JCSE Journal of Computing Science and Engineering

Regular Paper

Journal of Computing Science and Engineering,
Vol. 11, No. 2, June 2017, pp. 58-68

Warp-Based Load/Store Reordering to Improve GPU Time Predictability

Yijie Huangfu and Wei Zhang*

Department of Electrical and Computer Engineering, Virginia Commonwealth University, Richmond, VA, USA
huangfuy2@vcu.edu, wzhang4@vcu.edu

Abstract

While graphics processing units (GPUs) can be used to improve the performance of real-time embedded applications that require high throughput, it is challenging to estimate the worst-case execution time (WCET) of GPU programs, because modern GPUs are designed for improving the average-case performance rather than time predictability. In this paper, a reordering framework is proposed to regulate the access to the GPU data cache, which helps to improve the accuracy of the estimation of GPU L1 data cache miss rate with low performance overhead. Also, with the improved cache miss rate estimation, tighter WCET estimations can be achieved for GPU programs.

Category: Embedded Computing

Keywords: GPU; Data cache; WCET estimation

I. INTRODUCTION

Graphics processing units (GPUs) are now broadly used as accelerators for compute- and/or data-intensive applications. Especially, for applications that can be parallelized into a massive number of threads with few dependencies among one another, GPUs are the ideal accelerators to boost the performance, due to their single-instruction multiple-thread (SIMT) execution model and the large number of computing units/cores that execute the threads simultaneously [1]. Such kinds of applications are also common in embedded systems, such as the physics simulations [2] and human pose recognition [3], etc. Therefore, there are more GPUs and GPU systems targeting embedded applications, e.g., the Tegra processors [4] and the DRIVE PX2 [5] platform.

Applications in hard real-time systems, like traffic sign recognition and autonomous navigation, also need to process a massive amount of data, where GPUs can be a promising accelerator, if we can safely and tightly estimate the worst-case execution time (WCET) of these programs on GPUs. However, many advanced GPU architectural features, such as dynamic scheduling, out-of-order execution, etc., harm the time predictability, though they benefit the average-case performance considerably. Specifically, the cache memory, whose behavior is dependent on runtime memory access history and is hard to predict statically, is one of the major sources that harm time predictability. Due to the large number of threads and the dynamic warp scheduling in GPUs, the cache behavior is not just dependent on the runtime memory access history, but also on the scheduling and execution orders of the threads, warps and the instructions in each thread. This makes the WCET analysis for GPU caches much more complicated and challenging. Modern GPUs use all kinds of cache memory to improve the average-case performance. Therefore, it is necessary to improve the time predictability of GPU cache memory to

Open Access http://dx.doi.org/10.5626/JCSE.2017.11.2.58 http://jcse.kiise.org

Received 21 April 2017; **Accepted** 02 June 2017
*Corresponding Author

 pISSN: 1976-4677 eISSN: 2093-8020

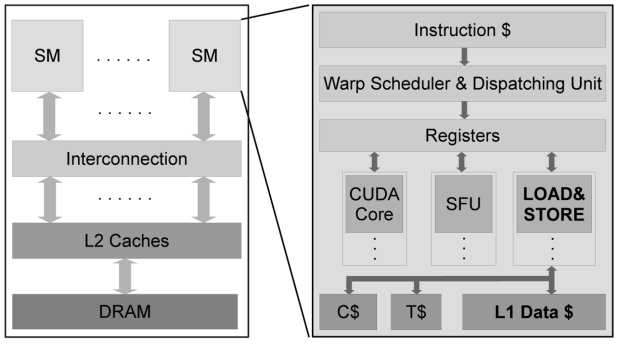


Fig. 1. Basic NVIDIA GPU architecture.

enable GPU real-time computing.

In this study, we build a framework of compiler-based static analysis and architectural extensions to enable tight and accurate miss rate analysis for GPU L1 data caches, while allowing the dynamic warp scheduling, which is critical for performance. Experiment results reveal that the proposed framework can tightly estimate the miss rate of GPU L1 data cache, while achieving better performance than a pure round- robin scheduling. By providing the estimated miss rate information to our GPU WCET analyzer, tighter WCET estimations can be achieved as well.

The rest of the paper is organized as follows. The background information about GPU, GPU memory architecture and the dynamic scheduling and execution is introduced in Section II. Section III describes the proposed reordering framework. Section IV describes the WCET estimator which can benefit from knowing the estimated L1 data cache miss rate. The evaluation methodology and experimental results are presented in Section V. Section VI reviews the related work and we make a conclusion in Section VII.

II. GPU ARCHITECTURE AND DYNAMIC BEHAVIORS

A. GPU Architecture

The basic architecture of an NVIDIA GPU—NVIDIA CUDA (Compute Unified Device Architecture) [6] terminologies are used in this paper, though our method can also be applied to other GPUs—is shown in Fig. 1. On a GPU chip, streaming multiprocessors (SMs) share a unified L2 cache and the global memory. Inside each SM, there are L1 data, instruction, constant and texture caches, while, in this paper, we focus on the L1 data cache, which is a major obstacle for time predictability.

GPU kernels are the general purpose programs that run on GPUs and can be written in CUDA C [6] or OpenCL [7]. CPUs set up the kernel configurations and launch the kernel. The following code shows how a CPU configures and launches a CUDA GPU kernel. An integer and a float pointer are passed to the kernel *K1* by the variables *m* and *A*. The *gridDim* and *blockDim* set the kernel to be with 256 (16 × 16 × 1) threads in one kernel block and 64 (8 × 4 × 2) kernel blocks, i.e., 16384 (256 × 64) threads in total.

```
...
int   m;
float  *A;
...
dim3  gridDim ( 8 , 4, 2 ) ;
dim3  blockDim ( 16 ,16 , 1 ) ;
K1<<<gridDim ,  blockDim>>>(m, A ) ;
...
```

In the execution of the kernel, each thread has its own thread ID and block ID. A *warp*, which contains 32 threads with consecutive IDs in the same kernel block, is the basic scheduling unit in kernel executions. In this example, there are 8 warps in one block and 512 warps in the kernel. Fundamentally, the term SIMT here means threads in the same warp move forward in a locked-step fashion; therefore, instructions in CUDA GPU kernel are also called warp instructions.

B. GPU Dynamic Behaviors

1) Dynamic Warp Scheduling

Whenever a warp instruction cannot be issued, e.g., the dependency is not met or the needed resource is unavailable, the warp scheduler will issue other active warps. Therefore, for the same warp instruction in, for example, 3 warps, the issuing order can be any out of the 6 possible combinations.

2) Out-of-Order Execution

The execution order of instructions in a warp does not always follow the order of the instructions in the kernel program, because, if dependencies are met and the required resources are available, an instruction can be executed immediately. Therefore, a trailing instruction can be executed earlier. For example, even if instruction *I1* is behind *I0*, the execution of *I1* can be earlier than *I0*.

Things become more complicated, when dynamic scheduling and out-of-order execution are combined. An example with 3 warps and 2 instructions is shown in Fig. 2. Although the figure only shows 2 possible orders, the total number of possible execution orders of 3 warps and 2 instructions can be as many as 6! (or 720).

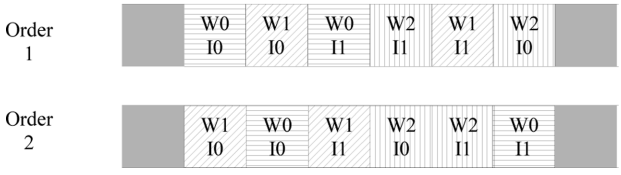


Fig. 2. Example of possible execution orders of warp instructions.

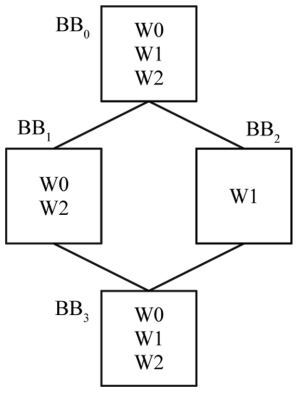


Fig. 3. Example of warp and basic block relations.

3) Independent Execution among Warps

Warps execute independently to each other if no special synchronization instruction is used. There is no synchronization at the boundaries of basic blocks by default. In the example shown in Fig. 3, *W0* and *W2* have different execution path than BB_0. A possible scenario example is that *W1* is in BB_3, when *W0* and *W2* are still in BB_1.

III. GPU L1 DATA CACHE ACCESS REORDERING FRAMEWORK

A. Challenges of GPU Cache Timing Analysis

Abstract interpretation is a technique that has been successfully used in cache timing analysis for CPUs [8]. A basic assumption of this analysis method is that, for each basic block, the memory access sequences to the cache can be statically derived from the control flow graph. However, this assumption *cannot* be guaranteed at all in GPUs due to the behavior of GPUs.

For CPUs, a range of memory space can be used for a data access whose address is unpredictable in static timing analysis. However, this is unaffordable for GPU kernels, because the large number of threads can easily have a huge number of memory accesses, which can lead to overly pessimistic and useless WCET estimations. Therefore, in this paper we assume the data access to the L1 data cache and the branch conditions are predictable, which is common in GPU kernels that access data and operate based on the thread and block IDs (There are some programming guidelines for hard real-time software running on CPUs, a similar guideline for developing real-time GPU kernels may be needed to ensure the compliance with this assumption).

B. Issues of Regulating the Warp Scheduling Orders

By reducing the dynamic behavior, e.g., enforcing a pure round-robin warp scheduling policy, the time predictability of GPUs can be improved, which, however, comes with considerable performance overhead, as shown in Section V. But, in most cases, even a pure round-robin warp scheduling policy still cannot guarantee the order of different warp instructions in different basic blocks, due to the independent execution of different warps, as explained in Section II-B-3. Moreover, regulating the warp execution order does not change the out-of-order execution of instructions in a certain warp, which can still impact the predictability of GPU data caches. So, the goal of this paper is to have predictable cache behavior while minimizing the constraints on the dynamic GPU behavior and the performance overhead.

C. The Reordering Framework

The proposed reordering framework has three major components, which are a CUDA kernel analyzer, a GPU L1 data cache miss rate estimator, and an architectural extension for warp-based load/store reordering. The kernel analyzer takes in a CUDA kernel and outputs the reordering configuration and the information of memory access of the kernel. The miss rate estimator uses the information from the analyzer and gives the estimation of the L1 data cache miss rate of the kernel. The reordering extension in the GPU regulates memory traffic to L1 data cache based on the reordering configuration. The details of these three parts, as shown in Fig. 4, are discussed in the following subsections.

D. Kernel Analyzer

The kernel analyzer, whose pseudo code is shown in Algorithm 1, uses the PTX code [9], the kernel input values, and the hierarchy configuration of the kernel to generate the desired L1 data cache access pattern and the memory access addresses of the global load/store instructions in the kernel. With the information of the kernel collected, the analyzer generates the control flow graph and the information about global load/store instructions in the kernel. For every warp in each kernel block, the analyzer parses the kernel with the information of the warp (block and warp IDs). Algorithm 2 describes the pseudo code of the parser.

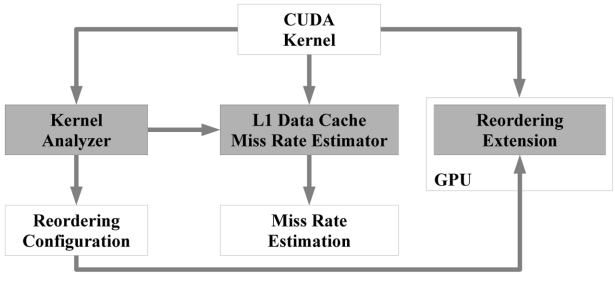


Fig. 4. General structure of the reordering framework.

Algorithm 1 Kernel Analyzer

```
1: Inputs = CollectKernelInputs(FileKernel, FileInput);
2: CFG = GenerateKernelCFG(FileKernel, FileInput);
3: LDSTPCList = GenerateLDSTPCs(FileKernel);
4: BlockAddrInfo = [];
5: BlockAccInfo = [];
6: for Each Kernel Block B_i ∈ k do
7:     for Each Warp W_i ∈ B_i do
8:         [WarpAddrInfo, WarpAccInfo] = KernelParser(Inputs,
   CFG, LDSTPCList, B_i, W_i);
9:     end for
10:    BlockAddrInfo.append(WarpAddrInfo);
11:    BlockAccInfo.append(WarpAccInfo);
12: end for
13: Return [LDSTPCList, BlockAddrInfo, BlockAccInfo];
```

Algorithm 2 GPU Kernel Parser

```
1: procedure KernelParser(Inputs, CFG, LDSTPCList, Block,
   Warp)
2:     INST = FirstInstruction(CFG);
3:     WarpAddrInfo = []×length(LDSTPCList);
4:     WarpAccInfo = []×length(LDSTPCList);
5:     while INST is not Exit do
6:         if INST is arithmetic instruction then
7:             UpdateRegisterValue(INST, Inputs, Block, Warp);
8:         end if
9:         if INST is global load/store then
10:            pc = GetInstPC(INST);
11:            pcidx = GetPCIndex(INST, LDSTPCList);
12:            AddrList = AddrListGen(INST, Warp);
13:            WarpAccInfo[pcidx]=True;
14:            WarpAddrInfo[pcidx]=AddrList;
15:        end if
16:        if INST is branch or end of current BB then
17:            INST = FindNextBB(CFG, INST);
18:        else
19:            INST = NextInstCurBB();
20:        end if
21:    end while
22:    Return [WarpAddrInfo, WarpAccInfo];
23: end procedure
```

Algorithm 3 Addresses Generation for Instruction *I* and Warp *W*

```
1: procedure AddrListGen(I, W)
2:     AddrList = []
3:     for Each Thread T_i ∈ W do
4:         if CheckAcitve(T_i) then
5:             CurAddr = GetAddr(T_i, I)
6:             Coalesced = False
7:             for Each Address A_j ∈ AddrList do
8:                 if Coalesce(CurAddr, A_j) then
9:                     Coalesced = True
10:                    Break
11:                end if
12:            end for
13:            if Not Coalesced then
14:                AddrList.append(CurAddr)
15:            end if
16:        end if
17:    end for
18:    Return AddrList
19: end procedure
```

The *KernelParser* starts with the first instruction of the kernel then follows the control flow graph. For an arithmetic instruction, the value of the target operand is updated. For a global load/store instruction, several actions are carried out. The parser first coalesces (Algorithm 3) all the addresses used by the threads in the warp to form a list of addresses of this warp instruction, which will be used later for the worst-case L1 data cache miss rate estimation as the memory access addresses from this warp instruction. Then, the corresponding value in the *WarpAccInfo* list will be set as true to indicate that this load/store instruction will access the L1 data cache. The parser finds the next basic block based on the control flow graph, if the current instruction is a branch at the end of the current basic block. The following example explains how the analyzer works.

```
. entry _example (
. param . u64 __cudaparm_input_cuda)
{
. reg . u32 %r<29>;
. reg . u64 %rd<33>;
. reg . f32 %f<20>;
. reg . pred %p<6>;
$Lbegin :
  ld . param. u64       %rd5 , [__cudaparm_in
  put_cuda ] ;
  cvt . s32 . u32       %r3 , %t i d . x ;
  mul . wide . s32      %rd3 , %r3 , 3 2 ;
  add . u64             %rd8 , %rd5 , %rd3 ;
  cvt . s32 . u32       %r1 , %c t a i d . y ;
  mov . s32             %r2 , 0 ;
  s e tp . eq . s32     %p1 , %r1 , %r2 ;
  @!%p1 bra             $L1 ;
  ld . g l o b a l . f32  %f1 , [%rd8 +4] ;
  bra $L2 ;
$L1 :
  st . g l o b a l . f32  [%rd8+2048] , %f 2 ;
$L2 :
  e x i t ;
$Lend :
} // _example$
```

For example, if the hierarchy of the above kernel is <<< *dim*3(1, 2, 1), *dim*3(16, 4, 1) >>> and the input value of *__cudaparm input cuda* is 0, the results of the analyzer are shown as follows. The analyzer output shows that the kernel has 2 global load/store instructions. There are 2 kernel blocks; each has 2 warps. The warps in the first kernel block execute the first load/store instruction, while the warps in the second kernel block execute the second load/store instruction. The list of memory access addresses and access types are also shown in the output.

```
–num_pcs 2
–pc_addrs [ 6 4 , 8 0 ]
–grid [ 1 , 2 , 1 ]
–block [ 1 6 , 4 , 1 ]
Block [ 0 , 0 , 0 ]
Warp0 [ 1 , 0 ]
Warp0 [ [ [ 0 , 128 , 256 , 3 8 4 ] , L] , None ]
Warp1 [ 1 , 0 ]
Warp1 [ [ [ 0 , 128 , 256 , 3 8 4 ] , L] , None ]
Block [ 0 , 1 , 0 ]
Warp0 [ 0 , 1 ]
Warp0 [None , [ [ 2 0 4 8 , 2176 , 2304 , 2 4 3 2 ] , S ] ]
Warp1 [ 0 , 1 ]
Warp1 [None , [ [ 2 0 4 8 , 2176 , 2304 , 2 4 3 2 ] , S ] ]
```

E. Architectural Extension for Warp-Based Load/Store Reordering

The proposed architectural extension is used to regulate the memory access before they go to the L1 data cache so that the load/store access order can be predictable, which thus enables accurate cache timing analysis. The modification to the default relation between the load/store unit (*LD/ST*) and the L1 data cache is shown in Fig. 5. A channel is added for each active warp 3, and each channel is a FIFO-like buffer to hold load/store requests from a warp to the L1 data cache. Besides the head and tail pointer of a normal FIFO, an extra search pointer is used to allow the *Reordering Unit* to search for the expected memory access in each channel buffer, which enables the reordering of memory access from the same warp. The memory access requests from the *LD/ST* are sent to a certain channel based on the warp ID (The warp ID here refers to the dynamic runtime warp ID for a warp when it is executing the kernel. The mapping between a runtime warp ID and the index of a warp in a kernel block can be calculated at runtime when a kernel block is selected to be active) of that request by the *Distributing Unit*.

The reordering happens at two locations in this extension. First, requests from the same warp are reordered in the channel for this warp, because instructions from the same warp can be executed out-of-order. For example, there can be two load/store instructions *I0* and *I1* from the same warp. The memory request from *I1* can arrive at the channel earlier than *I0*. In such case, the *Reordering Unit* will be reordered from *I0* to the L1 data cache first to regulate the access order. Second, the reordering across warps happens at the interface between the channels and the *Reordering Unit*. For instance, in the PTX code example there are 4 warps in total, which are *B0W0, B0W1*, *B1W0*, and *B1W1* that are mapped to the runtime warp IDs based on Table 1. Then the possible orders of memory requests before reordering and the order after reordering are shown in Fig. 6. It should be noted that the reordering is only applied to the load/store instructions after they are issued and arrive between *LD/ST* and L1 data cache. It does not affect dynamic behavior elsewhere and thus may not affect the overall performance as much as reordering all the instructions such as the pure round-robin scheduling.

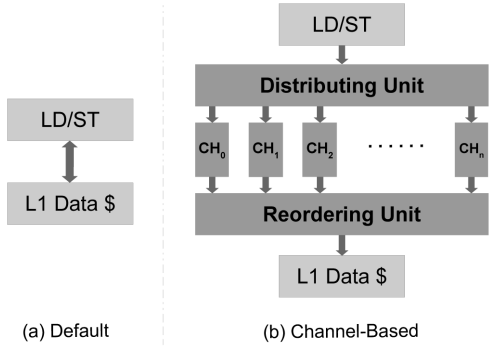


Fig. 5. Warp-based load/store reordering architectural extension.

Table 1. An example of mapping between static block/warp IDs and runtime warp IDs

Static block/warp ID	Runtime warp ID
B0W0 (block 0, warp 0)	W2
B0W1 (block 0, warp 1)	W3
B1W0 (block 1, warp 0)	W0
B1W1 (block 1, warp 1)	W1

The analysis results of the kernel analyzer are sent to the *Reordering Unit* as the initial reordering configuration. The *Reordering Unit* always searches for the warp (channel) with the smallest warp index (in a kernel block rather than runtime warp ID) and the smallest kernel block ID that still has the global load/store instruction with the lowest instruction address. Once the *Reordering Unit* gets a memory request from that channel and sends it to the L1 data cache, it updates the reordering configuration so that it can move on and wait for a different channel or a different instruction at the same channel. The access order to the L1 data cache is decided by the reordering configuration, which comes from the kernel analyzer. This is how the miss rate estimator can have an accurate estimation based on the information provided by the kernel analyzer.

An example is given in Fig. 7 to illustrate how the reordering works. The initial reordering configuration is depicted in Fig. 7(a), based on which the *Reordering Unit* knows it should wait at the channel for *B0W0*, which is CH_2. Even if the requests from other warps are in their channels already, the *Reordering Unit* keeps waiting at CH_2 until it gets the memory request from that warp and then sends it to the L1 data cache. After this the reordering configuration is changed as in Fig. 7(b), based on which

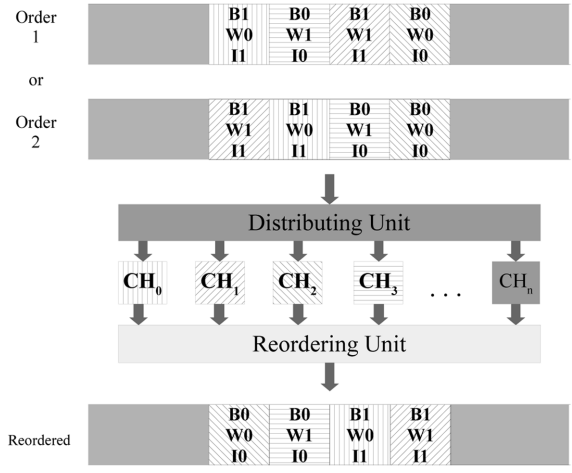


Fig. 6. An example of memory warp instruction reordering.

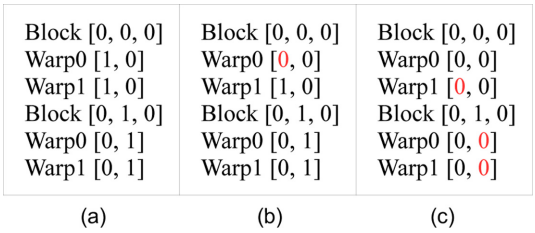


Fig. 7. An example of reorder configurations.

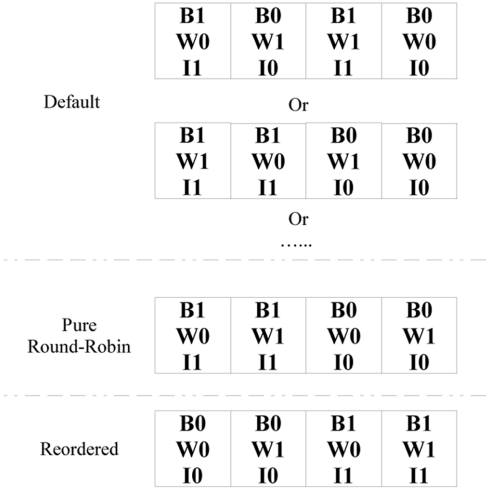


Fig. 8. Access orders of different schemes.

the *Reordering Unit* knows it should wait at CH_3 for *B0W1* now. Eventually, the reordering configuration becomes Fig. 7(c) after the *Reordering Unit* sends all the memory requests to the L1 data cache in the predictable order.

Different sequences of the memory access in the above example are shown in Fig. 8. The access sequence to the L1 data cache can be arbitrary in the default scheme. Under the pure round-robin warp scheduling policy, the order follows the runtime warp ID. With the proposed reordering framework, the sequence of the accesses is controlled by the reordering configuration as explained above. Both the reordering framework and the pure round-robin scheduling policy can improve the predictability in the sequence of GPU L1 data cache accesses, compared to the default system. However, the performance impacts can be quite different.

A performance difference example of four warps is shown in Fig. 9. The time points from Ⓐ to Ⓕ are as shown in the figure. Due to the latency introduced by the reordering extension, the time point for a warp to be ready in the reordering framework may be later than in the other two schemes, e.g., Ⓒ and Ⓕ. The warp schedulers start with *W0*. Under the pure round-robin policy, the next warp is *W1*, which will be ready at Ⓒ. But the dynamic scheduler can choose to schedule other ready warps, instead of waiting for *W1*. So, the dynamic scheduler has better performance.

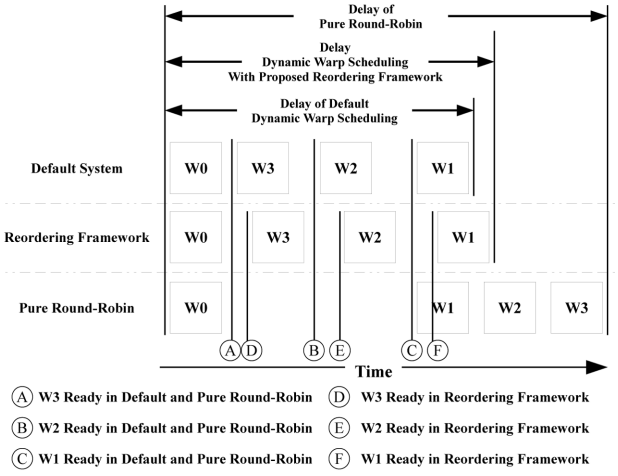


Fig. 9. Delays of different schemes.

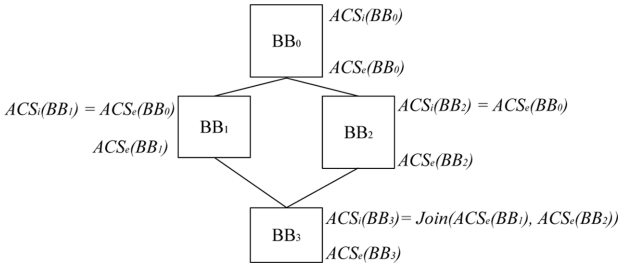


Fig. 10. An example of abstract interpretation based static timing analysis.

F. GPU L1 Data Cache Miss Rate Estimation

For CPUs Abstract Cache State (ACS) is used in the abstract interpretation method [8] to analyze the content and behavior of the cache at a certain point in a program. Fig. 10 illustrates the basic concept of this method; every basic block has initial and exiting stats, namely ACS_i and ACS_e. Upon each memory references in each basic block, the ACS is updated using a specific cache replacing/updating policy, e.g., LRU. A basic block with only one predecessor gets the ACS_e of the predecessor as its ACS_i. When there are more than one predecessors, the ACS_e of each of its predecessor is *Join*ed together to form the ACS_i. Depending on whether the analysis is targeting "always hit" or "always miss", the *Join* operation can be set intersection or set union.

In the CPU cache abstract interpretation, a program only diverges and converges at the boundaries of basic blocks, not in between. Different program traces also do not interfere between each other. However, since warps execute independently in GPUs, these cannot be guaranteed across warps. Putting together the examples in Fig. 3 and Fig. 10, the $ACS_e(BB_0)$ should be only decided by the content of BB_0 in CPU programs. However, in GPUs, when *W0* and *W2* are still in BB_0, *W1* could be in BB_2

already, i.e., the $ACS_e(BB_0)$ is affected by the content of BB_2. Therefore, the CPU cache abstract interpretation method cannot be applied to GPU caches directly, because the boundaries between basic blocks are destroyed by the independent and dynamic execution of different warps. With the proposed reordering framework, the boundaries are restored for global memory requests. For instance, with the reordering framework, only after all the accesses from BB_0 for all the active warps are done, requests from BB_1 or BB_2 can access the L1 data cache. For branches, access from one path, e.g., BB_1, need to be finished before accesses from another path, e.g., BB_2, can access the L1 data cache. With this regulation, the miss rate estimator uses the information of memory access addresses from the kernel analyzer (Section III-D) and the scheduling policy (Section III-E) to generate the sequence of memory access addresses for a kernel. This address sequence is used by the estimator to update the abstract cache models with different configurations to estimate the miss rate. Since the estimator knows how the reordering framework regulates the access order, it can generate a considerably accurate estimation.

1) Limitation of the GPU Data Cache Timing Analyzer

Not all types of GPU kernels can be analyzed with the proposed method. Kernels with input-dependent data references, i.e., whether a load/store instruction will execute depends on statically unpredictable value, currently cannot be analyzed by this cache timing analyzer. Also, the proposed framework requires knowing the loop upper bound statically, which is typical for WCET analysis.

G. Hardware and Performance Overhead of the Reordering Framework

The channels in the architectural extension are memory requests buffers; one channel for one active warp. CPUs store the reordering configuration in the global memory together with the data needed by the GPU kernel. The reordering extension gets the information about the active kernel blocks when they are selected to run. The method of prefetching can be used to hide the latency of fetching the reordering information. Compared with sending requests directly to L1 data cache, reordering can introduce some performance overhead, which is, however, much less than that of using pure round-robin scheduling to regulate the access order, as shown in the evaluation results.

IV. GPU WCET ESTIMATOR

A. Warp Instruction Segment Based WCET Analysis

Based on the dependencies between instructions, the PTX code of a GPU kernel can be divided into segments,

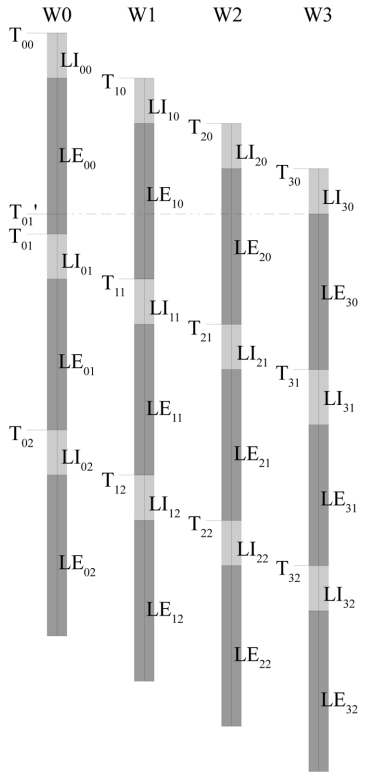


Fig. 11. Example of warp instruction segments.

each of which has one or more instructions. The dependencies between these instruction segments lead to the fact that the instructions in one segment cannot be issued until the instructions in the previous segments, which are the instructions that write to the register used by the instructions in the current segment, have finished their executions and written back the results. Fig. 11 shows an example of the relations of four warps (*W0* to *W3)*, each of which has three instruction segments. T_{ij} *represents* the time point when the GPU can start to issue the instruction segment *j* of warp *i*. LI_{ij} stands for the latency of issuing the instruction segment *j* of warp *i*, while LE_{ij} *represents* the latency of executing the same instruction segment. After initializing the starting issuing time point of each warp by Eq. (1), the rest of the time points in this example can be calculated using Eq. (2), which basically means that the time point when one instruction segment in a warp can start to issue depends on the maximal latency between the latency of the previous instruction segment in the same warp and the latency of issuing the segments in other warps before the scheduler gets back to this warp. The time point for the second segment in *W0* to start to issue could be T'_{01} in the figure, if LE_{00} is less than $LI_{10} + LI_{20} + LI_{30}$.

$$\begin{aligned} &T_{00} \Leftarrow 0 \\ &T_{i0} \Leftarrow T_{i'0} + LI_{i'0} (i > 0) \\ &i' = (i-1) \end{aligned} \tag{1}$$

$$T_{ij} \Leftarrow MAX(T_{i'j}+LI_{i'j},\ T_{ij'}+LI_{ij'}+LE_{ij'})$$
$$i' = (i==0)\ ?\ (N-1) : (i-1)$$
$$j' = j-1$$
$$N : Number\ of\ Wraps \quad (2)$$

B. Latency Calculation

To use the equations in Section IV-A to estimate the WCET, the latencies of issuing and executing each type of instructions should be available. The latency of issuing instructions in a segment is generally the number of instructions in that segment, while the executing latency of a segment depends on the instruction with the longest latency in the segment. The executing latency of arithmetic instructions is usually decided by the latency of the related function units. But, even without considering the L1 data cache, the latency of global memory instructions involves several parts, as shown in Eq. (3). L_{base} *is* the baseline latency of accessing the global memory. $COAL_{max}$ represents the maximal number of coalesced memory requests from one warp instruction. $N_{ConflictSM}$ is the number of SMs that can send requests to the global memory at the same time (Since the major focus of this paper is on the reordering framework rather than the WCET estimator, only the meanings of the components of the equations are given here without detailed descriptions).

When the existence of the L1 data cache is considered, the latency is calculated as in Eq. (4), in which MR_{L1D} *is* the estimated L1 data cache miss rate. As shown in the experiment results, a tighter WCET estimation can be achieved by taking the estimated L1 data cache miss rate into consideration.

$$L = COAL_{max} + (L_{base} + COAL_{max} \times N_{ConflictSM}) \quad (3)$$

$$L = COAL_{max} + MR_{L1D} \times (L_{base} + COAL_{max} \times N_{ConflictSM}) \quad (4)$$

V. EVALUATION METHODOLOGY AND EXPERIMENTAL RESULTS

A. Evaluation Methodology

We use the GPGPU-Sim [10] simulator to implement and evaluate the proposed reordering framework. The configuration of the simulator is shown in Table 2. The default dynamic warp scheduling policy is the loose round-robin policy. The cache configurations have three different sizes with the associativity from 4-way to 8-way and 16-way and the least recently used (LRU) replacing policy. The benchmarks are chosen from the Rodinia benchmark suite [11].

Table 2. GPGPU-Sim configuration

Number of SMs	15
Number of 32-bit registers per SM	32768
Size of L1 data cache per SM	16/32/64 kB
Size of shared memory per SM	48 kB
L1 data cache block size	128 B
L1 data cache replacing policy	LRU
L1 data cache associativity	4/8/16
L2 cache	256 kB
Default warp scheduling policy	Loose round-robin
Optional warp scheduling policy	Pure round-robin
Number of reordering channels	48
Length of a reordering channel	10

B. Experimental Results

1) Performance Results

The normalized performance results in execution cycles are shown in Fig. 12. The performance results of three configurations with a 16-kB L1 data cache are normalized to the performance with the default configuration. The dynamic loose round-robin warp scheduling policy is used in both the default and reordering configurations. This dynamic policy searches for ready warps based on the order of warp IDs, while the pure round-robin policy does not move to the next warp until one instruction is issued for the current warp. Results reveal that the default configuration has the best average-case performance, while the reordering configuration has much less performance overhead compared to the pure round-robin configuration, 24.4% better on average.

The performance results, which are normalized to the performance results of the default configuration with 16-kB L1 data cache, with different L1 data cache sizes with three configurations are shown in Fig. 13. With different cache sizes, the reordering framework still has much less

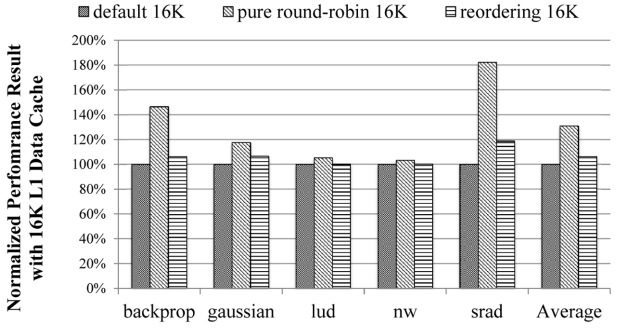


Fig. 12. Normalized performance results with 16 kB L1 data cache.

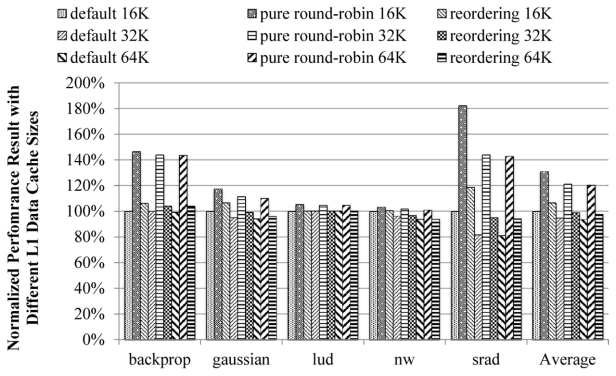


Fig. 13. Normalized performance results with different L1 data cache sizes.

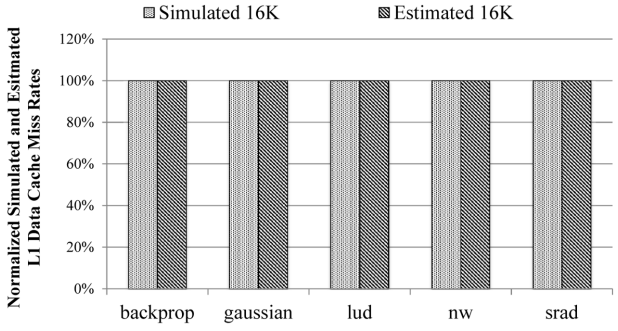


Fig. 14. Miss rate estimation results with 16 kB L1 data cache.

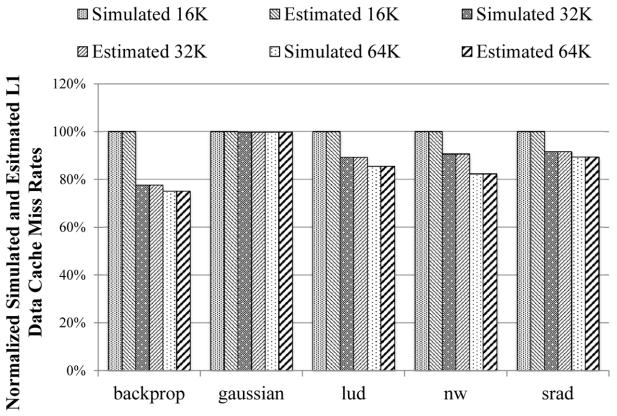


Fig. 15. Miss rate estimation results with different L1 data cache sizes.

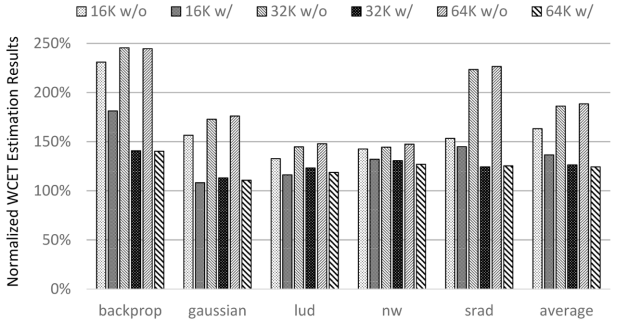


Fig. 16. WCET estimation results without and with cache miss rate information.

performance overhead than the configuration with pure round-robin policy and a small amount overhead compared to the default configuration.

2) Cache Miss Rate Estimation Results

The normalized GPU L1 data cache miss rate estimation results of a 16 kB cache are shown in Fig. 14. The estimated miss rate results are normalized to the simulated miss rates with the reordering configuration. With the support of the reordering extensions and the static analyzer, the miss rate estimation can be very accurate as shown in the figure.

Fig. 15 shows the simulated and estimated GPU L1 data cache miss rate results of three different cache sizes. The results show that the miss rate estimator can still provide accurate miss rate estimations in different cache sizes.

3) WCET Estimation Results

By integrating the estimated L1 data cache miss rate results to the WCET estimator, we get the comparison between the WCET estimations without and with knowing the L1 data cache miss rate estimation, as shown in Fig. 16. When the WCET estimator does not have the L1 data cache miss rate information, it must assume all the global memory requests go to the global memory and take long latency, which results in large overestimation in WCET; more than 60%–80% on average. But having the accurate L1 data cache miss rate estimations, the WCET estimator can achieve much tighter WCET estimations; 20%–40% on average. This is because the estimator, knowing the miss rate estimation, does not have to assume the maximal latency for global memory requests anymore. In the figure, the overestimation of benchmark *backprop* is noticeably larger than the others. This is because there are input-dependent branch behaviors in this benchmark (The input-dependent behaviors in this benchmark do not affect the global memory instructions and thus the reordering framework still works on this benchmark). Due to the branch divergence behavior of GPU warps, the WCET estimator must assume both branch paths will execute, which results in higher overestimation. In the *srad* benchmark, the overestimation increases greatly with 32 kB and 64 kB caches, when the estimator does not know the miss rate estimation. This is because the performance of this benchmark benefits from larger cache sizes, while the WCET estimation is still the same since no cache miss rate information is used. But when the WCET estimator has the miss rate estimation, WCET estimation gets much closer to the actual execution time, as shown in the figure.

VI. RELATED WORK

GPUs have become a major type of accelerator for compute- and/or data-intensive applications, such as building large hash table of millions of elements in real-time efficiently [12] and improve the performance of real-time AES-CBC encryption application [13].

Studies on real-time scheduling for GPUs and heterogeneous processors [14-16] focus on the scheduling algorithms to use them in real-time applications, while assuming the WCET of the real-time tasks is known. This indicates the importance of the time predictability of GPUs.

For cache memory, lots of work have been done to improve their time predictability, e.g., cache locking [17-19]. Alternatives to normal caches are also proposed, such as the scratchpad memory (SPM) [20] and method cache [21], which have better time predictability than normal caches. There are studies on WCET analysis of caches [8, 22-25] as well, which all focus on caches in CPUs rather than GPUs.

In regulating the memory accesses to GPU caches, Xie et al. [26] propose a compiler-based method to bypass the memory accesses with bad localities to improve the performance, while Jia et al. [27] use reordering and bypassing to get more cache-friendly access orders. But both aim at improving the average-case performance rather than the predictability.

Several studies have been done on the WCET analysis of GPU applications [28, 29], which are based on measurement of program segments or the whole application, while the WCET analyzer proposed in this work does not rely on measurement of the GPU program.

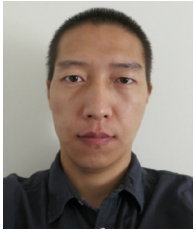
VII. CONCLUSION

The dynamic architectural features of GPUs harm the time predictability a lot and, thus, hinder their application in real-time systems. To address this issue, we propose a framework based on both compiler and architectural extension, which regulates the access order to the GPU L1 data cache with a small performance overhead, even with dynamic scheduling policies, and enables accurate estimations of the miss rate in GPU L1 data cache. Also, by integrating the L1 data cache miss rate estimation in our GPU kernel WCET estimator, tighter WCET estimation can be achieved. In our future research, we would like to extend our GPU cache and WCET analysis to support kernels with irregular memory access patterns and dynamic branches.

REFERENCES

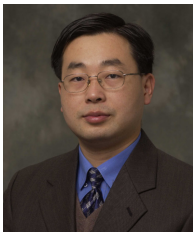
1. S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach, "Accelerating compute-intensive applications with GPUs and FPGAs," in *Proceedings of Symposium on Application Specific Processors (SASP 2008),* Anaheim, CA, 2008, pp. 101-107.
2. J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips, "GPU computing," *Proceedings of the IEEE*, vol. 96, no. 5, pp. 879-899, 2008.
3. J. Shotton, T. Sharp, A. Kipman, A. Fitzgibbon, M. Finocchio, A. Blake, M. Cook, and R. Moore, "Real-time human pose recognition in parts from single depth images," *Communications of the ACM*, vol. 56, no. 1, pp. 116-124, 2013.
4. NVIDIA Tegra mobile processors, http://www.nvidia.com/object/tegra.html.
5. NVIDIA DRIVE PX2, http://www.nvidia.com/object/drive-px.html.
6. NVIDIA CUDA Toolkit Documentation v7.0, https://developer.nvidia.com/cuda-toolkit.
7. J. E. Stone, D. Gohara, and G. Shi, "OpenCL: a parallel programming standard for heterogeneous computing systems," *Computing in Science & Engineering*, vol. 12, no. 3, pp. 66-73, 2010.
8. M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm, "Cache behavior prediction by abstract interpretation," *Static Analysis, Lecture Notes in Computer Science vol. 1145*, Heidelberg: Springer, 1996, pp. 52-66.
9. NVIDIA CUDA Parallel Thread Execution ISA version 4.2, http://www.nvidia.com.
10. A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *Proceedings of IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2009),* Boston, MA, 2009, pp. 163-174.
11. 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 IEEE International Symposium on Workload Characterization (IISWC 2009)*, Austin, TX, 2009, pp. 44-54.
12. D. A. Alcantara, A. Sharf, F. Abbasinejad, S. Sengupta, M. Mitzenmacher, J. D. Owens, and N. Amenta, "Real-time parallel hashing on the GPU," *ACM Transactions on Graphics (TOG)*, vol. 28, no. 5, article no. 154, 2009.
13. U. Verner, A. Schuster, and M. Silberstein, "Processing data streams with hard real-time constraints on heterogeneous systems," in *Proceedings of the International Conference on Supercomputing*, Tucson, AZ, 2011, pp. 120-129.
14. B. Andersson, G. Raravi, and K. Bletsas, "Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors," in *Proceedings of 2010 IEEE 31st Real-Time Systems Symposium (RTSS)*, San Diego, CA, 2010.
15. 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.
16. G. Elliott, B. Ward, and J. Anderson, "Gpusync: architecture-aware management of GPUs for predictable multi-GPU real-time systems," in *Proceedings of 34th IEEE RTSS*, Vancouver, Canada, 2013, pp. 33-44.
17. X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," *ACM SIGMETRICS Performance Evaluation Review*, vol. 31, no. 1, pp. 272-282, 2003.
18. V. Suhendra and T. Mitra, "Exploring locking & partition-

ing for predictable shared caches on multi-cores," in *Proceedings of the 45th annual Design Automation Conference*, Anaheim, CA, 2008, pp. 300-303.
19. H. Ding, Y. Liang, and T. Mitra, "WCET-centric partial instruction cache locking," in *Proceedings of 2012 49th ACM/EDAC/IEEE Design Automation Conference (DAC),* San Francisco, CA, 2012, pp. 412-420.
20. R. Banakar, S. Steinke, B. S. Lee, M. Balakrishnan, and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems," in *Proceedings of the 10th International Symposium on Hardware/Software Codesign*, Estes Park, CO, 2002, pp. 73-78.
21. M. Schoeberl, "A time predictable instruction cache for a Java processor," in *On the Move to Meaningful Internet Systems: OTM 2004 Workshop.* Heidelberg: Springer, 2004, pp. 371-382.
22. D. Hardy and I. Puaut, "WCET analysis of multi-level non-inclusive set-associative instruction caches," in *Proceedings of Real-Time Systems Symposium (RTSS),* Barcelona, Spain, 2008, pp. 456-466.
23. Y. Yan and W. Zhang, "WCET analysis for multi-core processors with shared L2 instruction caches," in *Proceedings of Real-Time and Embedded Technology and Applications Symposium (RTAS'08),* St. Louis, MO, 2008, pp. 80-89.
24. Y. Li, V. Suhendra, Y. Liang, T. Mitra, and A. Roychoudhury, "Timing analysis of concurrent programs running on shared cache multi-cores," in *Proceedings of Real-Time Systems Symposium (RTSS)*, Washington, DC, 2009, pp. 57-67.
25. B. K. Huynh, L. Ju, and A. Roychoudhury, "Scope-aware data cache analysis for WCET estimation," in *Proceedings of 2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS),* Chicago, IL, 2011, pp. 203-212.
26. X. Xie, Y. Liang, G. Sun, and D. Chen, "An efficient compiler framework for cache bypassing on GPUs," in *Proceedings of 2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD),* San Jose, CA, 2013, pp. 516-523.
27. W. Jia, K. A. Shaw, and M. Martonosi, "MRPB: memory request prioritization for massively parallel processors," in *Proceedings of 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA),* Orlando, FL, 2014, pp. 272-283.
28. A. Betts and A. Donaldson, "Estimating the WCET of GPU-accelerated applications using hybrid analysis," in *Proceedings of 2013 25th Euromicro Conference on Real-Time Systems (ECRTS),* Paris, France, 2013, pp. 193-202.
29. K. Berezovskyi, L. Santinelli, K. Bletsas, and E. Tovar, "WCET measurement-based and extreme value theory characterisation of CUDA kernels," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, Versaille, France, 2014, p. 279.



Yijie Huangfu

Yijie Huangfu received his B.S. degree in the major of Automation and M.S. degree of Communication and Information System from the Dalian University of Technology in 2006 and 2009, respectively. He received his Ph.D. degree in Electrical and Computer Engineering from the Virginia Commonwealth University in 2017. He has become an Architecture Engineer at NVIDIA since June 2017.



Wei Zhang

Wei Zhang is a professor in the Department of Electrical and Computer Engineering at Virginia Commonwealth University. Dr. Zhang received his Ph.D. from the Pennsylvania State University in 2003. From August 2003 to July 2010, he worked as an assistant professor and then as an associate professor (tenured) at Southern Illinois University Carbondale (SIUC). His research interests are in embedded and real-time computing systems, computer architecture, compiler, and low-power systems. He has received the 2009 SIUC Excellence through Commitment Outstanding Scholar Award for the College of Engineering, and 2007 IBM Real-time Innovation Award. Dr. Zhang has received 5 research grants from the National Science Foundation. In addition, his research and educational efforts have been supported by industry including leading IT companies such as IBM, Intel, Motorola, and Altera. He has published more than 120 papers in refereed journals and conference proceedings. He is a senior member of the IEEE, and an associate editor of the Journal of Computing Science and Engineering. He has served as a member of the organizing or program committees for several IEEE/ACM international conferences and workshops.