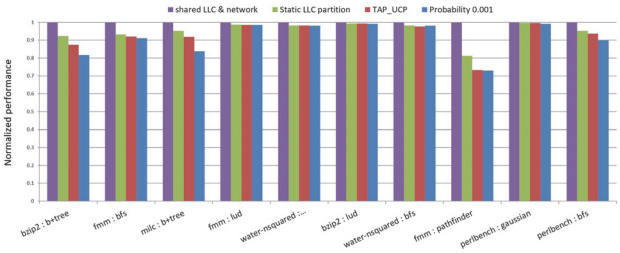


Fig. 19. CPU performance comparison between TAP-UCP and probability based LLC replacement.

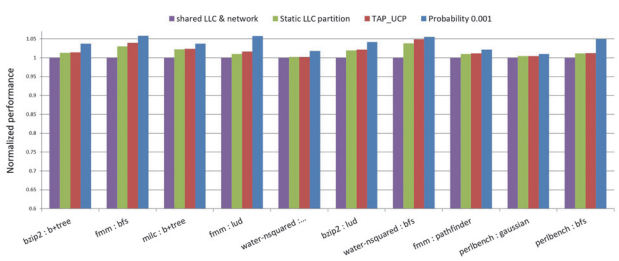


Fig. 20. GPU performance comparison between TAP-UCP and probability based LLC replacement.

The TAP-UCP tends to provide more LLC space to GPU applications compared to our method with probability 0.001. On average, the TAP-UCP has a slightly better GPU performance (about 1.8%) than our method. But our method performs 2% better than TAP-UCP on average for the CPU applications. For some CPU benchmark like *milc*, the probability method is 8% better than TAP-UCP in terms of CPU performance.

Overall, the TAP-UCP and the probability-based method are comparable for the performance. But our method has much less complexity. Both methods need modification to the original cache line replacement control logic. Other than that, we only need one random number generator. But for the TAP-UCP, an LRU stack is required for each sampled set (UCP collects the information only from sampled sets to reduce the overhead of maintaining an LRU stack for each set [7]) and a hit counter for each way in all the sampled sets. It also needs to keep track of the access ratio of CPU and GPU.

H. Sensitivity Study

We reduce the cache size by half (reducing the number of sets with the same associativity) to do the sensitivity study. The CPU and GPU performance results of different methods are shown in Figs. 21 and 22, which are normalized to the CPU and GPU execution cycles, respectively, of the integrated CPU-GPU with the shared LLC and NoC. With a smaller LLC, the interconnection network has more traffic due to the increased miss rates

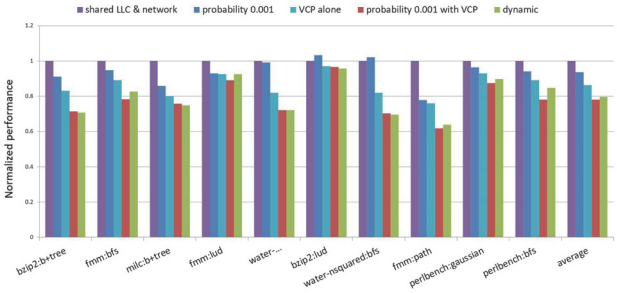


Fig. 21. Normalized CPU performance with a smaller cache.

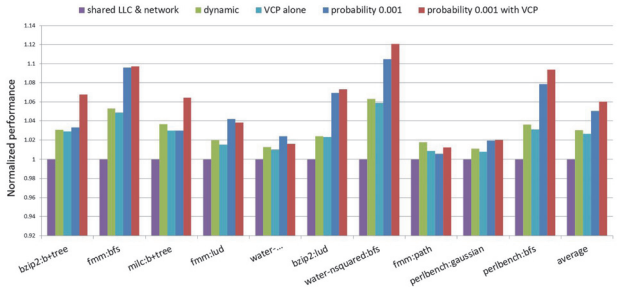


Fig. 22. Normalized GPU performance with a smaller cache.

of the LLC. Some of the CPU benchmarks like *bzip2* and *water-nsquared* have even worse performance with the probability 0.001 alone since the network contention becomes a dominant factor. On average, CPU performance can be improved by 7% with probability 0.001 alone, and by 22% improvement by combining the probability 0.001 and VCP. On average, the dynamic approach increases the CPU performance by 21% with 3% GPU performance overhead.

IX. RELATED WORK

Techniques that control interferences among different applications can be generally classified into two categories. The first one is core management or throttling schemes. The key idea is to reduce the number of threads such that the congestion in on-chip network and memory is reduced. Some core throttling techniques are proposed in [16-18] for CPUs. Kayiran et al. [19] studied a dynamic CTA scheduling algorithm to allocate an optimal number of CTAs per core to reduce contention in the memory for GPGPU architecture. The objective of GPU thread throttling is to keep the GPU cores busy, but not with too much work. Based on the same idea, Kayiran et al. [3] proposed to manage GPU concurrency in the CPU-GPU heterogeneous architecture.

Alternatively, the inter-application interference problem can be solved by managing shared resources at inter-connection network, cache, and memory. Qureshi and Patt [10] proposed Utility-Based Cache Partitioning in the context of multi-core CPUs to dynamically allocate more cache ways in the LLC to the applications that can benefit greatly from larger caches. However, the Utility-Based Cache Partitioning may allocate more cache ways to GPU applications that are not sensitive to the cache size for the integrated CPU-GPU heterogeneous architecture [7]. Thus, Lee and Kim [7] designed a modified version of it, TLP-aware cache management policy (TAP), to work on the heterogeneous architecture. Mekkat et al. [6] proposed heterogeneous LLC management (HeLM) to take advantage of the GPU's tolerance for memory access latencies to throttle GPU LLC accesses and yield LLC space to cache-sensitive CPU applications.

Subramanian et al. [20] studied the application slowdown model to accurately estimate application slowdowns at both the shared cache and main memory. For the interconnection network management, Trivino et al. [21] explored the benefits of virtualizing the NoC with a partitioning mechanism for Chip Multiprocessor systems (CMPs). Lee et al. [4, 8] proposed virtual channel partitioning to reduce interferences between CPU and GPU packets in heterogeneous architectures. Jang et al. [22] designed virtual channel monopolizing and asymmetric virtual channel partitioning for GPGPUs. For memory scheduling management between CPU and PU applications, Jeong et al. [23] and Ausavarungnirun et al. [5] dynamically adapted the priority of CPU and GPU memory requests based on tracking information of GPU workloads. However, none of the shared resources management techniques considers both the LLC and interconnection network in an integrated manner to boost CPU performance without any significant impact on GPU performance for the integrated CPU-GPU architecture, which is done in the present work.

X. CONCLUSION

Shared resources contention can have a significant impact on the performance of applications running concurrently on the integrated CPU-GPU architecture. It is important to reduce the inter-application interferences in such systems to achieve balanced performance improvement. This problem can be solved from two different angles. One is the throttling scheme that reduces the memory subsystem traffic mentioned in the related work. The other is to manage shared resources more effectively. In this paper, we focus on the management of the shared resources in an integrated manner to substantially reduce the GPU's negative impact on CPU performance.

For the integrated CPU-GPU heterogeneous architecture shown in Fig. 3, contentions in the shared LLC and virtual channels of the interconnection network are the major factors that can disparately slow down the CPU applications. To solve these problems, we propose a new probability based method to effectively control the GPU's evicting power on CPU cache lines in the LLC without any significant impact on GPU performance. Moreover, to further improve the CPU performance, we propose to combine the VCP and the probability based method either statically or dynamically to reduce the contentions in both the LLC and NoC. The static integration of VCP with a fixed probability of 0.001 can lead to the best CPU performance but it has relatively large performance overhead for GPU applications. Dynamically combining VCP and the probability-based LLC control method can improve CPU performance by 21% with only 3% GPU performance degradation on average. We also find that the DRAM latency alone contributes to only a small portion to the overall performance, and prioritizing the CPU requests to the DRAM has a very limited impact to further boost the CPU performance.

Currently, the probability values are based on experimental results. These values may not be the best across different architectures. Our future work is directed towards the design of a model considering some major architectural features.

ACKNOWLEDGMENT

This paper is based on our work published in 2018: H.

Wen and W. Zhang, "Reducing inter-application interferences in integrated CPU-GPU heterogeneous architecture," in *Proceedings of 2018 IEEE 36th International Conference on Computer Design (ICCD)*, Orlando, FL, 2018, pp. 278-281.

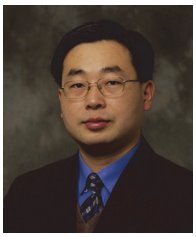
This work was funded in part by the NSF (No. CNS-1421577).

REFERENCES

1. Khronos OpenCL Working Group, "The OpenCL Specification," 2008; https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_API.html.
2. NVIDIA, "CUDA C/C++ SDK Code Samples," 2020; https://docs.nvidia.com/cuda/cuda-samples/index.html.
3. O. Kayiran, N. C. Nachiappan, A. Jog, R. Ausavarungnirun, M. T. Kandemir, G. H. Loh, O. Mutlu, and C. R. Das, "Managing GPU concurrency in heterogeneous architectures," in *Proceedings of 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, Cambridge, UK, 2014, pp. 114-126.
4. J. Lee, S. Li, H. Kim, and S. Yalamanchili, "Adaptive virtual channel partitioning for network-on-chip in heterogeneous architectures," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 18, no. 4, article no. 48, 2013.
5. R. Ausavarungnirun, K. K. W. Chang, L. Subramanian, G. H. Loh, and O. Mutlu, "Staged memory scheduling: achieving high performance and scalability in heterogeneous systems," in *Proceedings of 2012 39th Annual International Symposium on Computer Architecture (ISCA)*, Portland, OR, 2012, pp. 416-427.
6. 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 (PACT)*, Edinburgh, UK, 2013, pp. 225-234.
7. 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 (HPCA)*, New Orleans, LA, 2012, pp. 1-12.
8. J. Lee, S. Li, H. Kim, and S. Yalamanchili, "Design space exploration of on-chip ring interconnection for a CPU–GPU heterogeneous architecture," *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1525-1538, 2013.
9. B. Valentine, "Introducing Sandy Bridge," 2010; https://en.wikichip.org/w/images/f/fa/introducing_sandy_bridge.pdf
10. 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.
11. S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens, "Memory access scheduling," in *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, Vancouver, Canada, 2000, pp. 128-138.
12. H. Kim, J. Lee, N. B. Lakshminarayana, J. Sim, J. Lim, and T. Pho, "Macsim: a CPU-GPU heterogeneous simulation framework user guide," 2012; https://code.google.com/p/macsim/.
13. Standard Performance Evaluation Corporation, "SPEC CPU 2006," 2018; https://www.spec.org/cpu2006/.
14. Splash 2 for Linux/x86, http://kbarr.net/splash2.
15. 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.
16. K. K. W. Chang, R. Ausavarungnirun, C. Fallin, and O. Mutlu, "HAT: heterogeneous adaptive throttling for on-chip networks," in *Proceedings of 2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, New York, NY, 2012, pp. 9-18.
17. H. Y. Cheng, C. H. Lin, J. Li, and C. L. Yang, "Memory latency reduction via thread throttling," in *Proceedings of 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, Atlanta, GA, 2010, pp. 53-64.
18. M. Thottethodi, A. R. Lebeck, and S. S. Mukherjee, "Self-tuned congestion control for multiprocessor networks," in *Proceedings HPCA 7th International Symposium on High-Performance Computer Architecture*, Monterrey, Mexico, 2001, pp. 107-118.
19. O. Kayiran, A. Jog, M. T. Kandemir, and C. R. Das, "Neither more nor less: optimizing thread-level parallelism for GPGPUs," in *Proceedings of the 22nd international Conference on Parallel Architectures and Compilation Techniques*, Edinburgh, UK, 2013, pp. 157-166.
20. L. Subramanian, V. Seshadri, A. Ghosh, S. Khan, and O. Mutlu, "The application slowdown model: Quantifying and controlling the impact of inter-application interference at shared caches and main memory," in *Proceedings of 2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Waikiki, HI, 2015, pp. 62-75.
21. F. Trivino, F. J. Alfaro, and J. Flich, "Exploring NoC virtualization alternatives in CMPs," in *Proceedings of 2012 20th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, Garching, Germany, 2012, pp. 473-482.
22. H. Jang, J. Kim, P. Gratz, K. H. Yum, and E. J. Kim, "Bandwidth-efficient on-chip interconnect designs for GPGPUs," in *Proceedings of the 52nd Annual Design Automation Conference*, San Francisco, CA, 2015.
23. M. K. Jeong, M. Erez, C. Sudanthi, and N. Paver, "A QoS-aware memory controller for dynamically balancing GPU and CPU bandwidth use in an MPSoC," in *Proceedings of the Design Automation Conference (DAC)*, San Francisco, CA, 2012, pp. 850-855.

Hao Wen

Hao Wen is a Ph.D. student at the Department of Electrical and Computer Engineering of Virginia Commonwealth University. He received his bachelor's degree in Electrical Engineering in July 2007 from Southeast University, Nanjing, China, and master's degree in Microelectronics in July 2010 from Peking University, Beijing, China. He worked as an IC verification engineer in VIMICRO, Beijing, and Spreadtrum, Shanghai from 2010 to 2013. Hao Wen's research focuses on computer architectures and GPU computing.



Wei Zhang

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.