UNIVERSITY OF CALIFORNIA,
IRVINE

Reliability and Timing Aware GPU Management on Embedded Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Haeseung Lee

Dissertation Committee:
Professor Mohammad Al Faruque, Chair
Professor Fadi Kurdahi
Professor Pai Chou

2017

DEDICATION

To my wife and family
for their unconditional love and support

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ALGORITHMS

LIST OF ABBREVIATIONS

ADAS Advanced Driving Assistance Systems

APIs Application Programming Interfaces

AVF Architectural Vulnerability Factor

CMPs Chip Multi-Processors

CR Capacity Rate

CUDA Compute Unified Device Architecture

DMR Dual-Modular Redundancy

DVFS Dynamic Voltage and Frequency Scaling

GPU Graphics Processing Unit

HCI Hot Carrier Injection

ILP Instruction Level Parallelism

ILV Instruction-Level Vulnerability

IVI Instruction Vulnerability Index

JIT Just-In-Time

JLSP Job-Level Static-Priority

MEBF Mean Executions Between Failures

MPSoC Multiprocessor System-on-a-Chip

NBTI Negative Bias Temperature Instability

NoC Network-on-Chip

PSO Particle Swarm Optimization

QoS Quality of Service

RMT Redundant Multithreading

RTL Register Transfer Level

SFU Special Functional Unit

SM Streaming Multiprocessor

SMT Simultaneous Multithreading

SP Stream Processor

ACKNOWLEDGMENTS

It has been a long and wonderful journey to pursue the doctorate degree during the past five years, and life would have been much more difficult without the support from my professors, family, colleagues and friends. Thus, I would like to take this opportunity to thank these great people for encouraging and inspiring me in this journey.

First and foremost, I would like to gratefully and sincerely thank my advisor, Professor Mohammad A. Al Faruque, for his guidance, motivation, understanding and patience during my graduate study at UC Irvine. It has been a great learning experience to be a member of his research group and a teaching assistant of his class. His considerable insights, kind personality and sense of humor created an enjoyable and pleasant working environment in our group. Thanks to this, I have learned much more from our meetings and conversations in the past few years, not only on researching and teaching, but also on life and philosophy. This is a treasure that will benefit me for a whole lifetime.

Next, I would like to thank Professor Fadi Kurdahi and Pai H. Chou, for taking time out of their busy schedules to serve as my committee members and provide me constructive suggestions and feedback on my dissertation.

Many thanks to my colleagues in the Advanced Integrated Cyber-Physical Systems (AICPS) group, including Jiang Wan, Korosh Vatanparvar, Sujit Rokka Chhetri, Sina Faezi, Anthony B. Lopez, and Xiaohong Li, for the fruitful talks, discussions and teamwork. I appreciate all their help during my graduate program.

I would like to express my appreciation to my parents, Eunyoung Lee and Hunam Jeon, for their unconditional love and endless faith in me. Their support is the very source of power and morale for me to pursue my dream.

Also, I can never forget my parents-in-law Seohyang Kim, who I am always grateful for her generous inspiration and their firm believe in me.

Last but not least, I would like to express my deepest gratitude to my family. My best friend and wife Jiyoun Kim, her encouragement, dedication and love during my PhD years was in the end what made this dissertation possible. John Seungju Lee, my baby boy, who has been the light of my life for the last two years and who has given me the extra strength and motivation to get things done.

CURRICULUM VITAE

Haeseung Lee

EDUCATION

Doctor of Philosophy in Computer Engineering **2017**
University of California, Irvine *Irvine, California*

Master of Science in Electrical Engineering **2013**
Arizona State University *Tempe, Arizona*

Master of Science in Computer Engineering **2009**
Chung-Ang University *Seoul, South Korea*

Bachelor of Engineering in Electrical and Electronics Engineering **2004**
Chung-Ang University *Seoul, South Korea*

RESEARCH EXPERIENCE

Graduate Research Assistant **2013–2017**
University of California, Irvine *Irvine, California*

Graduate Research Assistant **2011–2013**
Arizona State University *Tempe, Arizona*

TEACHING EXPERIENCE

Teaching Assistant **2014–2017**
University of California, Irvine *Irvine, California*

REFEREED JOURNAL PUBLICATIONS

Run-time Scheduling Framework for Event-driven Applications on a GPU-based Embedded System **2016**
IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), vol. 35, no. 12, pp. 1956-1967, March 2016.

GPU Architecture Aware Instruction Scheduling for Improving Soft-error Reliability **2017**
IEEE Transactions on Multi-Scale Computing Systems (TMSCS), vol. 3, no. 2, pp. 86-99, February 2017.

Aging-aware Workload Management on Embedded GPU Under Process Variation **2017**
IEEE Transactions on Computers (TC) (Under submission)

REFEREED CONFERENCE PUBLICATIONS

GPU-EvR: Run-time Event Based Real-time Scheduling Framework on GPGPU Platform **Apr 2014**
IEEE/ACM Design Automation and Test in Europe (DATE), Dresden, 2014, pp. 1-6.

PAIS: Parallelization Aware Instruction Scheduling for Improving Soft-error Reliability of GPU-based Systems **Mar 2016**
IEEE/ACM Design Automation and Test in Europe (DATE), Dresden, 2016, pp. 1568-1573.

Low-overhead Aging-aware Resource Management on Embedded GPUs **Jun 2017**
ACM/IEEE Design Automation Conference (DAC), Austin, TX, USA, 2017, pp. 1-6.

ABSTRACT OF THE DISSERTATION

Reliability and Timing Aware GPU Management on Embedded Systems

By

Haeseung Lee

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2017

Professor Mohammad Al Faruque, Chair

The demand for low-power and high-performance computing has been driving the semiconductor industry for decades. In order to satisfy these demands, the semiconductor technology has been scaled down and multi/many-core processors have been proposed. Among the multi/many-core processors, Graphics Processing Units (GPUs) have been employed in the critical path of applications due to its programmability, high-performance, and low power consumption. Moreover, state-of-the-art GPUs have the capability to process multiple GPU workloads concurrently. Therefore, GPUs have been considered to be an essential part of embedded systems because of the increased number of throughput-oriented applications on real-time embedded systems, such as autonomous driving and advanced driving assistant applications. However, there are several challenges for using the GPUs in embedded systems. First, due to the small feature size, the state-of-the-art nano-scale multi-core processors, including GPUs, has faced severe reliability challenges like soft-error and processor degradation. Next, there is a noticeable (die-to-die and within-die) parameter variation due to the advanced semiconductor technology. Therefore, the lifetime and workload management of embedded GPUs under process variation is considered one of the most important aspects to ensure functional correctness over a long period of time. Last, existing application scheduling frameworks on a GPU do not have enough flexibility to handle the dynamic behavior of multiple event-driven applications.

In order to tackle the above mentioned challenges, in this thesis, we propose a reliability and timing aware workload management framework on GPU-based real-time embedded systems. The proposed framework consists of two parts: design-time and run-time workload management. The proposed design-time workload management unit analyzes GPU kernel functions and generates PTX instruction schedules that maximizes the soft-error reliability. At the same time, the application profiles are generated for run-time workload management. The proposed run-time workload management unit includes two parts: Streaming Multiprocessor (SM) scheduling unit and aging-aware workload distribution unit. During run-time, depending on the system status and requirements, the proposed scheduling unit partitions the GPU workloads into sub-workload and generates sub-workloads launch sequences to handle the dynamic behavior of the event-driven applications. Concurrently, in the SM, the proposed aging-aware workload distribution unit jointly considers the current aging status and the process variation status and distributes the workload across the SM to maximize the lifetime of the GPU.

Chapter 1

Introduction

1.1 Introduction

In recent years, embedded systems demand higher computational power, and need to respond to many random external and internal events. The number of throughput-oriented applications in the embedded systems keeps increasing [29, 34, 53, 76, 79, 106, 136, 142]. To satisfy these demands for low-power and high-performance computing, semiconductor industries have done extensive technology scaling and have proposed various multi/many-core processor architectures. Among these multi/many-core processors, Graphics Processing Units (GPUs) have been considered and employed in the critical path of applications in embedded systems due to their programmability, high-performance, and low power consumption [11, 13, 43, 68, 76, 78, 85, 105, 112, 118, 146]. In addition to the previously mentioned advantages, CPUs and GPUs, in GPU based embedded systems, share the same memory which results in a near-zero data transfer latency [76].

Figure 1.1 shows an example of a real-world GPU-based embedded system. The figure provides the abstract diagram of an *intelligent driving assistance system*, which is similar

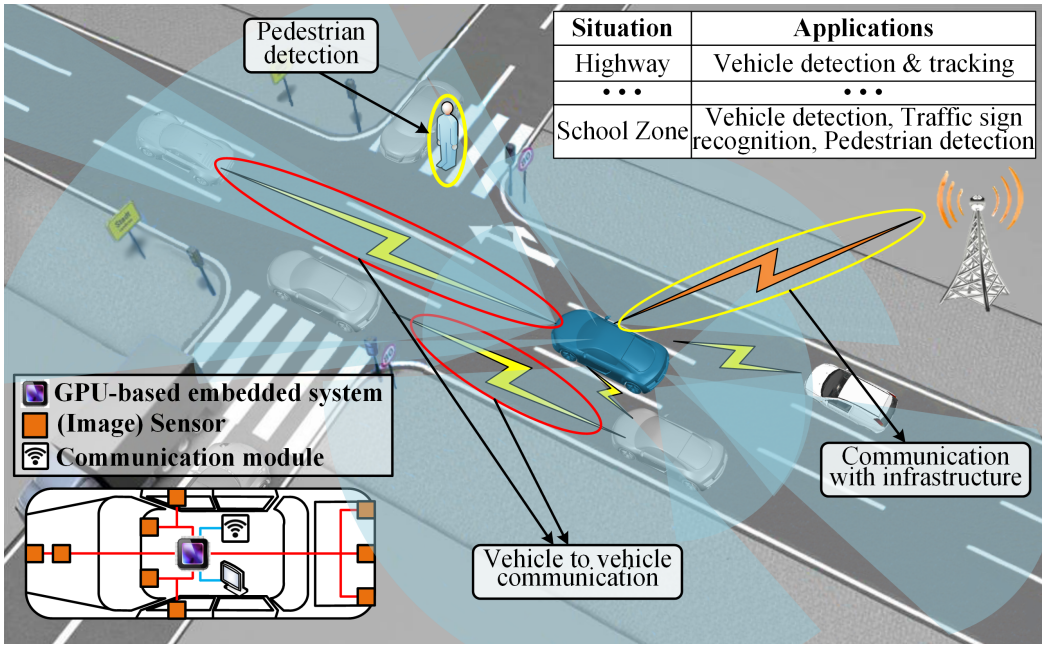


Figure 1.1: Intelligent Driving Assistance System Similar to Audi A7's Auto Pilot System Utilizing GPU-based Embedded System [95].

to the Audi A7's *auto pilot system*, on a GPU-based embedded system (a Tegra mobile embedded system from Nvidia) [95]. In Figure 1.1, each box represents one (image) sensor and the shaded area represents the coverage of each (image) sensor. These sensors monitor the vehicle's surroundings. Each sensor detects different objects, for example, sensors near the windshield read traffic signs and sensors in the front detect pedestrians. Meanwhile, the vehicle is connected to the vehicular network (e.g. vehicle to vehicle communication network) through various communication modules.

While driving, a mixture of safety critical and non-safety applications is launched by the system depending on the current system status. For example, when driving on a highway, the system would launch a set of safety critical and high priority applications for *vehicle detection*. On the other hand, when driving in a school area, the system would launch a different set of safety critical and high priority applications for *pedestrian detection* and *traffic sign recognition*. At the same time, non-safety critical and low priority applications may be running on the system.

However, there are several challenges for using the GPU-based embedded systems. First, in embedded systems, each application would have a different priority and deadline depending on the current system status. Note that in this type of embedded system, small timing violations may cause degradation in *Quality of Service* (QoS), such as glitches in an instance of a traffic sign image. Therefore, the system is a soft real-time system and the QoS of the system highly depends on the status of safety critical and high priority applications. However, existing application scheduling frameworks on a GPU do not have enough flexibility to handle the dynamic behavior of the event-driven applications. This is because in the existing scheduling frameworks: 1) only temporal preemption is considered and 2) one application occupies the GPU at a time.

Second, due to the small feature size, the state-of-the-art nano-scale multi-core processors, including GPUs, have faced several reliability challenges such as soft-error [16, 19, 38, 40, 74, 120, 129, 130], aging effects [14, 19, 20, 37, 61, 84, 102, 137], and (die-to-die and with-in-die) parameter variation [1, 4, 57, 64, 101, 110, 119, 132]. The real-world GPUs are susceptible to the soft-errors even under normal conditions [24, 40, 90, 98]. The probability that the soft-error occurs on a single hardware component is proportional to the time that the hardware component is used [143]. The soft-error reliability of the GPU is important because the GPU-based system handles most of its computation by using the GPU. In other words, if the GPU produces incorrect results, the entire system may behave incorrectly. In order to improve the soft-error reliability of the GPU, many methodologies have been proposed [32, 55, 73, 135, 139]. However, instruction scheduling has not been considered for improving the soft-error reliability of the GPU. Negative Bias Temperature Instability (NBTI) and Hot Carrier Injection (HCI) are considered among the most critical aging-related reliability challenges in nano-scaled semiconductors [14, 19, 20, 37, 61, 84, 102, 137]. The amount of transistor degradation caused by NBTI and HCI is proportional to the time a transistor is stressed or switched [19, 75, 88, 103]. Various workload management techniques have been proposed to balance the stress level across the chip to minimize NBTI and HCI effect. To alleviate

the effects of process variation, typically, multi-core processors have employed a chip-level guardbanding technique, which operates at the lowest frequency. However, the state-of-the-art multi-core processors uses a core-level guardbanding technique, which applies different frequencies for each core, to further improve the overall performance. Due to the core-level guardbanding technique, each core of a single-chip would have different operating frequency and duty cycle, which leads to varying stress levels and thermal variations across the chip. These stress/thermal variations likely introduce randomness in the system state. The existing workload management techniques have a limitation in minimizing the aging effect with the core-level guardbanding technique. Especially, with the core-level guardbanding technique, the GPU randomly distributes the instructions to its cores. This is due to the asynchronous behavior of the components in the GPU.

In summary, the state-of-the-art GPU-based embedded systems suffer from the following challenges:

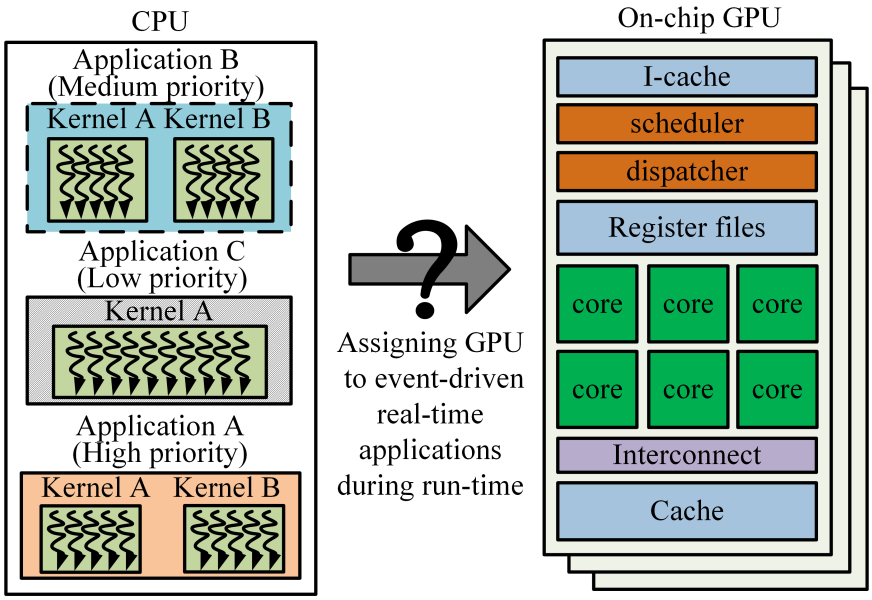
- Existing application scheduling frameworks on a GPU do not have enough flexibility to handle the dynamic behavior of the event-driven applications.
- Due to the small feature size, the semiconductor technology has faced severe reliability challenges like soft-error and processor degradation. The soft-error has been improved by using various methodologies such as redundancy methodologies. However, the GPU compiler has yet to be considered for improving the soft-error reliability of the GPU.
- A noticeable within-die parameter variation and core-level guardbanding technique increase the randomness of the system state. Therefore, existing aging management techniques have a limitation in maximizing the lifetime of embedded GPUs in the presence of the parameter variation.

1.2 Motivational Case Studies

In order to highlight the above mentioned challenges, we perform several case studies on real-world applications. Figure 1.2 illustrates a scenario where multiple applications, with different priorities, are executed on the target GPU-based embedded system. Figure 1.2a describes the abstract behavior of the target system while Figure 1.2b provides the information about the input applications. Since the current GPU processes the workloads sequentially, the response time of each application heavily depends on the arrival time. In our example, the high priority application A starts executing after the lower priority applications B and C have completed. Note that application A does not meet its deadline. However, application A could meet its deadline if the target GPU-based embedded system may be capable of generating schedules during run-time, in addition to supporting preemption.

Note that after the host launches a kernel function, only the GPU can control the kernel behavior [92]. In other words, due to the limitation of the GPU hardware, the GPU does not allow the reallocation of the GPU resources for ongoing kernels. Therefore, the host can either wait for the completion of the kernel, or alternatively, launch another kernel. However, typically, a kernel is a collection of thousands of identical GPU threads and the host partitions a kernel based on the configuration. After submitting an entire kernel to the GPU, it is possible that some part of the kernel is running on the GPU and the other part of the kernel is waiting on the GPU hardware queue. Therefore, when partitioning a GPU kernel in the host, an opportunity arises to reallocate the GPU resources and implement both temporal and spatial preemption on a GPU-based embedded system.

In order to see the feasibility of multiple sub-kernel launches, we measured the average performance overhead for an empty kernel launch while using two different GPUs (Nvidia GTX660 and Tegra K1). Figure 1.3a shows the source code while Figure 1.3b shows the corresponding observational results. According to Figure 1.3b, the average empty kernel



(a) Target GPU-based Embedded System.

App	Prio.	Arr. time	deadline	Exec. time	Cmpl. time
A	High	2.5ms	4.2ms	1.5ms	6.5ms
B	Mid	1ms	3.5ms	2ms	3ms
C	Low	2ms	6.5ms	2ms	5ms

(b) Input Event-driven Application Information.

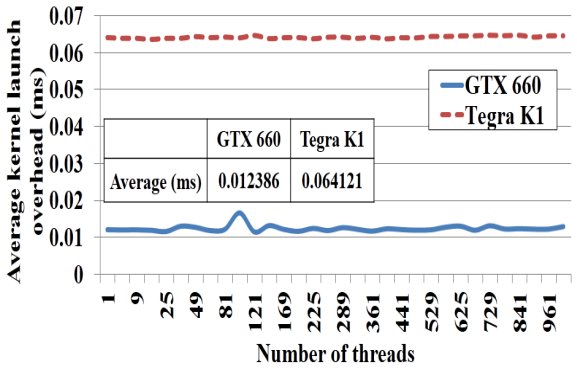
Figure 1.2: Motivational Example Scenario on the GPU-based Embedded System.

launch overheads for GTX660 and Tegra K1 are $12\mu s$ and $64\mu s$, respectively.

The example scenario shown in Figure 1.2 indicates that the GPU needs to be efficiently assigned and provide preemption capabilities, in order to meet the deadlines of the applications. Moreover, the results in Figure 1.3 imply that the performance overhead of launching multiple sub-kernels may be relatively small compared to the execution time, of the kernel, within a single launch. The following capabilities are what make our groundbreaking approach possible; 1) partitioning application kernels into multiple sub-kernels in order to represent the GPU resources (re)allocation and 2) launching the sub-kernels with the objective of assigning more GPU cores to higher priority applications in order to apply the GPU resources (re)allocation.

```
1 __global__ void empty_kernel(input) {;}
2
3 int main()
4 {
5   ....
6   // Time measure start
7   ....
8   for (int i=0; i<repeat; i++) {
9     // launch empty kernel
10    empty_kernel<<<grid, block>>>(input);
11    // wait until empty kernel finish
12    cudaDeviceSynchronize();
13  }
14  // Time measure end
15  ....
16  // Take the average and cleanup
17  ....
18 }
```

(a) Part of Source Code for Measuring Kernel Launch Overhead.



(b) Instruction Distribution with Process Variation.

Figure 1.3: Kernel Launch Overhead for Varying Number of Threads.

In order to observe how the instruction scheduling affects the soft-error reliability, we have created a total of three matrix multiplications by modifying its instruction schedule. The vulnerable period is the metric to measure the soft-error reliability of a GPU application [55, 86] (see Section 3.2 for more details). The vulnerable period is the time from the moment that the data is produced until the last moment that the data is consumed [86]. We then measured the vulnerable period of these three matrix multiplication applications by using the GPGPU-Sim [8] simulator and show the experimental results in Figure 1.4. Below we provide some observations of these experimental results.

The total amount of the vulnerable period is not proportional to the number of threads in a thread block. The vulnerable periods vary depending on the number of threads in a thread block.

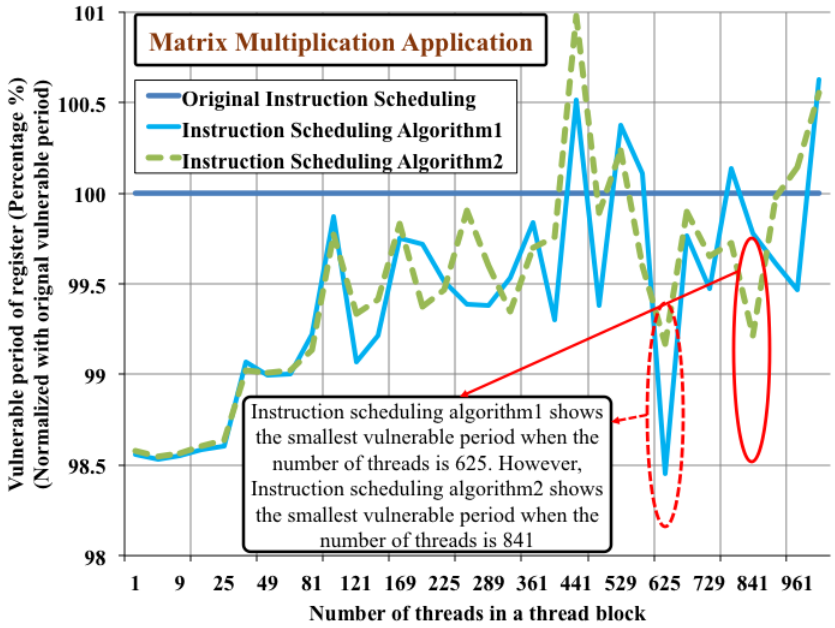
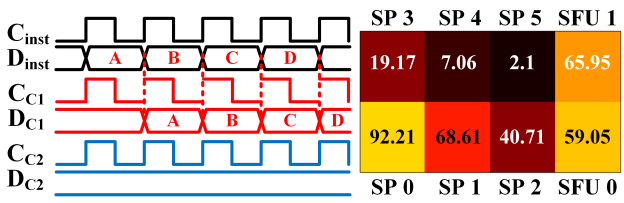


Figure 1.4: Motivational Example to Illustrate the Relation Between Vulnerable Period and Instruction Schedule.

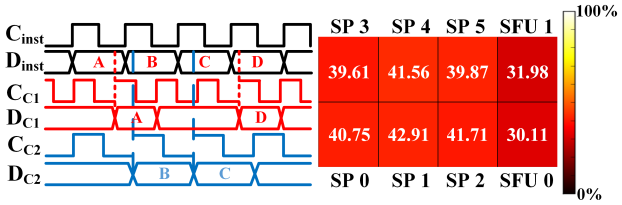
In addition, there is no instruction schedule that always shows the minimum vulnerable period. For example, *Scheduling Algorithm 1* shows the smallest vulnerable period when the number of threads is 625. However, when the number of threads is 841, *Scheduling Algorithm 2* shows the smallest vulnerable period.

These experimental results indicate that the vulnerable period of an application does not only depend on the instruction scheduling but also on the parallel behavior of the GPU. The result may show a small amount of change in the vulnerable period. This is because of the simple kernel function and a small number of threads. However, the result in Figure 1 still shows the possibility that the vulnerable period, which is related to the soft-error reliability, may be improved through the instruction scheduling. Therefore, the parallel behavior of the GPU and the instruction scheduling need to be considered together to further improve the soft-error reliability.

Figure 1.5 provides example case studies to show how the process variation and core-level guardbanding affect the workload distribution on the GPU. Figure 1.5(a) describes the sim-



(a) Instruction Distribution without Process Variation.

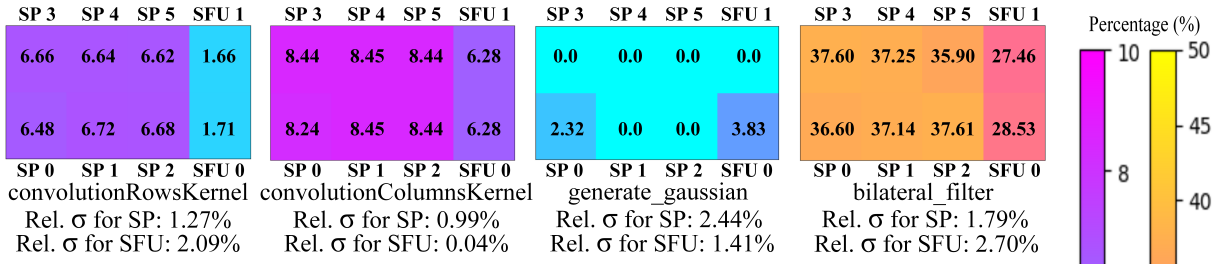


(b) Instruction Distribution with Process Variation.

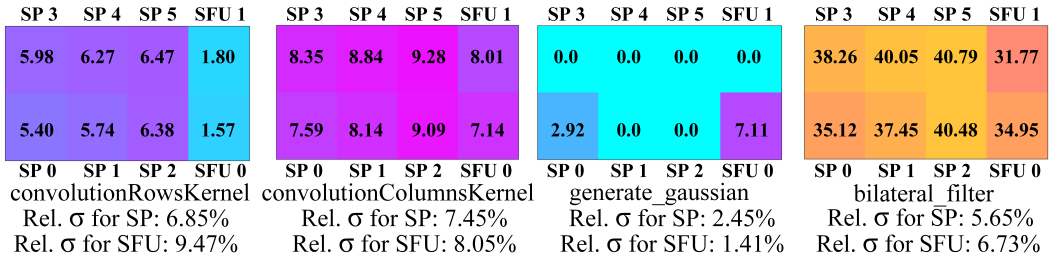
Figure 1.5: Example Behavior of the Existing Warp Scheduler and Instruction Dispatcher with/without Process Variation.

plified behavior of the instruction dispatcher and two clusters ($C1$ and $C2$) without the process variation and core-level guardbanding. Without the core-level guardbanding, all the clock signals (C_{inst}, C_{C1}, and C_{C2}) are synchronized and four instructions (A, B, C, and D) are sent to the cluster 1, $C1$. The cluster 2, $C2$, will get the instructions after the cluster 1's pipeline is full. However, with the core-level guardbanding, the instruction dispatcher and the clusters show different behavior. As described in Figure 1.5(b), each component has a different operating frequency due to the core-level guardbanding. Besides, because of the asynchronous behavior, each cluster gets two instructions. The instructions A and D are assigned to the $C1$ and the instructions B and C are assigned to the $C2$. The example in Figure 1.5 implies that the process variation and the core-level guardbanding cause some degree of workload distribution. However, as shown in Figure 1.5(b), the workload is not evenly distributed because $C1$ and $C2$ have different operating frequencies.

In order to demonstrate the aforementioned unbalanced workload distribution with the process variation and core-level guardbanding, we perform experiments with real world applications. In state-of-the-art embedded GPUs, such as NVIDIA's Tegra TK1, multiple instructions are issued in a single clock cycle. Each instruction represents the behavior of 32 threads which is handled by the cluster of 32 cores and called *warp*. This *warp* is the basic execution unit of NVIDIA's GPU. However, due to the process variation and cluster-level guard banding, the stress is not evenly distributed across the GPU even though each cluster processes the same number of instructions. We use the GPGPU-Sim simulator [8],



(a) Stress Distribution Map Results for Multiple Kernels on the Embedded GPU without Process Variation



(b) Stress Distribution Map Results for Multiple Kernels on the Embedded GPU with Process Variation

Figure 1.6: Simulation Results to Show the Effect of the Process Variation on Embedded GPUs.

which is a configurable cycle-level GPU architectural simulator. We configured the GPGPU-Sim to have a similar configuration with the GPU in NVIDIA's Tegra TK1 (1 Streaming Multiprocessor, 192 CUDA cores, etc.) and executed the following benchmark applications: bilateral filter and 2D convolution. In GPGPU-Sim, each Stream Processor (SP) and Special Functional Unit (SFU) handles the execution of a single *warp*. The SP and SFU units are mapped into a process variation map that is generated in [38, 120].

During the experiments, we evenly distributed the instructions to the SP and SFU units. Then, the pipeline status of the SP and the SFU units are collected to generate a stress (workload) distribution map for each benchmark application with/without the process variation. Figure 1.6 shows experimental results for various GPU kernel functions. Figure 1.6(a) shows the stress map results when all the SP and SFU units operate with the same frequency. The stress map and the standard deviation imply that the stress is evenly distributed across the GPU. On the other hand, after considering the process variation and cluster-level guard-

bandinf technique, the stress is not evenly distributed. Figure 1.6(b) shows the stress map and standard deviation results with the process variation. The stress map shows that each SP and SFU unit has a different amount of stress while processing the same number of instructions. In addition, the standard deviation of stress for the SP and SFU units is increased by 4.3 and 52.3 times in average, respectively. The results indicate that that current warp schedulers and instruction dispatchers have a limitation in minimizing the NBTI and HCI effects under the process variation. Therefore, there is a need for a fine-grained workload management technique that reconfigures the cluster of cores and distributes the workload to minimize the NBTI and HCI effects on the embedded GPUs under the process variation.

1.3 Contributions

1.3.1 Problem and Research Challenges

The above mentioned problems for GPU-based embedded systems poses the following *research challenges*:

- How to partition GPU kernels into multiple sub-kernels during run-time, such that multiple application kernels could concurrently occupy the GPU, where the number of the applications that meet their deadline is maximized.
- Estimation of the soft-error of GPU applications by considering the accurate GPU execution model and Generation of the instruction schedule in order to maximize the soft-error reliability of a GPU application during design-time.
- Due to the variation in degradation and core-level guard banding, the process variation aware cluster (warp) formation is required. In addition, the warp scheduler and the

instruction dispatcher should balance the workload distribution across the GPU by considering the degradation levels and process variation together.

1.3.2 Our Novel Contributions

In order to address the above mentioned challenges, we propose a reliability and timing aware workload management framework which includes:

- A **GPU architecture-aware instruction scheduling algorithm** (Section 3) that maximizes the soft-error reliability of a GPU application during design-time by considering the impact of the parallel behavior of the GPU.
- A **timing-aware workload scheduling framework** (Section 5) which partitions the GPU kernel into multiple sub-kernels during run-time in order to implement spatial preemption. Depending on the current status of the target GPU-based embedded system and the application priority, the proposed workload splitter decides the number of sub-kernels and the size of each sub-kernel.
- A **aging-aware workload distribution unit** (Section 4) that generates core cluster information based on the current NBTI and HCI information during run-time. After that, the proposed workload distribution unit is configured based on the above mentioned information. The instructions are distributed based on the instruction distribution ratio to minimize the aging effects under the process variation.

The rest of this dissertation is organized as follows: Chapter 2 discusses related works. Chapter 3 explains the design-time part of the proposed framework. Chapter 5 and 4 discuss the run-time part of the proposed framework. Chapter 5 provides a discussion on the proposed flexible run-time workload scheduling. Chapter 4 gives an explanation for the proposed

aging-aware workload distribution under the process variation. Finally, Chapter 6 concludes the dissertation.

Chapter 2

Related Works

A large body of research aims to maximize the lifetime of embedded GPUs while increasing the flexibility on workload scheduling. Research has been conducted to improve the soft-error reliability of the processors including GPUs [32, 39, 54, 73, 111, 121, 128, 135, 139]. Some research works have shown the impact of soft-errors by using radioactive sources [97, 133]. However, usage of these radioactive sources is difficult and not for the general purpose applications, therefore some research works have focused on the technique to model the soft-error behavior [6, 49, 62] and evaluate the soft-error resiliency of an application [31, 45, 86, 97, 122, 133]. Research works have proposed to detect the occurrence of the soft-errors and ensure the correctness of an application by using various techniques (i.e. redundant execution [54, 121, 128, 135, 139], insertion of protection code [32, 73], and leveraging architectural characteristics [39, 111]). In the rest of this section, we discuss in detail the above mentioned research works.

Various research works have been conducted to demonstrate the impact of soft-error and evaluate the soft-error resilience of the GPU application [31, 45, 86, 97, 122, 133]. One research [86] proposed a metric to quantify the soft-error reliability based on the detailed

timing behavior of an application. Other research works [97] and [133] showed the impact of the radiation-induced soft-error on the NVIDIA's GPU. In order to inject the actual soft-error into the GPU, the target system is exposed to the neutron beam. Another research in [117] has evaluated the soft-error resilience of several safety-critical applications. An embedded GPGPU platform has been exposed to neutron flux in order to measure the soft-error resilience. A soft-error estimation framework is proposed in order to accurately estimate the soft-error rate, work in [45]. Unlike the traditional netlist-based technique, the proposed framework estimated the soft-error rate from the layout of the target processor. The impact of soft-error reliability has been discussed in [31] and [122]. Various techniques (i.e. debugger based fault injection) are used to show the impact of soft-error reliability of the GPU. These research works have successfully demonstrated the impact of soft-error in real-world environment and applications. However, these works do not propose the technique to improve the soft-error reliability.

Research has been conducted to model the soft-error behavior. One research in [6] has proposed a soft-error model to provide the failure probability of various interleaving techniques for the SRAM. Other research in [49] has discussed about the soft-error model for 25nm technology. Another research in [62] has proposed a cross layer analysis approach for modeling the soft-error behavior on the FinFET transistors. The proposed approach performs 3D simulations from the interactions in FinFET structures up to circuit level. A soft-error susceptibility estimation technique has been proposed in [83]. The proposed technique uses symbolic modeling to estimate the soft-error susceptibility of a combinational logic circuit. However, since the detailed GPU architecture is not available to the research community, the research works are not applicable.

Research has been conducted to provide the soft-error injection tools because it is extremely difficult to perform the experiment in a radiation environment [30, 72]. One research in [72] has proposed a soft-error injection tool that injects a single bit-flip into the data object in the

binary. Other research in [30] has proposed a soft-error injection technique that randomly selects the instruction and changes its result. However, these research works have limitations in modeling the random behavior of soft-error. For instance, although not all the injected soft-errors cause bit-flips, these research works inject a bit-flip whenever the soft-error occurs.

Research has been conducted in order to find the incorrect behavior and ensure the functional correctness [54, 77, 121, 128, 135, 139]. One research in [77] has proposed an application framework to handle the soft-errors on GPU DRAM. The DRAM errors in GPUs are detected by using the dual parity technique and the application is recovered from the checkpoints. Other research in [139] has proposed a software level Dual-Modular Redundancy (DMR) technique. Each stage has a monitor function and its result is compared to the result from that monitor function. Another research in [54] has proposed a redundant technique that utilizes the GPU idle time caused by the branch divergence. A duplication technique proposed in [121] has redundant execution for the critical parts of the GPU pipeline and recomputes erroneous results when error is detected. A redundant execution methodology is proposed in [128] and it uses the idle time of the GPU in order to minimize the performance overhead. An automatic Redundant Multithreading (RMT) technique is proposed in [135]. The proposed technique modifies an application's code during compile-time to add redundant execution. However, these works may cause a significant amount of performance and power overhead, because the proposed techniques are based on redundant execution or recomputation.

Research has also been conducted to improve the soft-error reliability by leveraging the parallel behavior of the GPU [39, 111]. It is shown in [39] that the GPU's soft-error reliability is affected by its parallel behavior. For example, one research in [111] has demonstrated that the Mean Executions Between Failures (MEBF) of a GPGPU application is affected by the parallel behavior of the GPU. In order to show the relationship between the MEBF and the parallel behavior of the GPU, the *grid* and the *block size* of a GPGPU application is modified. However, these research works do not provide the techniques to find the *grid* and

the *block size* and improve the soft-error reliability of the GPU.

Various instruction scheduling algorithms have been proposed to improve the soft-error reliability [114, 115, 116]. One research project in [115] has proposed a metric to quantify the soft-error reliability by using both the detailed timing behavior of the application and the hardware information. Based on the proposed metric, an instruction schedule is generated to maximize the soft-error reliability. Other research in [114] has proposed an instruction schedule algorithm to maximize the soft-error reliability under various performance constraints. Another research in [116] has proposed an instruction schedule to maximize the soft-error reliability for a specific component. Based on the compiler option, the proposed instruction scheduling algorithm maximizes the soft-error reliability of the selected components. For example, if the compiler option selects the register file, then the proposed instruction generates an instruction schedule that maximizes the soft-error reliability of the register file. However, since these instruction scheduling algorithms are based on RISC processors such as SPARC-V8 architecture, they are not applicable to the GPU. In addition, due to the fact that the probability of the soft-error occurrence is proportional to the time that the hardware component is used, the performance-aware instruction scheduling may be considered to improve the soft-error reliability [52]. However, since the performance-aware instruction scheduling focuses on maximizing the GPU resource utilization, improvement in the soft-error reliability is limited.

Research has been conducted for detection and protection of the vulnerable parts in GPGPU applications [21, 32, 73, 100]. One research in [32] has proposed the checker functions that are inserted to protect the potentially vulnerable parts of a GPGPU application. Other research in [100] has proposed a compilation technique that improved the control-flow reliability. Another research in [73] has proposed a compile-time methodology that protects the memory access instructions by inserting checker instructions. One study in [21] has proposed an application-level technique that modifies the loop code during the compile-time. However,

these works have limitations in improving the soft-error reliability of the GPU because the soft-error may occur in any part of a GPGPU application.

A large body of research aims to minimize the effects of NBTI and HCI on multi/many-core systems. The work in [42] surveys various trade-off points between power, performance and reliability in multi/many-core systems-on-chip. The surveys show that there are multiple control knobs to optimize power, performance, and reliability of multi/many-core systems.

A task migration technique [9] is proposed to minimize the effect of NBTI. The proposed technique uses the spare processor in order to migrate tasks from the near-to-die primary processor to the young processor. However, this technique is not applicable to the GPUs because they do not have spare cores. A circuit design technique is proposed to minimize the amount of NBTI-induced clock skew [17]. A Network-on-Chip (NoC) router architecture was proposed to increase the lifetime of the NoC-based general purpose Chip Multi-Processors (CMPs) [63]. The proposed architecture optimizes the duty cycle of the NoC during its idle time. Whenever the NoC router does not have any packet to process, the proposed exercise module operates and optimizes the duty cycle as much as possible. However, the proposed architecture is not applicable to the GPUs because it is tailored to the NoC-based general purpose CMPs and has a considerable amount of area/power overhead that restricts its application to GPUs which have many small cores. The work in [3] has proposed a dynamic routing algorithm for heterogeneous NoC architectures to overcome the NBTI-induced performance degradation. Based on the NBTI information of each router, the routing path is generated to minimize the performance degradation.

Task allocation techniques are proposed to minimize the impact of NBTI degradation on heterogeneous multi-core processors [103, 126]. The proposed techniques assign the application's function to processors based on the application information and the current NBTI status. However, unlike existing NoC-based multi-core architectures, the computational cores in the GPU form clusters of cores and share the instructions. Therefore, the afore-mentioned works

are not scalable to embedded GPUs. The concept of Capacity Rate (CR) [125] has been proposed to indicate whether a core is able to accept the workload or not. The proposed CR is estimated based on the critical path delay and the power consumption status of the core. Based on the estimated CR, tasks are assigned to the cores to balance workloads while minimizing communication overhead. However, since there is no direct communication between the cores in the GPUs, the proposed technique is not suitable. The work in [15] shows that the power gating technique can be used to reduce the NBTI effect. Using the power gating and several circuit models, the work shows that the power gating may allow synthesizing low-leakage circuits with maximum lifetime. However, this work does not discuss how the power gating should work with complex multi-core architectures to minimize the NBTI effect. An analytical model [47] has been proposed to estimate the lifetime of Multiprocessor System-on-a-Chip (MPSoC). Based on the proposed analytical model, a mapping between the tasks and the processors are generated. However, the proposed model aims for the MPSoC platform, thus it is not scalable to the GPUs. An on-chip NBTI degradation sensor is proposed in [60]. The proposed sensor uses a delay-locked loop to measure the threshold voltage in a PMOS transistor. However, there is no discussion on how to maximize the lifetime of the GPUs.

Research has been conducted to control the behavior of processors under the process variation. A thread-to-core mapping technique is proposed in [120] to maximize the performance of a many-core processor. During run-time, the proposed technique dynamically generates thread-to-core mappings based on the current thermal and power profile. The work in [38] has proposed a thread-to-core mapping technique to mitigate the aging effect on many-core processors under the process variation. The proposed technique generates an aging prediction map based on the power consumption pattern and the process variation. A scheduling algorithm [46] is proposed to maximize the performance of MPSoC-based systems under the process variation. The proposed scheduling algorithm considers the spatial correlation between the processors to minimize the effect of the process variation. A run-time workload

distribution technique [104] has been proposed to minimize the effect of the process variation. It employs a scheduling policy that uses a linear programming and a bin-packing formulation together. A variation-aware task mapping algorithm for MPSoC is proposed in [82] that assigns the execution time of each core. However, given that the above mentioned techniques aim for Network-on-Chip (NoC) based multi/many-core platforms, they are not scalable to the GPUs. A frequency assignment technique has been proposed in [2] to maximize the performance of GPUs under process variation. The proposed technique assigns different frequencies for each SM core. After this step, SMs are assigned to applications based on their workload profile. The work in [81] has proposed a technique to minimize the effect of process variation on a processor. The proposed technique controls the mapping between tasks and cores based on the aging status and implements per-core Dynamic Voltage and Frequency Scaling (DVFS) to maximize the lifetime of the processor. A notion called Instruction-Level Vulnerability (ILV) has been proposed in [109]. Through the ILV, the process variation can be exposed to the software to maximize the performance of systems. However, since the proposed ILV is tailored to RISC processors, it is not applicable to the GPUs. The work in [23] has proposed a technique to map application threads to Simultaneous Multithreading (SMT) cores under the process variation. The proposed technique utilizes the operating system and hardware performance counters to create an application profile under the process variation. After creating the profile, based on the application profile, the mapping between the application threads and the cores is generated to maximize the performance of the system. However, the application's profile based on the SMT cores is not able to capture the characteristics of the GPUs.

Research has been conducted to alleviate the aging effects on the GPUs under the process variation. A system-level technique [25] is proposed to measure the effect of NBTI on the GPU and to find the defected cores. However, since the proposed technique only detects defected cores, the proposed technique does not control workload distribution and maximize the lifetime of the GPU. In order to minimize the NBTI effect on the warp scheduler in a

GPU, the work in [147] has proposed a two-stage architecture. In order to minimize the workload on the warp scheduler, the proposed two-stage architecture implements an input module that checks the warps and sends some warps which are ready to execute. An NBTI-aware register file allocation technique has been proposed in [88] that includes additional hardware to perform aging-aware register assignment. The work in [127] has proposed a technique to mitigate the effects of the NBTI and the process variation on the register file of GPUs. The proposed technique classifies the register files into two categories: fast and slow register files. After this classification, based on the application profile and NBTI status, the fast and the slow register files are assigned. However, these works do not consider the utilization of computational cores on the GPU.

To minimize the NBTI effect and maximize the lifetime of the GPU, compiler-based techniques [75, 108] have been proposed. During run-time, the Just-In-Time (JIT) compiler generates a healthy kernel function based on the current aging status of the GPU. This healthy kernel includes some additional workloads to transfer the workloads from the degraded cores to healthy cores. The amount of additional workload depends on the number of degraded cores. However, the proposed techniques require the JIT compiler support, which causes a non-negligible performance overhead, for run-time kernel compilation. Moreover, the additional workloads may cause a significant performance overhead to transfer the workloads between the degraded cores and healthy cores. An SM level clock gating technique [19] has been proposed to maximize the lifetime of the GPUs. The proposed technique finds the optimal number of SMs to handle GPU applications and controls the clock signal at SM granularity. However, in general, the embedded GPUs have few SMs [94]. Thus, the proposed technique is not scalable in this case. A technique to minimize the process variation effect on the GPU has been proposed in [70]. The proposed technique maximizes the performance of the GPUs by applying maximum frequency to each core, which may aggravate aging or disable slow cores which may result in performance loss. Overall, this technique does not consider the lifetime of the GPUs under the process variation. A workload management

technique [69] is proposed to maximize the lifetime of the GPU. It assigns the instructions to different clusters based on the current aging status. However, since it does not consider the process variation, it may not improve the lifetime of the GPU in the presence of process variation.

State-of-the-art application scheduling frameworks have been proposed in order to leverage the GPU-based embedded system's performance and low power consumption. Works in [51, 67] use GPUs to improve the performance of real-time graphics applications such as 3D image reconstruction and facial detection. Work in [148] improves the IP routing performance by using GPUs. Work in [28] discusses the limitations of GPUs in the real-time applications. Works in [33, 99, 131, 145] use GPUs to process the compute-intensive part of medical imaging applications and archive real-time performance. Works in [22, 36] use GPUs to improve the performance of object tracking applications. Work in [50] proposes *Application Programming Interfaces* (APIs) for real-time systems in order to ease the use of GPUs. Work in [134] proposes an algorithm to distribute the large amount of data and schedule the workloads to achieve high utilization of a multi CPU-GPU platform (under real-time constraints). Work in [87] presents a real-time road sign detection framework for the *Advanced Driving Assistance Systems* (ADAS), where the presented framework uses the *Particle Swarm Optimization* (PSO) algorithm to improve the detection performance. The above mentioned research works use the GPUs to improve the performance of a single application and achieve real-time performance. However, the above mentioned works are tailored for single applications, they have limitations in handling multiple and different applications.

In order to support multiple applications on complex real-time embedded systems (such as our target GPU-based embedded system), preemption is required. To overcome this problem, various research works have been conducted. Work in [89] proposes a lowest priority first based light-weight feasibility analysis of CPU-based real-time systems. However, since the proposed work focuses on the feasibility analysis of CPU-based real-time systems, there is

a limitation for distributing GPU resources to multiple applications. Work in [58] proposes a GPGPU execution model to ensure the response time of high priority applications. Work in [56] proposes an automatic task distribution technique on the CPU and the GPU for a real-time systems. Work in [35] proposes a framework that controls multi-CPU/multi-GPU real-time systems. Work in [27] proposes a locking protocol for globally-scheduled *Job-Level Static-Priority* (JLSP) real-time systems (which uses GPUs as shared resources). The utilization of the GPU is improved by using the proposed locking protocol. Work in [138] proposes a priority donation based locking protocol for the globally-scheduled multi-processor real-time systems. Work in [80] proposes a run-time task-scheduling and resource management mechanism for the medical imaging applications. Work in [91] proposes a run-time checkpoint framework that suspends and resumes the applications. Note that this proposed framework manages the checkpoint and controls the application flow in the main thread. Work in [26] describes the GPU as a shared resource and generates a GPU allocation schedule for a given application set. Work in [59] proposes a GPU scheduler for periodic applications. Different scheduling properties are assigned to each application. According to the scheduling property and the priority of the application, the application is selected and submitted to the GPU. However, a small number of applications is considered. Work in [144] proposes a locking protocol for real-time systems on GPU-based real-time systems. The utilization of the GPU is improved by using the proposed locking protocol. However, one application can occupy the GPU at a time. The above mentioned research works are limited in providing flexible and fine-grained GPU resource management due to the limitation of allowing only one application to occupy the GPU. Moreover, although the GPU is a multi-core processor, spatial preemption is not considered and only temporal preemption is implemented (see Section 5.3.1 for our definition of temporal and spatial preemptions).

Work in [10] proposes a fine-grained GPU resource management framework. The proposed framework partitions the GPU workload into small sub-workloads to implement preemption. However, since this framework only considers periodic applications, it has limitations

in partitioning random GPU workloads during run-time. Work in [68] proposes a scheduling framework for event-driven real-time systems. This framework generates the mapping between the SMs and the applications in order to provide temporal and spatial preemption. However, the scheduling policy in [68] assigns at least one SM to each running application. This may cause a bottleneck when the number of running applications is larger than the number of SMs.

In summary, the previously mentioned state-of-the-art workload management frameworks suffer from the following limitations:

- Existing workload management techniques do not jointly consider periodic applications and temporal preemption. They allow only one application to occupy the GPU at any given time. Moreover, a performance bottleneck might be observed in the event that the number of running applications is larger than the number of SMs.
- Existing workload management techniques distribute the workload based on the chip-level guardbanding technique. Therefore, their workload management techniques may not minimize the aging effect in the presence of **the process variation** and **core-level guardbanding**. Although the register file and instruction scheduler have been considered, **computational cores** have not been considered from the perspective of aging optimization.
- They control the parallel behavior of the GPU and modify the application source code to improve the soft-error reliability. However, the instruction scheduling methodologies have not been considered to further improve the soft-error reliability.

Chapter 3

Design-time: GPU Architecture-aware Instruction Scheduling Algorithm

In this chapter, in order to improve the soft-error reliability of the GPU, we propose a novel GPU architecture-aware instruction scheduling algorithm. The proposed algorithm jointly considers the parallel behavior of the GPU hardware and the applications, and minimizes the vulnerability of the GPU applications during instruction scheduling. In addition, the proposed algorithm is able to complement any hardware based soft-error reliability improvement techniques. We compared our instruction scheduling algorithm with the state-of-the-art soft-error reliability-aware techniques and the performance-aware instruction scheduling algorithm. We have injected the soft-errors during the experiments and have compared the number of correct executions that have no erroneous output.

3.1 GPU Application Model

In this section, we describe our target GPU-based embedded system. In this dissertation, for the sake of consistency, we use terminology that is defined in NVIDIA's Compute Unified Device Architecture (CUDA) [92]. Our target platform is a GPU-based embedded system that has similar GPU architecture to NVIDIA's Fermi architecture. It has a multi-core CPU and a single on-chip GPU that has a total of N_{tot} SMs. Note that both Kepler and Fermi architectures contain similar hardware units to execute the GPU kernel [93].

We assume that the target platform has on-chip delay monitors that are shown in [19, 124]. These delay monitors provide aging information to the proposed technique. Note that the in-situ aging monitors like the one proposed by [66] can be used to obtain aging information. The embedded GPU in the target platform is assumed to have Dynamic Voltage and Frequency Scaling (DVFS) functionality and include an on-chip power-gating unit, which is shown in [141]. Generally, a GPGPU application consists of two parts: a host code and a kernel code. The host code represents the code segment that is handled by the CPU (host) and controls the behavior of a GPGPU application. The kernel code represents the computationally intensive code segment, which is handled by the target GPU. When the host launches a GPU kernel, the host must set the $configuration$ for the GPU kernel to create the thread hierarchy, where the *grid* indicates the dimension of the thread block and the *blocksize* indicates the number of threads in the thread block. Right after the host launches a kernel function, the mapping between the thread blocks and the SMs are created based on the $configuration$ and the entire kernel function is submitted into GPU hardware. Generally, after the device starts executing the kernel function, it may not be suspended without forced termination because of the hardware limitations of the GPU.

When the GPU executes a kernel function, the threads in the same thread block are grouped into a *warp*. Note that a single thread block can have multiple *warps*. $Warp$ is the basic

unit of execution of the GPU. The threads in the same *warp* fetch and execute the same instruction. The behavior of *warps* in the same thread block is sequential because they share the single SM. Since multiple SMs operate at the same time, the behavior of thread blocks could be either parallel or sequential. Since the detailed architecture of the GPU is not available to the research community, we assume that the GPU uses the round robin algorithm to schedule the warps as has been adopted in [8].

During run-time, the target system must respond to many random external and internal events, such as physical events, user commands, messages from other embedded systems, and so on. Therefore, the applications in the target system are event-driven and run concurrently. A small timing violation may cause degradation in the QoS. However, this does not result in a system failure. For example, let us assume that one image-processing application in the system has a small timing violation (see Figure 1.1). The effect of this small timing violation may cause a few graphical glitches in the video or image. However, these glitches may not cause a system failure. Therefore, we assume that our target GPU-based embedded system is a *soft real-time system*. In addition, the system is expected to increase the number of higher priority applications meeting their deadlines as much as possible.

3.2 Fault Model

Neutron-induced soft-errors are considered as a primary reason for a bit-flip. Since the natural radiation is evenly distributed across the chip, we assume that neutron-induced soft-errors are evenly distributed over the GPU area. As discussed in Chapter 2, several metrics and techniques have been proposed to quantify the impact of soft-errors.

Among these soft-error quantification metrics, the *Architectural Vulnerability Factor* (AVF) [86] and the *Instruction Vulnerability Index* (IVI) [115] are two of the most well-known

metrics used for quantifying the soft-error reliability. The AVF represents the probability that a single soft-error on a hardware component causes a visible error. In AVF metric, the probability of the soft-error is proportional to the amount of the time that the data resides on a hardware component. In order to estimate the AVF, the detailed timing behavior of hardware components needs to be obtained. Based on the usage of each hardware component, the AVF is calculated. In AVF metric, a hardware component is susceptible to the soft-error while the hardware component is in use. This susceptible period is referred to as **vulnerable period**. The IVI metric does not only consider the temporal effect, which is considered in the AVF metric, but also the spatial effect, the area of each component. Since the soft-error is evenly distributed throughout the processor, the probability that the soft-error occurs on the hardware component is affected by the area of the component as well.

In this dissertation, since a detailed Register Transfer Level (RTL) model, which must be required for the IVI metric, is not available to the research community, we use the AVF metric to quantify the soft-error effect. However, if the RTL model of the GPU is provided, then our algorithm is also scalable to the IVI metric as has been done for the single-core SPARC-V8 CPU in [115].

3.3 Vulnerable Period Estimation for GPU

Since the parallel behavior of the GPU affects the soft-error reliability of the GPU as well, the architectural characteristics of the GPU also need to be considered to estimate the vulnerable period of the GPU application. During run-time, each *warp* runs instructions in lock-step, and it is assumed that the GPU hardware schedules the *warps* in a round-robin manner [8]. The latency of the instruction depends on the parallel behavior of the GPU pipeline and the memory access latency [44]. By using the state-of-the-art GPU performance models, we may estimate the timing behavior of the GPU and extract information to estimate the vulnerable

period. Below, we provide more details about the latency estimation of the instruction which uses the state-of-the-art GPU performance model from [7, 44]. The model parameters are generated from the actual GPU by using micro benchmarks.

3.3.1 Latency of Instruction Execution

The n^{th} instruction can be issued after all *warps* have issued the $(n-1)^{th}$ instruction and all the data for the n^{th} instruction is ready to use. Therefore, the amount of time to execute n instructions may be represented by the following equation:

$$t_{I_n} = \sum_{i=1}^{n} \left(L^{i}_{ready} + L_{issue} \right) + L^{n}_{pipe} + L^{n}_{mem} \tag{3.1}$$

where L^{n}_{ready} represents the latency to make the n^{th} instruction ready. L^{n}_{issue} represents the latency to issue all the instructions for the n^{th} instruction. L^{n}_{pipe} and L^{n}_{mem} represent the pipeline and the memory access latency for the n^{th} instruction, respectively. The instruction issue latency, L_{issue}, shown in Figure 3.1 is represented by the following equation:

$$L_{issue} = \frac{n_{warp} \times sizeof(warp)}{n_{core} \times r_{issue}} \tag{3.2}$$

where n_{warp} represents the number of *warps* in a thread block. n_{core} represents the number of streaming processors in a single SM, and r_{issue} represents the instruction issue rate. The pipeline latency, L_{pipe}, depends on the instruction and can be obtained empirically.

As mentioned in the beginning of this section, the n^{th} instruction will be issued after all the data dependencies are resolved. After the $(n-1)^{th}$ instruction is issued completely, if there is unresolved data dependency then the n^{th} instruction will wait until all the data is ready to use. The instruction ready latency, L^{n}_{ready}, is a function of the L^{p}_{pipe} and L^{p}_{mem} of the producer instructions p. Therefore, L^{n}_{ready}, depends on the latencies of the producer instructions and may be represented by the following equation:

$$L^n_{ready} = max(t^{n-1}_{issue\_comp}, t^n_{data\_ready}) - t^{n-1}_{issue\_comp} \tag{3.3}$$

$t^{n-1}_{issue\_comp}$ represents the time that all the $(n-1)^{th}$ instructions are issued and $t^n_{data\_ready}$ represents the time that all the data for the n^{th} instructions are ready to use. Moreover, the instruction execution may overlap with the other instruction's issue and the memory operation. Therefore, the completion time of the n^{th} instruction may be described by the following equation:

Equation 3.1 shows that the latencies caused by the data dependencies are related to both the arithmetic and memory operations. To accurately measure the data dependent latency, the latencies of the arithmetic and memory access instructions need to be considered together.

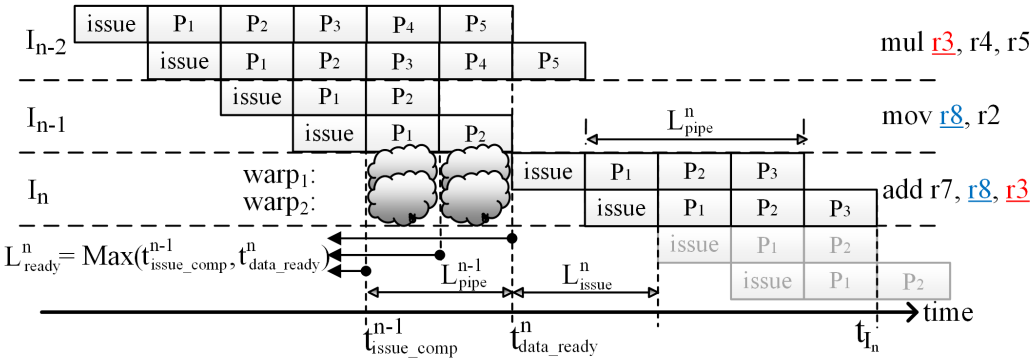


Figure 3.1: Example Pipeline Stages of Arithmetic Instructions.

Latency of Arithmetic Instruction

The latency of the arithmetic instruction consists of two latencies: the instruction ready latency and the pipeline latency. The instruction ready latency is the period between the issue completion time of the previous instruction and the latest completion time among the predecessor instructions. Figure 3.1 shows an example of the instruction ready latency. The issue completion time may be estimated by the summation of the instruction ready latencies and the L_{issue} for all the instructions preceding the n^{th} instruction. The instruction ready latency may be estimated by the instruction whose execution time is the longest and has finished after the issue completion time of the dependent instructions. Note that

the pipeline latency likely overlaps with the issue latency. Therefore, the instruction ready latency is calculated by subtracting the issue latency from the pipeline latency. In Figure 3.1, the instruction I_{n-2} actually dominates the instruction ready latency since it has the longest execution time. The instruction ready latency is two clock cycles for I_n because one overlapped issue cycle of I_{n-2} and one overlapped issue cycle of I_{n-1} are subtracted from the longest execution time.

Latency of Memory Access Instruction

The latency of the memory access instruction is calculated by subtracting the issue latency from the total memory access time. Figure 3.2 shows an example of the memory access latency.

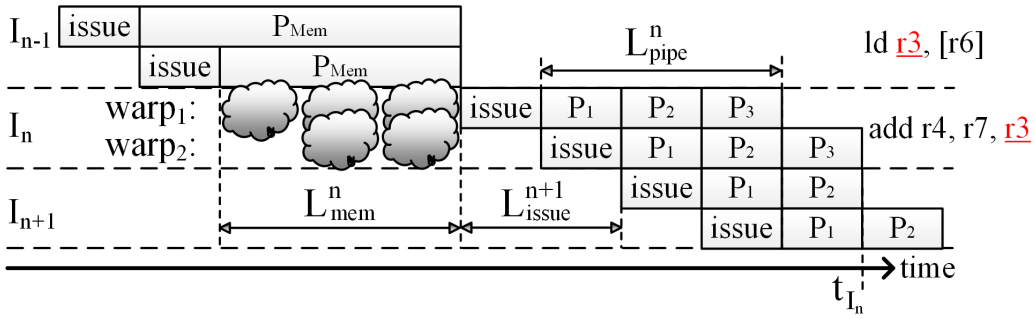


Figure 3.2: Example Pipeline Stages of Memory Access Instructions.

Assuming that the system has adopted GDDR memory, the maximum memory bandwidth, BW_{MEM}, is the bandwidth of the bus multiplied by the clock frequency of memory (Equation 3.4) [41]. Due to the fact that the exact cache behavior may be impossible to estimate during compile-time, the average memory access latency is used to estimate the memory access latency for the n^{th} instruction.

$$BW_{MEM} = Bus\_width \times Memory\_freq \times 2 \tag{3.4}$$

During run-time, the memory bandwidth is consumed by the memory accesses. The consumed bandwidth for each memory access may be estimated by the following equation:

$$bw = Fetch\_size \times Core\_freq \tag{3.5}$$

After that, the maximum number of concurrent memory accesses may be estimated by the following equation:

$$N_{mem\_req} = \lfloor BW_{MEM}/bw \rfloor \tag{3.6}$$

During run-time, the number of total memory accesses from the instruction is calculated by the following equation:

$$n_{mem\_req} = n_{warp} \times n_{SM} \tag{3.7}$$

Since all the memory accesses may not be served at the same time, the memory latency is estimated based on currently available bandwidth, number of ongoing memory accesses, and number of requested memory accesses. The memory latency consists of two parts: the latency for memory accesses (L_{mem}) and the delay due to the fully occupied memory bandwidth (d^n_{full}).

$$L^n_{mem} = L_{mem} + d^n_{full} \tag{3.8}$$

If there is no ongoing memory access, d^n_{full} becomes 0 and the memory access latency may be estimated by the following equations:

$$N_{full} = \left\lfloor \frac{n_{mem}}{N_{mem\_req}} \right\rfloor \tag{3.9}$$

$$N_{remain} = MOD(n_{mem}, N_{mem\_req}) \tag{3.10}$$

$$L_{mem} = \left(N_{full} + \left\lceil \frac{N_{remain}}{N_{mem\_req}} \right\rceil \right) \times L_{avg\_mem} \tag{3.11}$$

The above equations imply that the memory bandwidth is not always fully occupied. Therefore, in order to track the memory bandwidth and describe the memory access latency, we define three parameters: 1) the time when there is available memory bandwidth ($t^n_{nonfull}$), 2) the time when the memory access is completed ($t^n_{memfinish}$), and 3) the number of avail-

able memory accesses between $t^n_{nonfull}$ and $t^n_{memfinish}$ ($N^n_{nonfull}$). By using these parameters, we may describe the memory latency and track the bandwidth status. Note that $N^n_{nonfull}$ may represent the available memory bandwidth between $t^n_{nonfull}$ and $t^n_{memfinish}$ because the available memory bandwidth can be represented by the number of available memory accesses. In order to track the memory status, $t^n_{nonfull}$, $t^n_{memfinish}$, and $N^n_{nonfull}$ need to be updated for each instruction. If the n^{th} instruction is not a memory access instruction, then $t^n_{nonfull}$, $t^n_{memfinish}$, and $N^n_{nonfull}$ are the same as the previous instruction's values, which are $t^{n-1}_{nonfull}$, $t^{n-1}_{memfinish}$, and $N^{n-1}_{nonfull}$. However, if the n^{th} instruction is a memory access instruction, $t^n_{startmem}$, $t^n_{nonfull}$, $t^n_{memfinish}$, and $N^n_{nonfull}$ are updated based on the current memory status, which is represented by $t^{n-1}_{nonfull}$, $t^{n-1}_{memfinish}$, and $N^{n-1}_{nonfull}$. From Equation 3.1, by making $L^n_{mem} = 0$, it may be possible to estimate the time when the n^{th} instruction's memory accesses are sent to the memory system, $t^n_{startmem}$.

Based on $t^{n-1}_{nonfull}$, $t^{n-1}_{memfinish}$, and $t^n_{startmem}$, there are three different possible scenarios for estimating the memory accesses latency.

- $t^n_{startmem}$ is greater than $t^{n-1}_{memfinish}$. In this scenario, there is no ongoing memory access at time $t^n_{startmem}$. Therefore, the memory access latency may be estimated by using the above mentioned equations (Equations 3.9-3.11). In addition, $N^n_{nonfull}$ is same as N_{remain}.

- $t^n_{startmem}$ is between $t^{n-1}_{nonfull}$ and $t^{n-1}_{memfinish}$. In this scenario, there are ongoing memory accesses, but the memory bandwidth is not fully occupied. Some memory accesses can begin immediately and the other memory accesses can begin after $t^{n-1}_{memfinish}$. Therefore, d^n_{full} becomes 0 and based on the available bandwidth, $N^{n-1}_{nonfull}$, and N_{remain}, the memory latency and the parameters are updated. When N_{remain} is greater than $N^{n-1}_{nonfull}$, the last memory access is followed by the memory accesses that start at $t^{n-1}_{memfinish}$. Therefore, the parameters are updated as follows:

$$t^n_{nonfull} = t^n_{startmem} + L^n_{mem} \tag{3.12}$$

$$t^n_{memfinish} = t^{n-1}_{memfinish} + L^n_{mem} \tag{3.13}$$

$$N^n_{nonfull} = N_{mem\_req} - (N_{remain} - N^{n-1}_{nonfull}) \tag{3.14}$$

In the case hen N_{remain} is less than or equal to $N^{n-1}_{nonfull}$, the last memory access is followed by the memory access that starts at $t^n_{startmem}$. In addition, the available memory bandwidth, $N^n_{nonfull}$, is related to the memory accesses that are finished at time $t^{n-1}_{memfinish}$. Therefore, the parameters are updated as follows:

$$t^n_{nonfull} = t^{n-1}_{memfinish} + (N_{full} \times L_{avg\_mem}) \tag{3.15}$$

$$t^n_{memfinish} = t^{n-1}_{memfinish} + L^n_{mem} \tag{3.16}$$

$$N^n_{nonfull} = N_{mem\_req} - N_{remain} \tag{3.17}$$

In the case when N_{remain} is equal to 0, the memory access behavior is exactly the same as the previous one. Therefore, the parameters are updated as follows:

$$t^n_{nonfull} = t^n_{startmem} + (N_{full} \times L_{avg\_mem}) \tag{3.18}$$

$$t^n_{memfinish} = t^{n-1}_{memfinish} + (N_{full} \times L_{avg\_mem}) \tag{3.19}$$

$$N^n_{nonfull} = N^{n-1}_{nonfull} \tag{3.20}$$

- $t^n_{startmem}$ is less than $t^{n-1}_{nonfull}$. In this scenario, there is no available memory bandwidth. Some memory bandwidth becomes available after $t^{n-1}_{nonfull}$. This scenario is a special case for the previous scenario where $d^n_{full} > 0$ and the memory accesses at $t^{n-1}_{nonfull}$, because the memory access behavior will be similar as the previous scenario after $t^{n-1}_{nonfull}$. The memory access latency may be estimated by adding the delay d^n_{full} to the equations in the previous scenario. d^n_{full} may be estimated by the following equation:

$$d^n_{full} = t^{n-1}_{nonfull} - t^n_{startmem} \tag{3.21}$$

3.3.2 Control Flow Management During Vulnerable Period Estimation

Before estimating the vulnerable period of the application and scheduling of the instructions, the control flows of an application need to be considered. These control flows are implemented by using the following: the loops and the branches. We provide the following two observations to handle the loops and the branches during the vulnerable period estimation. Note that these observations are based on the state-of-the-art GPU architecture and performance models [8, 44].

***Observation* 1.** *Let's assume that the register R includes the moment of data production before the loop and the last moment of data consumption in the loop. The vulnerable period of the register R, VP_R, depends on the following:*

- *Execution time of one iteration in the loop, l_{loop}.*
- *Time from the data production to the first data consumption in the loop, $l_{pre-loop}$.*

Proof. During run-time, depending on the data, the number of iteration N is changed. In other words, two different instruction schedules will have the same number of loop iterations, if the input data is the same. The vulnerable period of the register R may be represented by the following equation:

$$VR_R = \begin{cases} N \times l_{loop}, & \text{if } \frac{l_{pre-loop}}{l_{loop}} \approx 0 \\ N \times l_{loop} + l_{pre-loop}, & \text{otherwise} \end{cases} \tag{3.22}$$

From Equation 3.22, we can see that $l_{pre-loop}$'s contribution to the VP_R becomes negligible if one of the following two cases is satisfied: 1) $\frac{l_{pre-loop}}{l_{loop}}$ is approximately zero; or 2) N is so large. Therefore, the vulnerable period of the loop can be improved during compile-time by minimizing both the l_{loop} and the $l_{pre-loop}$. Figure 3.3(a) shows an example code to demonstrate the proof of this theorem for a for-loop. □

***Observation* 2.** *For register R, which includes the moment of data production before the branch instruction and the last moment of data consumption within the branch body, the vulnerable period of R depends on the location of the last consumption of R.*

Proof. Unlike the general CPU, the GPU likely executes every branch body to handle control flow divergence no matter how the branch is taken. Moreover, for the same input data, two different instruction schedules show the same branch behavior. Therefore, the vulnerable period of the R is calculated based on the last place where R is used. Figure 3.3(b) shows one such example case to demonstrate the proof of this theorem. □

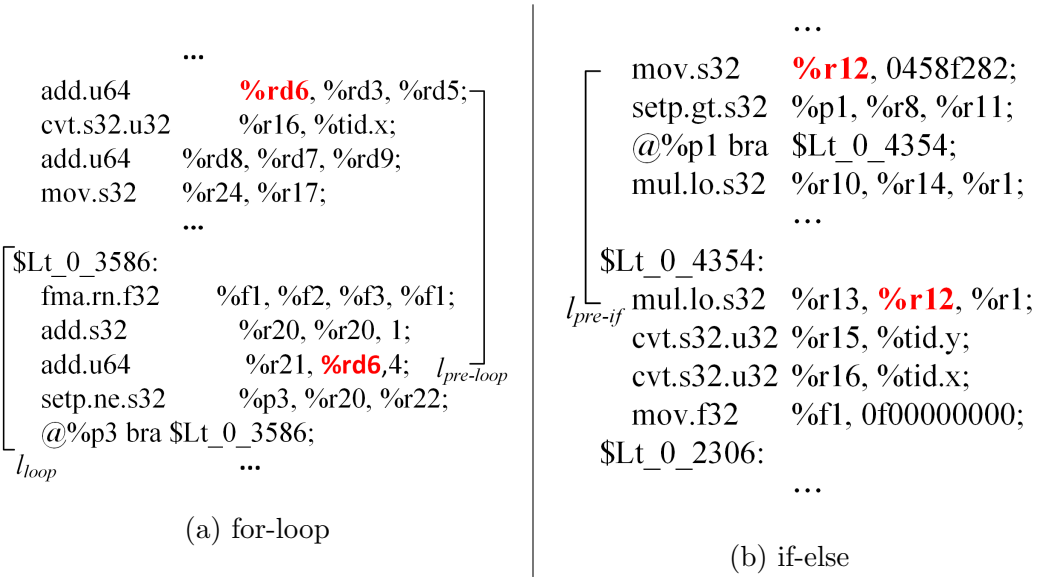


Figure 3.3: Example Kernel Code for For-Loop and If-Else.

3.3.3 Vulnerable Period Estimation

Based on the GPU execution model and the control flow management, we can estimate the vulnerable period of the application. The pseudo-code for the vulnerable period estimation is shown in **Algorithm 1**. The input to the Algorithm 1 includes the instruction flow graph of the kernel code, G, the configuration of the application, c, and the number of SMs, N_{SM} (Line 1). Each node in the graph represents the instruction and each edge in the graph

Algorithm 1: Algorithm for Computing Vulnerable Period.

```
   Input: Instruction Flow Graph G, configuration c, # of SM N_SM
   Output: Total vulnerable period TotVulnPeriod
1  Function EstimateVulnPeriod (G, c, N_SM)
2  begin
3      InitializeGraph(); // Initialize the node and the edge variables
4      TotVulnPeriod ← 0;
5      ProgMissMatchFlag ← 0;
6      G ← DuplicateGraph(G, c, N_SM);
7      foreach N ∈ G do
8          L_issue ← GetIssueLatency(N); // Equation 3.2
9          SetNodeInfo(L_issue);
10         foreach E_out ∈ N do
11             if E_out ← ∅ then
12                 if N = MemOp then
13                     n_concurrent ← FindConcurrentAccess(N) ; // Get the number of concurrent access
14                     L_In ← EstimateMemLatency(N, n_concurrent) ; // Equation 3.4-3.21
15                 else
16                     L_In ← GetPipeLatency(N);
17             SetOutgoingEdge(N, E_out, L_exe) ;
18     foreach G ∈ G do
19         Edges ← Null ;
20         foreach E ∈ G do
21             E_longest ← Edges.find(E) ;
22             if E_longest = NULL then
23                 Edges.add(E);
24             else
25                 SetLongest(Edges, E) ;
26     TotVulnPeriod = GetSummation(G, Edges) ;
27     return TotVulnPeriod ;
```

represents the data dependency. After that, based on the degree of parallelism provided by the configuration c and N_{SM}, Algorithm 1 updates the timing information in the graph (Line 6). The loop (Lines 7-17) starts to update the weight of the edges, which will store L_{I_n} from a node to its dependent nodes. For each node in the graph, L_{issue} is estimated (Line 8) and then for each outgoing edge of the node N, the execution latency L_{I_n} is determined according to the node type (Lines 10-17). In the second loop (Lines 18-25), the total vulnerable period is explored by searching and adding the longest edge from each node.

3.4 GPU Architecture Aware Instruction Scheduling to Improve Soft-error Reliability

As mentioned in Section 1.3.2, our primary goal is to find the instruction scheduling that has minimum vulnerable period. To do that, every possible instruction schedule needs to be tested and verified. However, finding an instruction schedule is known to be an **NP-hard** problem [123]. The overhead of finding an instruction schedule that minimizes the vulnerable period could be significant. Therefore, in order to find the best instruction schedule while minimizing the compilation overhead, we propose a heuristic instruction scheduling algorithm that schedules the instructions based on the data dependency.

3.4.1 Vulnerable Period Aware Instruction Scheduling

The vulnerable period of a register will reach a minimum if the data is immediately consumed after it is produced. Therefore, the primary objective of our heuristic is minimizing the distance between the producer instruction and the consumer instruction. Since our heuristic places instructions based on the data dependencies, it is important to know the last consumption of the data. Therefore, our heuristic uses a bottom-up strategy and places the instructions from the ending point of the application. At the beginning of instruction scheduling, the last instruction is selected and placed. Afterward, the producer instructions corresponding to the last instruction are examined based on the following three conditions:

- There is no synchronization instruction between the producer and the consumer instruction. The synchronization instruction is used to prevent the race conditions between multiple threads. For example, if the instruction (Line 6 in Figure 3.4) is moved after the synchronization instruction, then the functional correctness of the application is not guaranteed.

- There is no other instruction that consumes the same data produced by the producer instruction. If the instruction is moved after its consumer instruction, the consumer instruction has wrong data and the result will be corrupted. For example, in Figure 3.4, if the instruction in Line 8 is moved after Line 11, then the instruction in Line 11 will use the data in register $\%r22$, which is not updated properly.

- The loop depth value of the producer instruction and the consumer instruction must be identical. If the instruction is moved from the inside of the loop to the outside of the loop, the data will not be properly updated. For example, in Figure 3.4, if the instruction in Line 0 is moved after Line 4, which is outside of the loop, the value in register $\%f1$ will not be properly updated and the functional correctness is not guaranteed.

In order to minimize the vulnerable period of the last instruction, our heuristic schedules the predecessor instruction that satisfies all three conditions. After that, the scheduled predecessor instruction becomes the consumer instruction and our heuristic repeats the above mentioned process until all instructions are scheduled.

Algorithm 2 shows the pseudo-code for the proposed instruction scheduling algorithm. At the beginning, the algorithm selects the last instruction and places it at the end of the application code (Lines 10-13). For each predecessor instruction, the algorithm examines the aforementioned three conditions (Lines 15-25). If the predecessor instruction satisfies these three conditions, it will be placed near the consumer instruction (Lines 30-34). The overall complexity of the instruction scheduling algorithm is given by $O(n \times n \times n) = O(n^3)$ because of the three levels loop hierarchy.

Algorithm 2: Algorithm for Instruction Scheduling.

```
   Input: Instruction Flow Graph G_in
   Output: New Instruction Flow Graph G_new
 1 Function BuildFromBottom (G_in)
 2 begin
 3     G_new.clear();
 4     Pos ← |G_in|;
 5     n_tgt ← ∅; RegSet ← ∅;
 6     while G_in ≠ ∅ do
 7         CandidateSet ← ∅;
 8         pass ← true;
 9         if Pos = |G_in| then
10             n_tgt ← GetSinkNode(); G_new.push(n_tgt);
11             G_in.remove(n_tgt);
12             Pos ← Pos - 1;
13         else
14             for idx = 0; idx < Pos; idx++ do
15                 for cnt = idx; cnt < Pos; cnt++ do
16                     if G_in[cnt].is_sync() then
17                         pass ← false;
18                     if pass = true and G_in[cnt].is_consume(G_in[idx].GetDest()) then
19                         pass ← false;
20                 if pass = true and G_in[idx].is_in_loop() then
21                     if G_in[idx].GetLastLoopPos() < Pos then
22                         pass ← false;
23                 if pass = true and RegSet.exist(G_in[idx].GetDest()) then
24                     CandidateSet.push(G_in[idx]);
25             if CandidateSet = ∅ then
26                 n_tgt ← G_in.GetLastNode();
27             else
28                 n_tgt ← CandidateSet.front();
29             G_new.push(n_tgt);
30             G_in.remove(n_tgt);
31             RegSet.RemoveDestRegs(n_tgt);
32             RegSet.AddSrcRegs(n_tgt);
33             Pos ← Pos - 1;
```

3.4.2 Example of the Proposed Instruction Scheduling

An example of our scheduling process is shown in Figure 3.4. In the figure, there are four producer and consumer instruction pairs: instructions in Lines 2 and 5, in Lines 6 and 9, in Lines 8 and 13, and in Lines 10 and 14. The first pair of instructions, in Lines 2 and 5, violates the third condition because they are not in the same loop. The second pair of instructions, in Lines 6 and 9, violates the first condition due to the synchronization instruction in Line 7. The third pair of instructions, in Lines 8 and 13, violates the second condition because there is another consumption of the register $\%r22$ in Line 11. However, the last pair of

instructions, in Lines 10 and 14, satisfies all three conditions, and therefore the proposed heuristic is able to place the instruction in Line 10 around its consumer instruction, in Line 14.

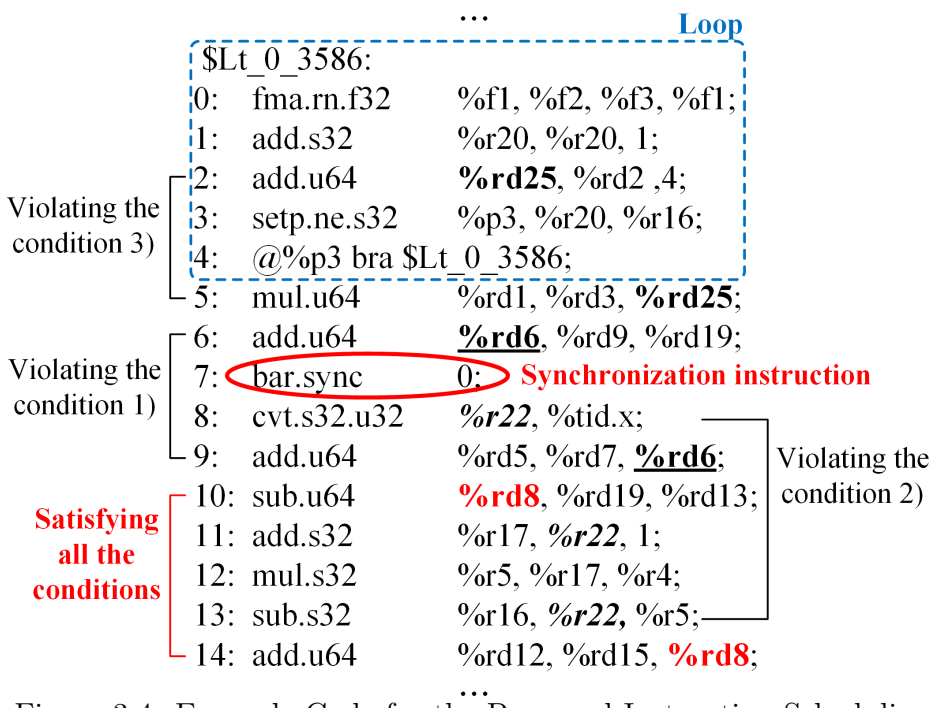
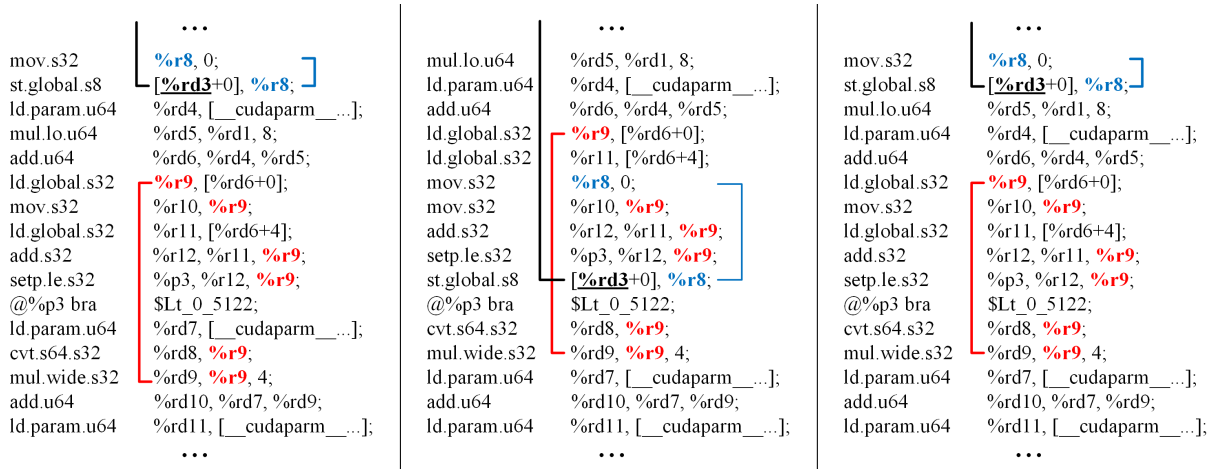


Figure 3.4: Example Code for the Proposed Instruction Scheduling.

3.4.3 Comparison with Performance-aware Instruction Scheduling Algorithm

As mentioned in Section 1.1, the probability that the soft-error occurs on a single hardware component is proportional to the time that the hardware component is used [143]. Therefore, it may be possible to improve the soft-error reliability through performance improvement. The instruction scheduling technique in [52] may be used to improve the performance of the applications.

Current state-of-the-art GPUs do not support out-of-order execution and the instructions will be held until all the data is ready to use [8, 44]. In order to improve the performance of the GPU during compile-time, the instructions are scheduled to increase the *Instruction Level*



(a) Instruction Scheduling from Default NVCC Compiler

(b) Performance Aware Instruction Scheduling [52]

(c) Our Proposed Instruction Scheduling

Figure 3.5: Instruction Scheduling Results for BFS Application from the NVCC, Performance Aware Instruction Scheduling [52], and Our Proposed Instruction Scheduling.

Parallelism (ILP). The increased ILP indicates that some of the data may not be consumed right after its creation and the lifetime of the data is likely increased. Therefore, with the performance aware instruction scheduling, the execution time of the GPU application may be decreased, although the total vulnerable period of the GPU may be increased. Moreover, the instruction scheduling in [52] does not change the instruction schedule in the basic block. There is a possibility that the soft-error reliability of the basic block is not improved through the performance aware instruction scheduling.

Figure 3.5 shows instruction scheduling results for the *Breadth First Search* (BFS) application. Figures 3.5(a), 3.5(b), and 3.5(c) represent the instruction scheduling results from the default NVCC compiler, the performance aware instruction scheduling [52], and our proposed instruction scheduling, respectively. In Figure 3.5, three registers, $\%rd3$, $\%r8$, and $\%r9$, are highlighted to show the difference between the three instruction schedules. For these three registers, the difference between the default instruction scheduling and our instruction scheduling is the lifetime of the register $\%r9$. Our instruction scheduling puts fewer instructions between the creation and the last consumption of the data in the register of $\%r9$. However, in Figure 3.5(b), the performance aware instruction scheduling puts the

largest number of instructions during the lifetime of the register $\%r9$. For the register $\%rd3$ and $\%r8$, both the default and our instruction scheduling put the same number of instructions during their lifetime. On the other hand, the performance aware instruction scheduling puts a larger number of instructions during the lifetime of the $\%rd3$ and $\%r8$.

The examples in Figure 3.5 show that the performance aware instruction scheduling may have the largest vulnerable period even though the execution time of the application is minimal. In Section 3.5, Figure 3.12 shows that the performance aware instruction scheduling achieves the highest performance. However, Figure 3.9 shows that the performance aware instruction scheduling cannot maximize the soft-error reliability of a GPU.

3.5 Evaluation

3.5.1 Experimental Setup

In order to evaluate our instruction scheduling algorithm, we have selected nine benchmark applications because of their intensive usage in GPU applications for image processing and scientific computing. Several applications are selected from the Rodinia benchmark suite [18], GPGPU-Sim benchmark [8], and CUDA examples. The selected benchmark applications are Backprop (Bp), BFS (Bfs), Srad, Kmeans (Km), Matrix Multiplication (Mat), Hotspot (Hs), BoxFilter (Box), ConvolutionSeparable (Conv), and Mandelbrot (Man). During the experiments, each benchmark application is executed 25 times with various fault injection rates (10, 50, and 100 faults/1 Million (M) cycles). In addition, we have implemented a clock cycle level fault injection tool which is integrated with the GPGPU-Sim [8] simulator. Figure 3.6 shows the overview of our fault injection tool.

In our fault injection tool, each kernel is executed twice with GPGPU-Sim, where the list of

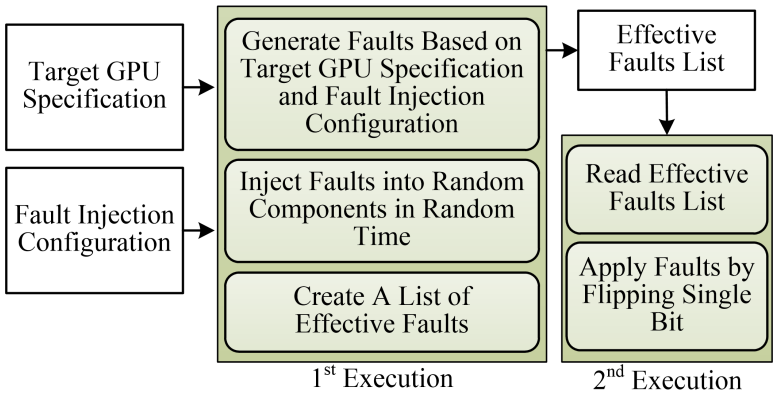


Figure 3.6: Experimental Setup for Fault Injection Flow.

effective faults is created during the first execution and the actual data is changed based on the effective fault list during the second execution. Since the injected faults do not all cause the bit-flip, it is essential to know which fault actually causes the bit-flip.

During the first execution, based on the given fault rate, our fault injection tool periodically and randomly generates the list of injected faults which includes the fault injection time (clock cycles) and the faulty components. The possible faulty components are shown in Figure 3.7, which shows the abstracted block diagram of the NVIDIA's Fermi architecture used for our experiments. Among these components, based on the area information from the GPGPU-Sim, our fault injection tool selects the following components to inject soft-errors: Register File, Load/Store unit, Integer ALU, Floating Point ALU, and Special Function Units. After that, on each clock cycle, the fault injection tool tries to find a match from the list of injected faults based on the following conditions:

- The fault injection time is matched with the current clock cycle number.
- The component in the injected fault list is in use.

If there is a match, the fault is considered as an effective fault, which means this fault causes a bit-flip. The fault injection tool adds the fault into the effective faults log with the following information: the clock cycle number (injection time), the SM number, the thread id, the instruction string, and the faulty component.

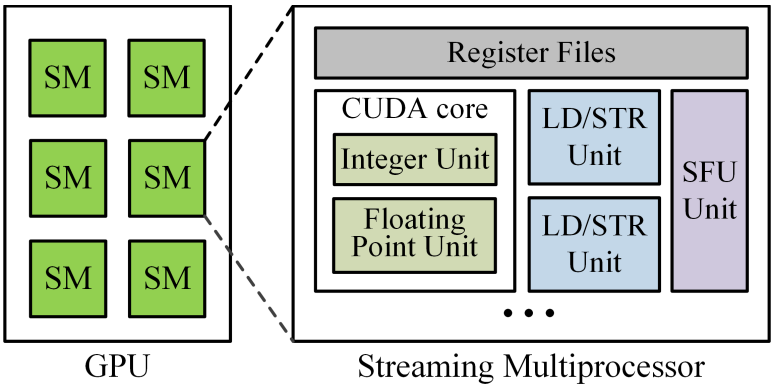


Figure 3.7: A High-Level Block Diagram of GPU Architecture Used for Our Experiments.

At the beginning of the second execution, the fault injection tool reads the effective faults log and obtains the information for the faults. During the second execution, the fault injection tool randomly selects one bit and flips the selected bit from the result of the instruction that uses the faulty component. After the second execution, the entire simulation results are compared to the correct results, which are generated without any fault injection. The output is classified into the following categories: *correct output*, *incorrect output*, and *application crash*.

3.5.2 Experimental Results

The algorithm execution time of our instruction scheduling is shown in Table 3.1. An Intel Core^{TM} i7 Quad-core processor at 3.5 GHz is used to measure the algorithm execution time. On average, our instruction scheduling algorithm requires 8.13 seconds to generate the reliable instruction schedule (standard deviation is 5.356 seconds). The proposed instruction scheduling algorithm adds two steps on top of vendor provided compilation process: 1) PTX code extraction; and 2) execution of instruction scheduling algorithm. After that, vendor provided compiler takes the optimized PTX code and generates the binary. For these two steps, extraction of PTX code has negligible overhead. Therefore, the execution time of the proposed instruction algorithm contributes most to the compilation overhead.

Application	Our algorithm exec. time (sec)
Backprop	13.215
BFS	0.94
Srad	4.41
Kmeans	2.355
Matrix	13.91
Hotspot	13.91
BoxFilter	3.24
convolutionSeparable	9.69
Mandelbrot	11.53
Avg.	8.13
Standard Dev.	5.356

Table 3.1: Our Algorithm Execution Time.

Figure 3.8 shows the vulnerable period improvement of our instruction scheduling compared to reliability enhancement methodologies proposed in [32], array bounds checker in [73], and performance aware instruction scheduling [52]. All results are normalized with the vulnerable period results from the default NVCC compiler (with -O3 option). The results show that our instruction scheduling can achieve the smallest vulnerable period for all, except the *Hotspot* application. This is because our algorithm could not modify the instruction schedule due to frequent usage of synchronization instructions in the *Hotspot* application's kernel function (Section 3.4.1). As mentioned in Section 3.4.3, some of the applications achieve similar vulnerable period improvements through the performance improvement. However, as shown in Figures 3.5 and 3.8, performance improvement cannot guarantee the minimum vulnerable period.

Table 3.2 and Figure 3.9 show the list of effective faults during the entire execution time and the soft-error reliability improvement compared to the state-of-the-art soft-error reliability improvement methodologies [32, 73] and the performance aware instruction scheduling [52], respectively. The results from Table 3.2 are related to the results in Figure 3.9, 3.8 and 3.11. Since the number of effective faults is proportional to the execution time, in order to properly compare the number of effective faults, we also need to consider the execution

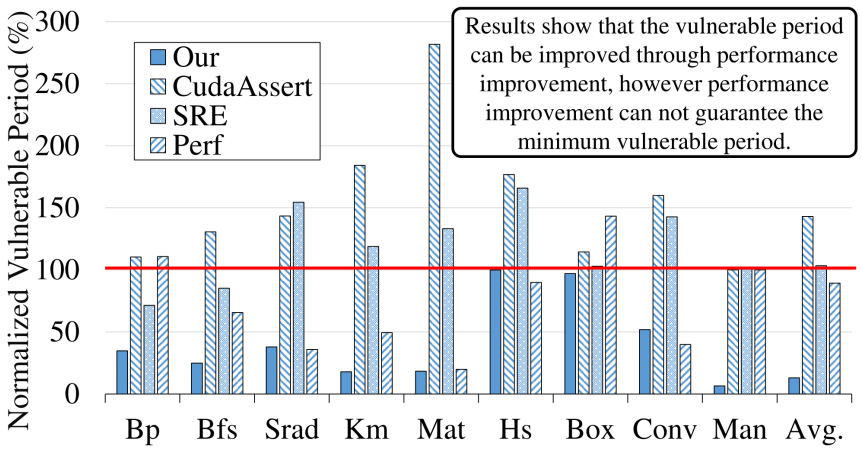


Figure 3.8: Vulnerable Period Improvement Compared to [32], [73], and [52].

time of the application. For example, our algorithm shows less number of effective faults for the *BFS* application, which has a small performance overhead. On the other hand, our algorithm has more effective faults for the *Mandelbrot* application. However, since our algorithm has large performance overhead, the actual effect of the effective faults may be smaller compared to other methodologies. The smaller amount of effect for the effective faults is shown in Figure 3.9 and Figure 3.8. Figure 3.8 shows that our algorithm has smaller amount of normalized vulnerable period and Figure 3.9 supports the result in Figure 3.8 with the soft-error reliability improvement.

Figure 3.9 shows the comparison with the actual output of the benchmark applications. The results show that our algorithm can further improve the soft-error reliability by 23% compared to other two compilation methodologies (up to 64% and standard deviation is 19%). In other words, with our algorithm, the failure probability is decreased by 23% on average (up to 64%). Compared to the performance aware instruction scheduling, our algorithm shows up to 12% improved soft-error reliability (up to 52% and standard deviation is 14%).

The major reason for the improvement is the decreased vulnerable period compared to other methodologies. The methodologies presented in [32] and [73] protect the particular parts

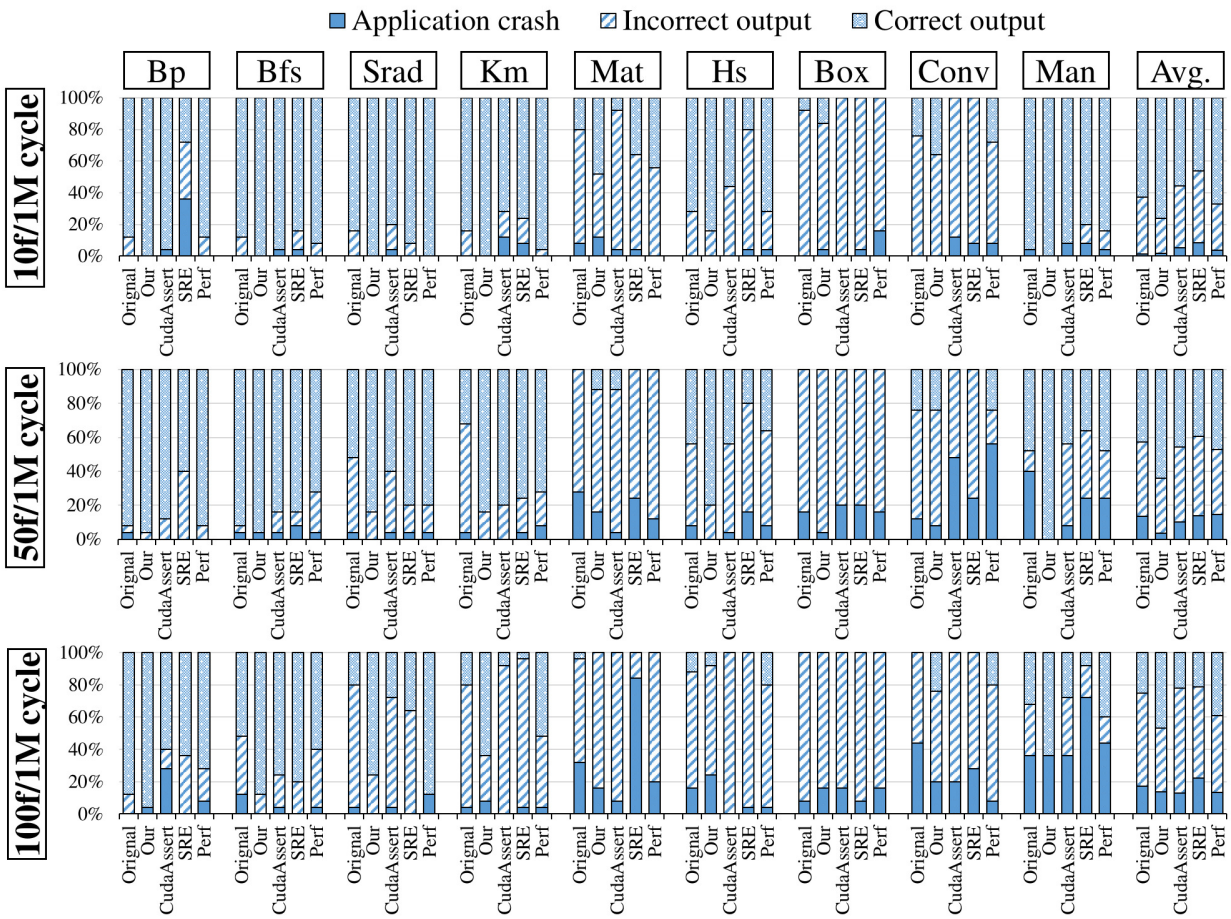


Figure 3.9: Soft-error Reliability Improvement Compared to [32], [73], and Performance Driven Instruction Scheduling [52]. Each Application is Executed 25 Times with Different Fault Injection Rates.

of applications with additional code. However, in general, the soft-errors randomly occur throughout the application, and these methods may provide worse results by forcing the use of additional protection functions. The additional functions increase the overall vulnerable period of application.

The performance aware instruction scheduling may improve the soft-error reliability by improving the performance and decreasing the vulnerable period. However, as shown in Section 3.4.3, the total vulnerable period is not minimized through the performance aware instruction scheduling. Therefore, the methodologies in [32], [73], and [52] may have less soft-error reliability improvement than our instruction scheduling does.

Application	Faulty Component	10 Faults / 1M Clock Cycle					50 Faults / 1M Clock Cycle					100 Faults / 1M Clock Cycle				
		Orig.	Our	[32]	[73]	[52]	Orig.	Our	[32]	[73]	[52]	Orig.	Our	[32]	[73]	[52]
Bp	REGISTER FILE	14	27	13	35	12	66	186	26	122	61	115	403	106	495	129
	LDSTR UNIT	5	7	3	20	7	13	32	9	25	25	27	51	30	66	37
	INT ALU	6	16	8	23	19	10	17	3	42	7	14	49	16	138	19
	FLOAT ALU	19	19	18	18	5	55	85	46	60	100	103	235	149	142	131
	SFU ALU	16	17	9	11	6	34	38	24	29	44	56	106	43	73	64
Bfs	REGISTER FILE	43	26	27	68	12	158	149	61	158	101	268	266	249	596	189
	LDSTR UNIT	6	1	7	7	2	4	6	12	3	5	5	8	5	8	5
	INT ALU	61	0	0	4	31	4	4	0	2	227	4	4	6	10	391
	FLOAT ALU	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	SFU ALU	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
Srad	REGISTER FILE	5	0	1	6	1	19	3	14	7	7	48	17	32	56	18
	LDSTR UNIT	1	0	0	1	1	8	5	4	4	6	9	5	22	8	0
	INT ALU	6	3	0	2	10	9	5	0	3	40	7	11	13	6	62
	FLOAT ALU	3	0	4	0	3	8	7	5	3	7	16	7	29	13	7
	SFU ALU	2	1	0	0	0	1	2	0	1	0	1	4	7	3	0
Km	REGISTER FILE	3	9	23	6	0	30	36	30	38	33	109	92	53	195	76
	LDSTR UNIT	3	1	3	10	2	14	7	4	10	11	38	13	26	23	19
	INT ALU	46	0	7	0	32	7	4	2	7	216	11	14	5	21	437
	FLOAT ALU	24	4	13	12	11	51	31	34	37	58	115	93	54	88	113
	SFU ALU	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Mat	REGISTER FILE	25	23	39	27	16	96	56	53	58	56	199	134	293	287	116
	LDSTR UNIT	15	6	25	14	10	55	24	32	35	37	120	46	92	160	65
	INT ALU	2	0	4	6	64	13	10	9	13	306	377	21	26	51	557
	FLOAT ALU	9	8	5	7	3	24	30	15	24	48	86	76	28	62	72
	SFU ALU	0	0	0	0	0	0	2	4	0	0	0	0	12	0	0
Hs	REGISTER FILE	6	5	14	21	5	19	19	34	41	35	66	87	189	228	73
	LDSTR UNIT	2	2	5	6	1	2	0	5	11	8	9	28	33	33	11
	INT ALU	16	20	46	31	67	29	25	78	48	310	87	110	257	248	585
	FLOAT ALU	7	6	16	18	3	9	12	32	27	18	68	35	146	118	42
	SFU ALU	3	0	3	0	2	5	0	8	6	1	8	8	24	23	9
Box	REGISTER FILE	29	31	68	80	67	248	178	259	308	274	443	477	472	457	434
	LDSTR UNIT	0	0	0	2	0	0	0	0	0	0	3	2	0	1	2
	INT ALU	41	25	53	35	37	242	89	219	238	212	538	325	469	424	409
	FLOAT ALU	46	30	43	28	50	177	195	195	216	195	426	425	434	380	338
	SFU ALU	18	16	37	31	23	111	67	140	159	122	255	185	219	271	229
Conv	REGISTER FILE	45	48	55	206	47	260	220	245	246	137	325	391	481	546	491
	LDSTR UNIT	32	36	24	25	35	176	145	64	152	71	313	269	26	269	364
	INT ALU	7	7	6	42	4	20	30	28	79	5	22	33	26	146	20
	FLOAT ALU	97	89	71	47	126	416	301	563	419	197	662	543	785	636	854
	SFU ALU	0	0	0	4	0	5	0	0	2	0	7	1	0	2	2
Man	REGISTER FILE	247	240	250	181	226	978	1086	852	967	1082	1860	2396	1599	1444	1614
	LDSTR UNIT	3	2	1	0	2	1	11	1	2	16	3	13	10	5	6
	INT ALU	0	7	0	1	1	5	10	7	14	2	11	42	15	20	6
	FLOAT ALU	367	1432	371	404	361	2086	6466	1903	2216	1974	4084	13469	3633	2370	2928
	SFU ALU	183	606	186	180	216	939	3130	861	1002	928	1747	6440	1685	1076	1400
Avg.	REGISTER FILE	46.33	45.44	54.44	70.00	42.89	208.22	214.78	174.89	216.11	198.44	381.44	473.67	386.00	478.22	348.89
	LDSTR UNIT	7.44	6.11	7.56	9.44	6.67	30.33	25.56	14.56	26.89	19.89	58.56	48.33	27.11	63.67	56.56
	INT ALU	13.89	8.67	13.78	16.00	26.11	37.67	21.56	38.44	49.56	147.22	119.00	67.67	92.56	118.22	276.22
	FLOAT ALU	63.56	176.44	60.11	59.33	62.44	314.00	791.89	310.33	333.56	288.56	617.78	1,653.67	584.22	423.22	498.33
	SFU ALU	24.67	71.11	26.11	25.11	27.44	121.67	359.89	115.22	133.22	121.67	230.56	749.33	221.11	160.89	189.33

Table 3.2: The List of Effective Faults During the 25-times of Soft-error Reliability Experiments. Each Number Indicates How Many Effective Faults are Occurred on Each Components During the Experiments.

From Figure 3.9, we observe that the simulation result for the hotspot application with 50 faults injection does not match with the vulnerable period estimation result from Figure 3.8. Our algorithm should show a similar failure rate, but our algorithm shows improved failure rate. This is because of the random fault injection. Since we randomly inject faults on random components, the randomly generated faults may cause extreme behavior and generate abnormal results. In order to verify and show the effect from the extreme behavior of random fault injection, we performed an additional 25 times of simulation for the hotspot application with 50 fault injections and plotted the ratio of correct output for all the 50 times

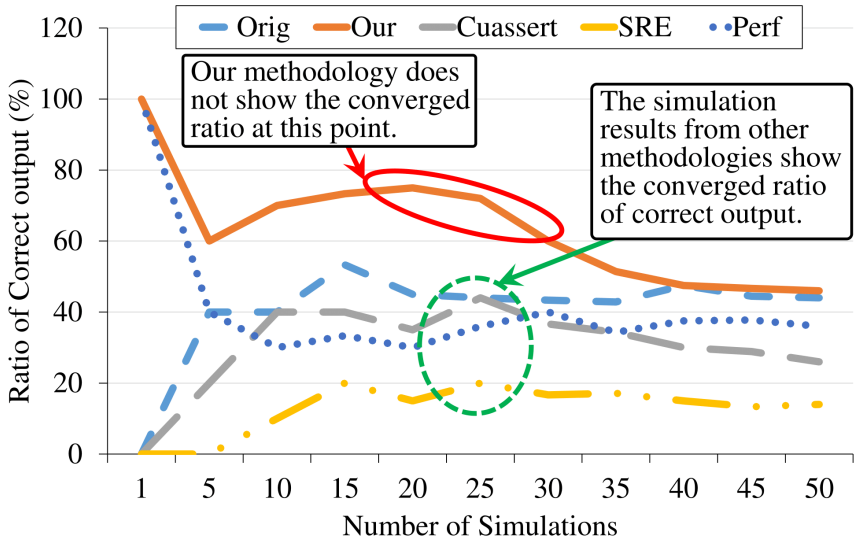


Figure 3.10: Ratio of Correct Output for Hotspot Application with 50 Faults Injection.

of simulation (Figure 3.10). From Figure 3.10, we may observe that our algorithm does not have converged in terms of ratio of correct output after 25 times of simulation. However, after 50 times of simulation, all the methodologies, including ours, have converged in terms of ratio of correct output. In addition, the results show that our algorithm has a similar failure rate to the original application and it matches with the vulnerable period estimation (Figure 3.8).

We observe that there is no correct output from the *Matrix multiplication* and *BoxFilter* applications when the fault injection rate is 100 faults/1M cycles. This is because the behavior of the *Matrix multiplication* and *BoxFilter* applications are sensitive to soft-errors. The failure rate of an application depends on two things: the timing behavior (vulnerable period) and the functional behavior (masking effect, propagation, and etc.). The applications default soft-error reliability may be decided based on the functional behavior. For example, lets assume we have two different applications that have the same vulnerable period: the application with multiplication instruction (MUL) and the application with addition instruction (ADD). Although they have the same vulnerable period, MUL may have a higher failure rate because the multiplication operation likely propagates the soft-error effect and

produces incorrect results. In Figure 3.9, the results from the *Matrix multiplication* and *BoxFilter* applications show examples of the above mentioned case. Both applications have a kernel function that consists of multiple loops with multiplication operation. Thus, the matrix multiplication and the box filter applications may have higher soft-error sensitivities than other applications. Therefore, with a 100 faults/1M cycles fault injection rate, the failure of the *Matrix multiplication* and *BoxFilter* applications is an expected outcome.

In Figure 3.9, for the *Srad* application, we can observe that the performance driven instruction scheduling shows better soft-error reliability than ours. This is one example of how the soft-error reliability can be achieved through performance improvement. Since the execution time of the *Srad* is really short, the performance improvement may reduce the time that the soft-error occurs during the execution time.

Figure 3.11 and Figure 3.12 show the performance and the power overhead. The performance and power overhead results are normalized with respect to the original application's performance and power consumption. In addition, in Figure 3.11 and Figure 3.12, the original application's performance and power consumption are described by red lines. As mentioned, since our algorithm sacrifices the performance to improve the soft-error reliability, our algorithm consumes less power compared to other methodologies. Note that [73] has less power consumption than our algorithm and it is because of the idle time from the additional memory operations. The performance overhead of our algorithm is 125% on average and the power overhead is -7% on average. This negative power overhead is an expected result because there is no change in the GPU hardware and our algorithm sacrifices the performance to improve the soft-error reliability. Therefore, since our algorithm consumes more time to execute the application, the less power consumption is an expected result.

From Figure 3.11, we can observe that our algorithm has a considerable amount of performance overhead (125%). However, without the *Mandelbrot* application, the performance overhead from our algorithm is only 47%, which is less than other two reliability method-

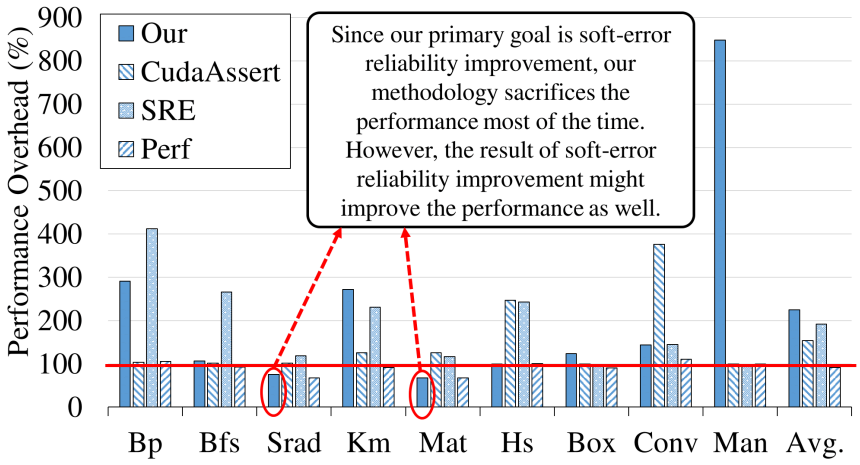


Figure 3.11: Performance Overheads Compared to [32], [73], and [52].

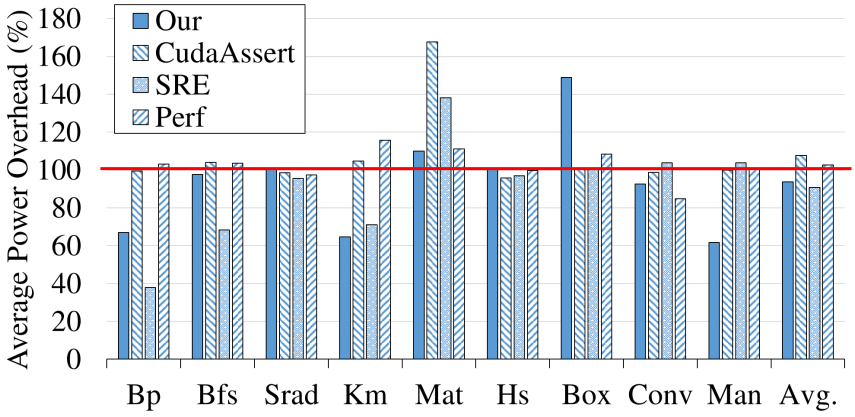


Figure 3.12: Average Power Consumption Overheads Compared to [32], [73], and [52].

ologies. The performance overheads from [32] and [73] are 53% and 92%, respectively. The major reason for the performance overhead is that 1) our algorithm does not have any additional source code; and 2) our algorithm tries to generate an instruction schedule and configuration in order to minimize the vulnerable period. *Srad* and *Matrix* applications are the examples that the proposed algorithm improves the performance. However, our algorithm shows a significant performance overhead in the *Mandelbrot* application because of the scheduling of memory access instructions and the change of the application's configuration to minimize the effect of the effective faults. These results show that the outcome of the soft-error reliability improvement is not always the performance improvement or the power

consumption improvement. Our algorithm may sacrifice performance to improve the soft-error reliability if it is necessary. In other words, the trade-off for the soft-error improvement is very application specific.

In summary, the experimental results imply that, as shown in Figure 3.5, the soft-error reliability is related to the detailed timing behavior of an application and the performance improvement does not guarantee the improvement in soft-error reliability.

3.6 Chapter Summary

In this chapter, we have proposed a novel GPU architecture-aware instruction scheduling algorithm in order to improve the soft-error reliability of the GPU-based system. The proposed instruction scheduling algorithm minimizes the vulnerable period and improve the soft-error reliability of an application. Based on the analysis of the state-of-the-art GPU architecture, we model the behavior of an application to estimate its soft-error vulnerability and generate the best instruction schedule and configuration. In addition, we have developed a fine-grained fault injection tool that is integrated with the state-of-the-art cycle-level GPU simulator to evaluate the proposed algorithm. We also have developed theorems and proofs to handle the application's control flows during the vulnerable period estimation. The experimental results show that our algorithm generates the instruction schedule within 8.13 seconds on average. Through our algorithm, the soft-error reliability is improved by 23% and 12% (up to 64% and 52%) compared to the state-of-the-art soft-error reliability improvement methodologies and performance aware instruction scheduling, respectively. Compared to the state-of-the-art methodologies, our algorithm has similar performance and power overheads in most cases while improving the soft-error reliability. In addition, the experimental results shows that the soft-error reliability of GPU is not related to the performance, but instead to the fine-grained timing behavior of an application.

Chapter 4

Run-time: Part I: Aging-aware GPU Workload Distribution Unit

In this chapter, we propose a low-overhead aging and aging-aware workload distribution unit for embedded GPUs under process variation. Due to the small feature size, chip aging and within-die parameter variations have been considered to be among the challenging problems for state-of-the-art processors, including GPUs. In order to deal with the process variation, the state-of-the-art multi-core processors improve their performance efficiency through core-level guardbanding that may use a different operating frequency for each core. Existing aging management techniques are based on the chip-level guardbanding, which assigns the same number of instructions to the cores that have the same aging status. However, in the presence of the process variation, existing aging management techniques have a limitation in minimizing the aging effect because each core has a different amount of stress for the same number of instructions. In order to tackle this problem, the proposed workload distribution unit considers the process variation and the current aging status together, and assigns a different number of instructions to clusters to minimize the aging effect in the presence of process variation.

4.1 Aging Model

Transistors age mainly when they are under stress (NBTI) and switch their state (HCI). NBTI mainly occurs on PMOS transistors and consists of two different phases. When a PMOS transistor is under stress, the threshold voltage, $|V_{th}|$, is increased due to traps, which are generated in the interface between the oxide layer and silicon/channel. After that, when the PMOS transistor is not under stress, some traps are filled and the $|V_{th}|$ is decreased (recovery phase). However, the $|V_{th}|$ shift cannot be fully recovered and the overall $|V_{th}|$ is increased over time. The $|V_{th}|$ shift depends on several aspects (i.e. supply voltage V_{dd}, duty cycle δ, usage, power-gating, etc.).

HCI mainly occurs on NMOS transistors. In NMOS transistors, the collision between the accelerated electrons and the gate oxide interface generates electron-hole pairs. After that, free electrons get trapped in the gate oxide layer and the $|V_{th}|$ is increased [96]. Since the electrons are accelerated when the NMOS transistor changes its state [96], the total amount of $|V_{th}|$ shift is very sensitive to the number of state transitions. In short, HCI depends upon the switching activity.

In this dissertation, we have employed a micro-architectural level aging analysis framework of [96] to estimate the impact of the NBTI and the HCI at the same time. In [96], the V_{th} shift at time t due to the NBTI is estimated by the following equation:

$$\Delta V_{th}(t) \leq \int_0^1 A_N u(V_{dd}) \frac{(v(T_B) \cdot \delta_B \cdot \delta_e \cdot t_m)^n}{w(\delta_B \cdot \delta_e, T_B, t)^{2n}} \mathrm{d}\delta_e \tag{4.1}$$

with

$$u(V_{dd}) = (V_{dd} - V_{th}) \cdot \exp((V_{dd} - V_{th})/E_0 \tag{4.2}$$

$$v(T) = \xi_4 \cdot \exp(-E_a/kT) \tag{4.3}$$

$$w = 1 - \left(1 - \frac{\xi_1 + \sqrt{\xi_3 \cdot v(T) \cdot (1 - \delta(t)) \cdot t_m}}{\xi_2 + \sqrt{v(T) \cdot t}}\right)^{\frac{1}{2n}} \tag{4.4}$$

where δ_B represents a duty cycle of a transistor inside a block B, δ_e represents the effective duty cycle, T_B represents the temperature of a block B, and t_m describes the sampling period. A_N, E_0, E_a, n, and ξ_i are technology dependent constants.

The V_{th} shift at time t due to the HCI is estimated by the following equation:

$$\Delta V_{th}(t) = A_H \cdot \sqrt{\alpha_{avg,B}} \cdot u(V_{dd}) \cdot v(T_B) \cdot \sqrt{\alpha_B \cdot f \cdot t} \tag{4.5}$$

with

$$u(V_{dd}) = \exp((V_{dd} - V_{th})/E_1) \tag{4.6}$$

$$v(T) = \exp(-E_a/kT) \tag{4.7}$$

where $\alpha_{avg,B}$ describes the average switching activity of all gates in a block B, α_B the activity factor of a block B, and f represents the clock frequency. A_H and E_1 are technology dependent constants.

From the ΔV_{th} due to NBTI and HCI (Equations 4.1 and 4.5), the amount of the relative change in delay at time t may be estimated by the following equation:

$$\Delta^{rel} d(t) = \left(1 - \frac{\Delta V_{th}(t)}{V_{dd} - V_{th}(t_0)}\right)^r - 1 \tag{4.8}$$

where r is a technology dependent constant. Definitions of the parameters used in the equations and detailed description of aging analysis framework can be found in [96].

4.2 Process Variation Model

We consider leakage power and frequency variations. In order to generate the process variation map of the target platform, we have employed the state-of-the-art process variation model used in [38, 107, 140]. To apply the process variation model, the GPU in the target platform is partitioned based on the area description in the GPGPU-Sim [8] and GPUWattch [71] simulator. After that, the entire GPU surface is modeled as a fine two-dimensional grid ($N_{chip} \times N_{chip}$).

In the presence of process variation, the value of the process parameter at grid cell (i, j) can be modeled as a Gaussian random variable with mean μ_p and standard deviation σ_p [107]. In [107], a correlation coefficient, $\rho_{i,j,k,l}$, is used to describe the process parameters at two different grid points. By using the model in [140], the spatial correlation between two grid points are represented by the following equation:

$$\rho_{i,j,k,l} = e^{-\alpha\sqrt{(i-k)^2+(j-l)^2}} \ \forall i,j,k,l \in [1, N_{chip}] \tag{4.9}$$

In [140], the maximum frequency of core C_i with critical path $CP(C_i)$ is represented by the following equation:

$$f_i = \alpha \times \min_{x,y \in S_{(CP,i)}} (1/\theta_{x,y}) \tag{4.10}$$

where α is the technology dependent constant and the $S_{(CP,i)}$ represents a set of grid points corresponding to the critical path, $CP(C_i)$. The total power consumption of C_i is estimated by the following equation:

$$p_{(i,j,k)} = p^{dyn}_{(i,j,k)} + \sum_{(x,y) \in C_i} p^{leak}_{x,y} \times e^{V_{th}\theta_{x,y}/V_T} \tag{4.11}$$

where $p^{dyn}_{(i,j,k)}$ is the dynamic power consumption of C_i when it executes a thread τ_k of an application A_j. $p^{leak}_{x,y}$ represents the nominal leakage power at grid point (x, y). $V_T = KT_i/q$

represents the thermal voltage where T_i represents the temperature of C_i. In addition, V_T captures the temperature dependence of leakage power. A more detailed description of the process variation model can be found in [107] and [140].

4.3 Aging-Aware Resource Management on Embedded GPUs under Process Variation

The proposed aging-aware resource management technique consists of the host part and the device part. The host part includes the aging-aware cluster formation algorithm and the process variation-aware instruction distribution algorithm. These algorithms operate on the device driver. Right before the host launches a kernel function, the host collects the critical path delay information from on-chip delay monitors [19, 124]. Using that delay information, the aging-aware cluster formation algorithm sorts the cores and creates groups of cores. The aged clusters are selected for power-gating based on the resource utilization information, which is generated in design-time. After the power-gating, in order to evenly distribute the stress across the embedded GPU, the proposed instruction distribution algorithm estimates the instruction distribution ratio by using the critical path delay and the process variation information. Next, the host sends the following information to the GPU: cluster configuration (including operating frequency), power-gating configuration, and the instruction distribution ratio. Then, the GPU is configured with the information from the host. At last, the host launches a kernel function and the GPU starts its computation. Figure 4.1 shows the overview and the high-level flow of the proposed aging-aware workload management on the embedded GPU under the process variation.

The degradation of the components in a multi/many-core processor is closely related to stress and power management. Moreover, the aging gap between the clusters will become

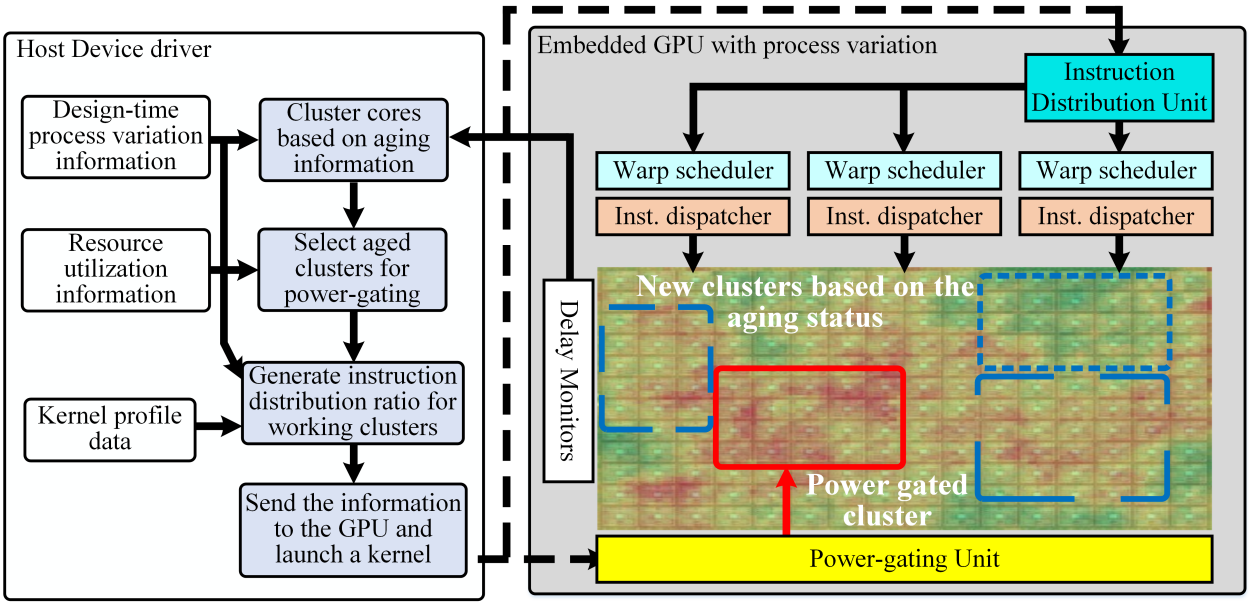


Figure 4.1: High-level Flow and Overview of the Proposed Technique.

larger over time due to the process variation, cluster-level guardbanding, and unbalanced stress distribution. In order to minimize the effects of the process variation and the cluster-level guardbanding, the proposed aging-aware cluster formation algorithm (re)configures the clusters by using the current aging information. Before the host launches a kernel function, the host obtains the current aging information for all the cores in the GPU through the delay monitors. Then, the host sorts the cores based on the aging information in descending order and groups the cores to create the clusters. Thus, each cluster has cores that have similar degradation levels and minimum aging variations. The sorting and clustering process do not need to consider the process variation at this point because the amount of degradation is the result of the process variation. After that, each cluster sets its operating frequency by finding a core with minimum operating frequency. Since the entire GPU may not be required to execute the kernel function, after the clustering, the proposed aging-aware cluster formation algorithm selects some degraded clusters for power-gating based on the GPU resource utilization information, which is generated in design-time.

Algorithm 3 shows the behavior of our aging-aware cluster formation algorithm. The

Algorithm 3: Aging-Aware Cluster Formation.

```
   Input: Component type T, Delay information D, Resource utilization for current kernel R_K, Warp size N_warp
   Output: Clustered computational components C, Operating frequency F
1  Function WarpFormation (T, D, R_K, N_warp)
2  begin
3      C ← ∅;
4      F ← ∅;
       // Cluster the computational components
5      foreach t ∈ T do
6          C_sort ← SortComponents(t, D) ; // Sort computational components based on the critical path delay
7          while C_sort ≠ ∅ do
8              C_cluster ← GetClusterForWarp(C_sort, N_warp);
9              C_sort.remove(C_cluster);
10             C.add(C_cluster) ; // Add the cluster to the cluster information
       // Select the cluster for the power-gating
11     foreach t ∈ T do
12         C_working ← ∅;
13         R_D ← ∅;
14         C_T ← GetClusterType(t, C) ; // Get the clusters based on the type
15         SortCluster(C_T) ; // Sort the clusters with the aging information
16         p_T ← GetPowerGateInfo(t, R_K) ; // Get the resource utilization information
17         foreach c ∈ C_T do
18             if c.GetOrder() > p_t then
19                 PowerGateCluster(c) ;
20             else
21                 C_working.add(c) ;
22         F.AddOperatingFreq(C_working) ;
23     return [C, F] ;
```

proposed algorithm takes the following information as input: the types of computational components T, delay information D, resource utilization for current kernel R_K, and warp size N_{warp} (Line 1). The computational components are sorted based on the current delay information (Line 6). After that, based on the warp size, N_{warp}, the components are clustered and the cluster information is updated (Lines 7-10). Next, the proposed algorithm selects clusters for power-gating based on the resource utilization information (Lines 11-21). The remaining clusters are selected as the working clusters (Lines 17-21). The operating frequency is selected for working clusters (Line 22). At the end, the clustering information, which includes power-gating, and the operating frequencies are returned (Line 23).

4.3.1 Process Variation-aware Workload Distribution Algorithm

Due to the process variation, the same number of instructions cause a different amount of degradation for each cluster. In order to evenly distribute the stress across the GPU, each working cluster should process a different number of instructions. The proposed process variation-aware workload distribution algorithm estimates the instruction distribution ratio between the active clusters.

For each kernel, the proposed algorithm assigns the same number of instructions for each working cluster and gets the aging estimation. Each working cluster will have a different aging status due to the different operating frequencies, which are caused by the process variation. Then, the proposed algorithm gets the average from the aging estimation of working clusters and sets this average as a desired aging status after executing the kernel function. The amount of stress for each cluster can be obtained by subtracting the current aging status from the desired aging status. By using the amount of stress and the process variation information, the instruction distribution ratio is estimated.

Algorithm 4 describes the behavior of the proposed process variation-aware instruction distribution algorithm. This algorithm takes the working cluster information $\mathbf{C}_{work}$, the process variation information **PV**, and the number of instructions N_{inst} as inputs. At the beginning, the algorithm evenly distributes the instructions to working clusters (Line 5). The algorithm estimates the aging status of each working cluster using the number of instructions n_{even_inst} and the process variation information (Lines 6 - 7). Then, the average aging is estimated based on the total amount of aging (Line 8). After estimating the average aging information, the algorithm estimates the amount of stress for each working cluster (Line 10). Based on this amount of stress and the process variation, the algorithm decides the number of instructions for each cluster (Line 13). Finally, the algorithm returns the instruction distribution ratio for each working cluster (Line 14).

Algorithm 4: Process Variation Aware Instruction Distribution Ratio.

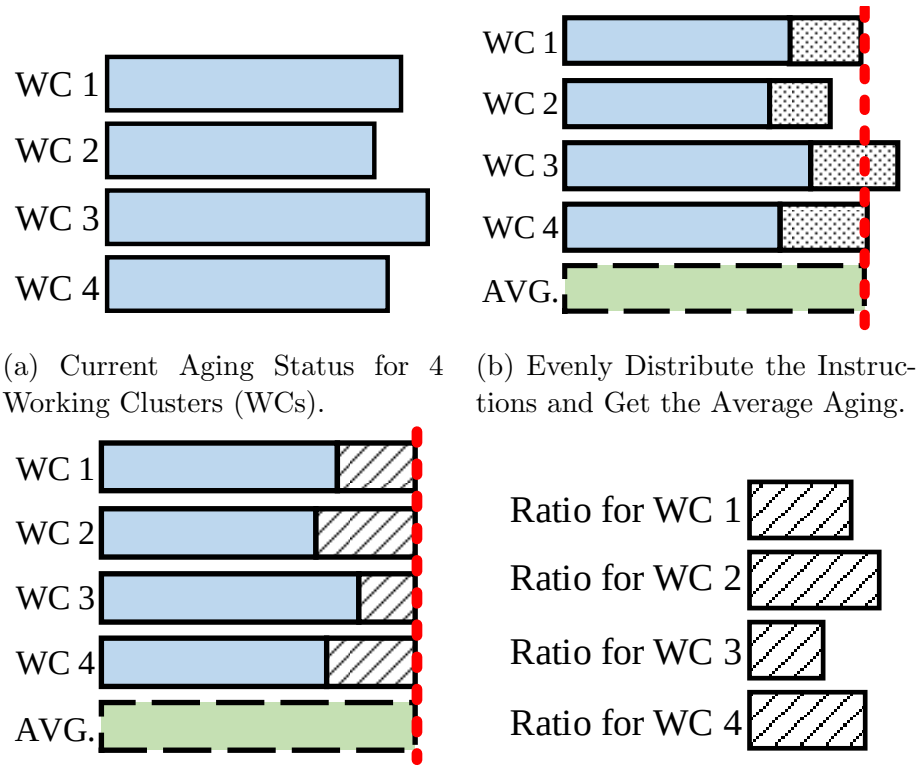
Input: Working Clusters $\mathbf{C}_{work}$, Process Variation information **PV**, Number of Instructions N_{inst}
Output: Instruction Distribution Ratio $\mathbf{R}_D$

```
1  Function GetInstDistRatio (C_work, PV, N_inst)
2  begin
3      R_D ← ∅;
4      AvgAge ← ∅;
       // Evenly distribute instructions
5      n_even_inst ← N_inst / sizeof(C_work);
6      foreach C ∈ C_work do
7          AgeSum ← AgeSum + GetAging(C, PV) ; // Get total aging information with evenly distributed
           instructions
       // Get the average aging and finalize instruction distribution ratio
8      AvgAge ← AgeSum / sizeof(C_work);
9      foreach C ∈ C_work do
10         ST ← AvgAge - GetStatus(C) ;
11         R ← GetInstNum(ST, PV) ;
12         R_D.add(R);
13     GenerateRatio(R_D);
14     return R_D ;
```

Figure 4.2 shows an example of estimating the instruction distribution ratio with 4 working clusters. Figure 4.2(a) shows the current aging status of 4 working clusters. In Figure 4.2(b), the algorithm evenly distributes the instructions to the working clusters. However, due to the process variation, each cluster has a different amount of aging. Then, the algorithm estimates average aging, which is represented by a dashed line box. The next step is shown in Figure 4.2(c). Using the average aging and the current aging information, the amount of stress for each working cluster is obtained. At last, as shown in Figure 4.2(d), the instruction distribution ratio for each working cluster is generated.

4.3.2 Instruction Distribution Unit

The proposed instruction distribution unit incorporates the existing warp scheduler and instruction dispatch unit to balance the stress across the GPU. The host configures the instruction distribution unit using the instruction distribution ratio before launching a kernel function. After the host launches a kernel function, the instruction distribution unit controls the behavior of the warp scheduler and the instruction dispatch unit to evenly distribute



(a) Current Aging Status for 4 Working Clusters (WCs).

(b) Evenly Distribute the Instructions and Get the Average Aging.

(c) Obtain the Amount of Stress for Each Working Cluster Based on the Average Aging.

(d) Finalize the Instruction Distribution Ratio.

Figure 4.2: Example for Estimation of Instruction Distribution Ratio with 4 Working Clusters (WCs).

the stress over the GPU. When an instruction is ready, the instruction dispatch unit sends the instruction to the first available cluster and increases the corresponding instruction distribution counter. When all the instruction distribution counters reach their limits, then the instruction distribution unit resets the counters and sends the instruction to the first available cluster.

Algorithm 5 describes the behavior of the proposed instruction distribution unit. The algorithm takes the following information as inputs (Line 1): the component types T, clustering information $\mathbf{C}$, operating frequencies $\mathbf{F}$, instruction distribution ratio $\mathbf{R}_D$, and instructions $\mathbf{I}$. The host configures the clusters using the clustering information $\mathbf{C}$ and the operating frequencies (Line 4). Then the algorithm power-gates the non-working clusters and sets the

Algorithm 5: Algorithm For Instruction Distribution Unit.

```
   Input: Component types T, Clustering information C, Operating frequencies F, Instruction distribution ratio R_D,
          Instructions I
 1 Function InstDistributionUnit (C, R_D, I)
 2 begin
       // Power-gate non-working cluster
 3     foreach t ∈ T do
 4         ClusterConfig(C, F) ;
 5         C_T ← GetClusterType(t, C) ; // Get the clusters based on the component type
 6         foreach c ∈ C_T do
 7             if c.is_working() = False then
 8                 PowerGateCluster(c) ;
 9             else
10                 SetInstDistCnt(c, R_D);
       // Distribute the instructions based on the ratio during run-time
11     foreach i ∈ I do
12         t ← GetType(i) ;
13         R ← GetRatioType(R_D, I) ;
14         c ← GetNextAvailCluster(t, R) ;
15         W ← GetCorrespndingWarpSch(c) ;
16         W.sendInst(i) ;
17         IncDistributionCnt(c) ;
18         if IsDistCntFull() = True then
19             ResetDistRatio();
```

instruction distribution ratio for working clusters (Lines 6-10). After the configuration, the host launches a kernel function and the instructions are fetched (Line 11). For each instruction, the algorithm gets the instruction type and the corresponding instruction distribution ratio (Lines 12-13). Next, the instruction is sent to the first available working cluster and the algorithm increases the instruction distribution counter (Lines 14-17). At last, the algorithm resets the instruction distribution counter if all the instruction distribution counters reach their limit (Lines 18-19).

Figure 4.3 shows an example of the instruction distribution unit. The host configures the GPU using the clustering and power-gating information and the instruction distribution ratio. In the figure, Cluster 3 is selected for power-gating. The instruction distribution ratio for Clusters 1, 2, and 4 are *4:3:3*, respectively. After the configuration, the host launches a kernel and the instruction distribution unit starts fetching and distributing the instructions based on the instruction distribution ratio.

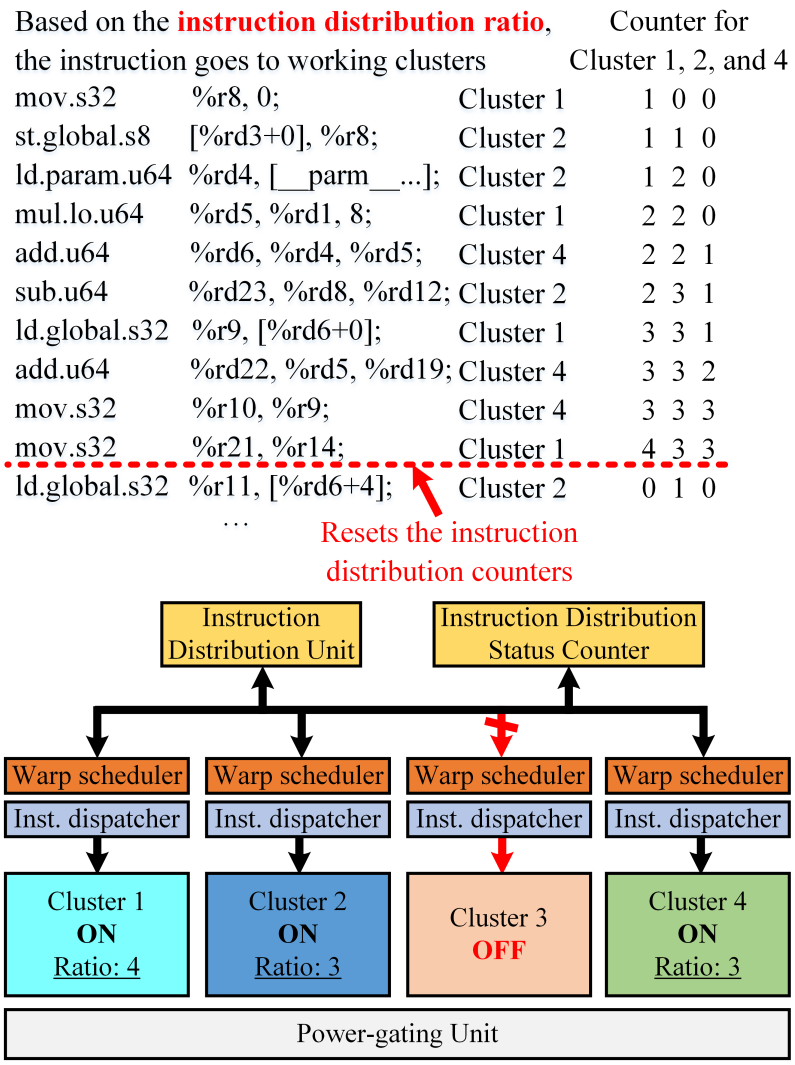


Figure 4.3: Example of the Instruction Distribution Unit.

4.4 Evaluation

4.4.1 Experimental Setup

In order to evaluate our technique, we have used GPGPU-Sim [8], which is a configurable cycle-level simulator. GPGPU-Sim includes a configurable and extensible GPU energy model called GPUWattch [71]. We configure GPGPU-Sim to have a similar configuration with the embedded GPU in the NVIDIA's Tegra TK1. The configuration details are in Table 4.1.

Table 4.1: GPGPU-Sim Configuration for NVIDIA's Tegra TK1

Name	Value	Name	Value
# of SM	1	L1 Cache	16 KByte
# of Cores	192	Shared Mem	48 KByte
# of Register	65536	Core Clk Freq.	876 MHz

We have selected various benchmark applications from NVIDIA's Compute Unified Device Architecture (CUDA) toolkit to evaluate our technique with different computational workloads for the embedded GPUs. The selected benchmark applications and their configurations are shown in Table 4.2.

Table 4.2: Benchmark Applications and Configurations.

Application	Abbreviation	Grid Size	Block Size
BilateralFilter	BL	48	256
BinomialOptions	BN	4	128
BlackScholes	BLK	480	128
ConvolutionSeparable	CONV	128	64
FastWalshTransform	FT	64	256
Montecarlo	MC	128	128
Reduction	RD	32	256
SobelFilter	SO	1024	64

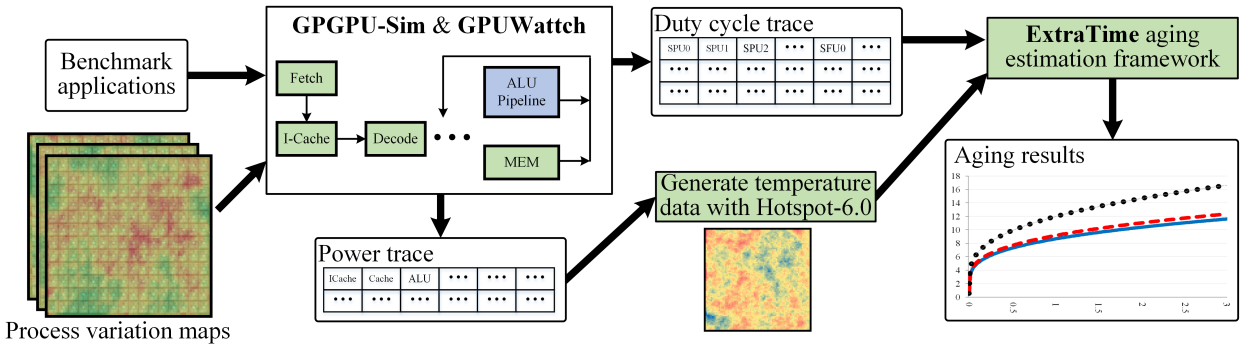


Figure 4.4: Overview of the Experimental Setup.

We have generated 50 different process variation maps by using the area information in GPGPU-Sim and the process variation model from [38, 65, 113]. During the experiment, we have extracted duty cycle information and power traces for all the benchmark applications

and process variation maps. Hotspot [48] simulator is used to estimate the embedded GPU temperature. Then, we feed the duty cycle information and the embedded GPU temperature to the aging estimation framework in [96]. Figure 4.4 shows the overview of our experimental setup.

During the experiments, the following state-of-the-art aging management techniques are used to evaluate our technique:

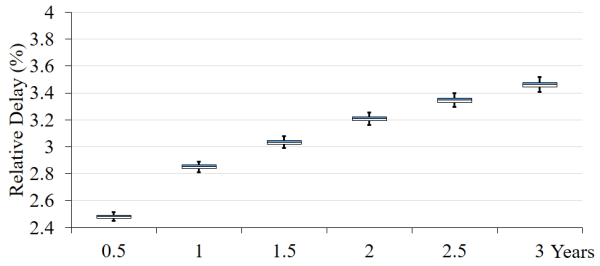
- **Compiler-based technique (Comp)** [75]: this techniques used a JIT compiler-based technique to maximize the lifetime of the GPU. Depending on the aging status, healthy kernel function is generated to transfer the workload from the aged cluster to the healthy cluster.
- **Even distribution algorithm (Even)** [69]: this technique assigns the same number of instructions to the clusters that have the same age.
- **Original application (Orig)**: the unmodified applications are used to set the baseline results.

4.4.2 Experimental Results

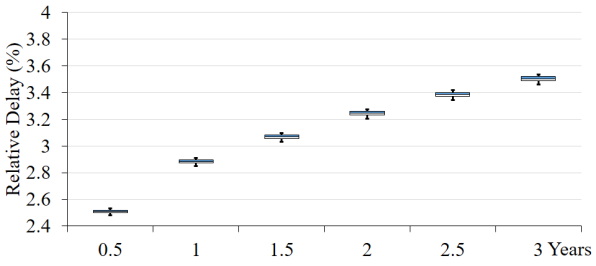
We measured the algorithm execution time with an Intel Core^{TM} i7 Quad-core processor at 3.5 GHz. Our aging-aware workload management algorithm requires 0.413 milliseconds on average (standard deviation is 0.0182 milliseconds).

Aging Improvement Comparison:

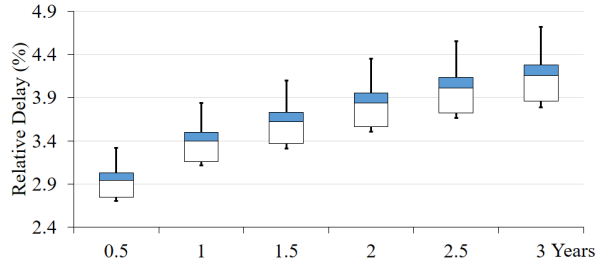
We evaluate our technique with 50 different process variation maps to demonstrate the effectiveness of our technique. During the experiment, for each process variation map, the relative critical path delay after 3 years is collected. Figure 4.5 shows the aging trace of SO application over the 3 years. The aging results are generated for each process variation map and the collected results are plotted in Figure 4.5. The results show that our technique and



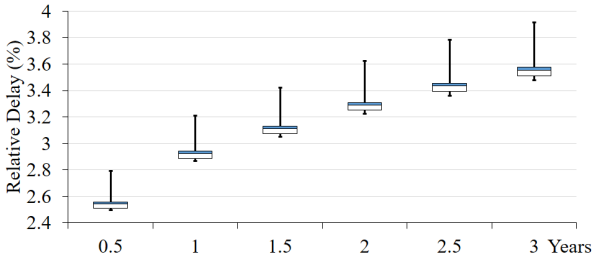
(a) Relative Delay with Our Technique.



(b) Relative Delay with Even Distribution.



(c) Relative Delay with Compiler-based Technique.



(d) Relative Delay with Original Application.

Figure 4.5: Relative Delay for SobelFilter Application with 50 Different Process Variation Maps: a) Our Technique, b) Even Distribution [69], c) Compiler-based Technique [75] and d) Original Application.

the even distribution algorithm can further improve aging compared to the compiler-based technique. Moreover, the compiler-based technique shows varying aging improvements for different process variation maps, whereas our technique and the even distribution technique consistently improve the aging of embedded GPUs. This is because the compiler-based technique only disables the aged clusters and sends all the workloads to the healthy clusters.

Figure 4.6 shows the average and standard deviation of relative delays after 3 years for the 50 process variation maps. Our technique improves the aging by 3% and 2.9% on average compared to the original applications and compiler-based technique, respectively (up to 3.75% and 19.53%). In addition, compared to the even distribution technique, our technique improves the aging up to 2.35%. From Figure 4.6, we can observe that our technique has a small standard deviation and shows aging improvement for all the applications. Since the other techniques do not consider the randomness of the process variation, their optimizations do not maximize the lifetime of embedded GPUs.

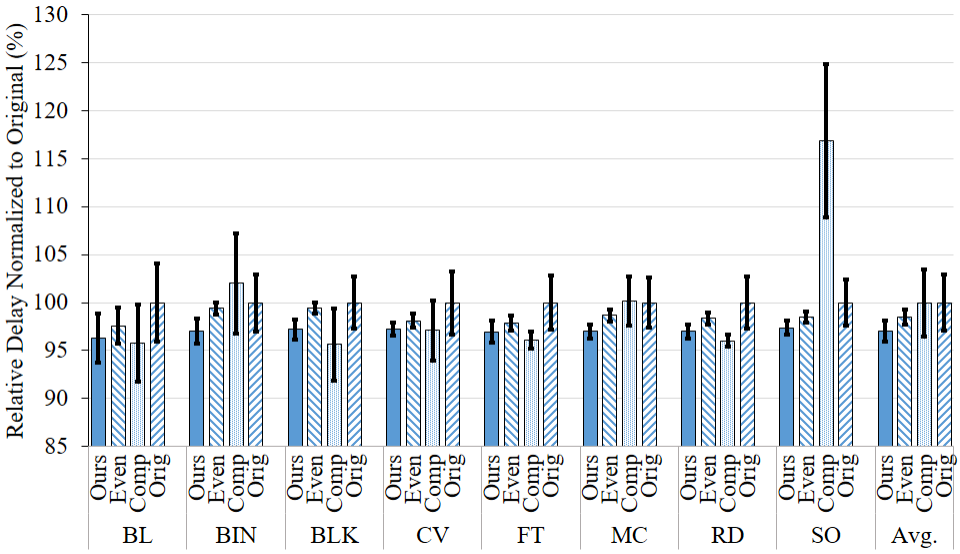


Figure 4.6: Normalized Average Relative Delay After 3 Years Compared to the Even Distribution [69], the Compiler-based Technique [75], and Original Applications.

Success Rate Comparison:

We collect the success rate of aging improvement for all the process variation maps to show the impact of randomness of the process variation. Figure 4.7 shows the success rates of aging improvement. The average success rate of aging improvement for our technique, the even distribution, and compiler-based technique are 97.25%, 75.25%, and 70%, respectively. In the figure, we can observe that the even distribution and the compiler-based technique have lower success rates compared to our technique. This is because the process variation and the cluster-level guardbanding may cause a different amount of stress for the same number of instructions and increase the randomness in the system state. Since this random behavior is not considered by the other techniques, they have lower success rate compared to our technique.

We measure the relative standard deviation of aging of SP/SFU units to observe the impact of the above mentioned randomness. Figure 4.8 shows the average relative standard deviation for all the 50 process variation maps. We can observe that the aging is well balanced across

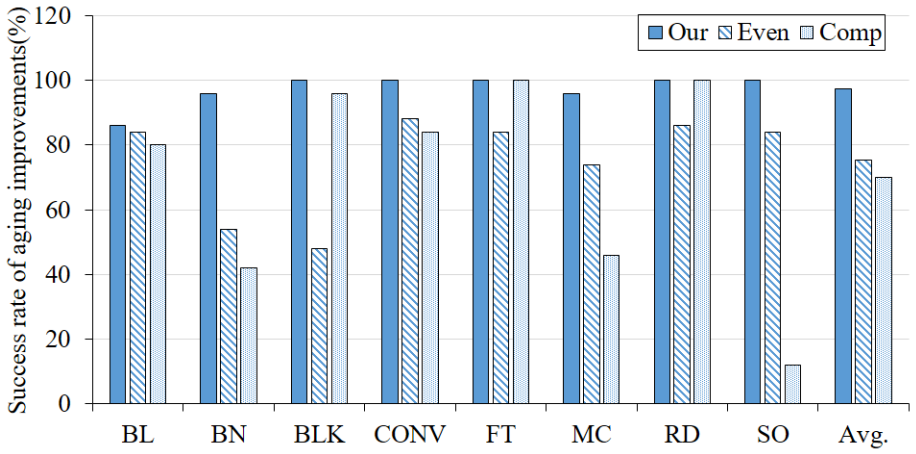


Figure 4.7: Success Rate of Aging Improvement for 50 Different Process Variation Maps Compared to the Even Distribution [69] and Compiler-based Technique [75].

the embedded GPU with our technique. However, the other techniques do not show the balanced aging distribution. This is because other techniques' optimization may not capture the randomness of the process variation. The results in Figure 4.7 and 4.8 imply that other techniques may worsen the aging of embedded GPU without proper consideration for process variation.

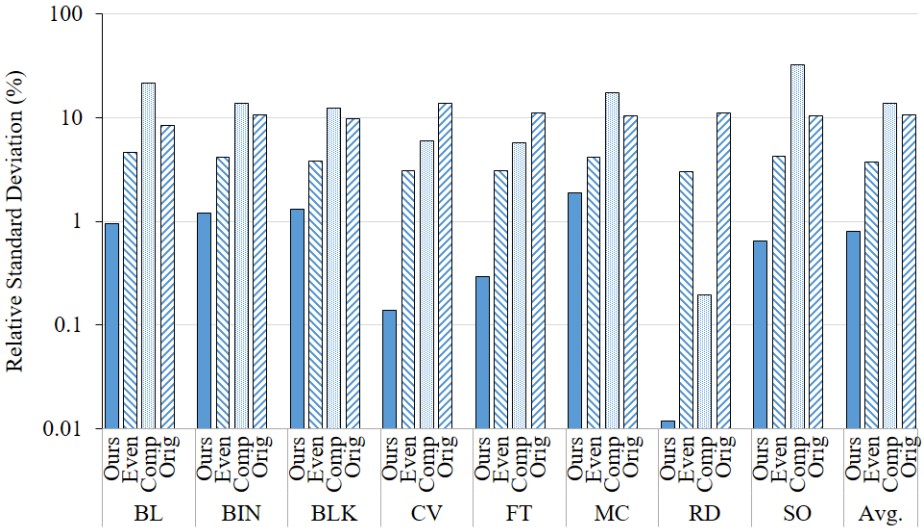


Figure 4.8: Average of Relative Standard Deviations of SP/SFU Units Across the Embedded GPU Compared to the Even Distribution [69], the Compiler-based Technique [75], and Original Applications.

Performance Overhead Analysis:

Figure 4.9 shows the normalized performance overheads compared to the even distribution and the compiler-based technique [75]. The results show that the performance overhead of our technique is 1.05% on average (up to 7.92%), whereas the compiler-based technique has an overhead of 42.87% on average (up to 359.33%). The even distribution technique requires 0.5% less execution time compared to the original one. The performance overhead of our technique is mainly caused by the instruction dispatch unit, which updates/resets the instruction distribution count and sends the instructions to the working cluster. However, the compiler-based technique has to use redundant workloads to transfer the workload information from the degraded cluster to the healthier cluster. Moreover, in order to transfer the workload between the clusters, the compiler-based technique uses atomic operations, which are quite expensive. Due to the atomic operation, all the read/write operations for workload transfer are serialized and cause a non-negligible performance overhead.

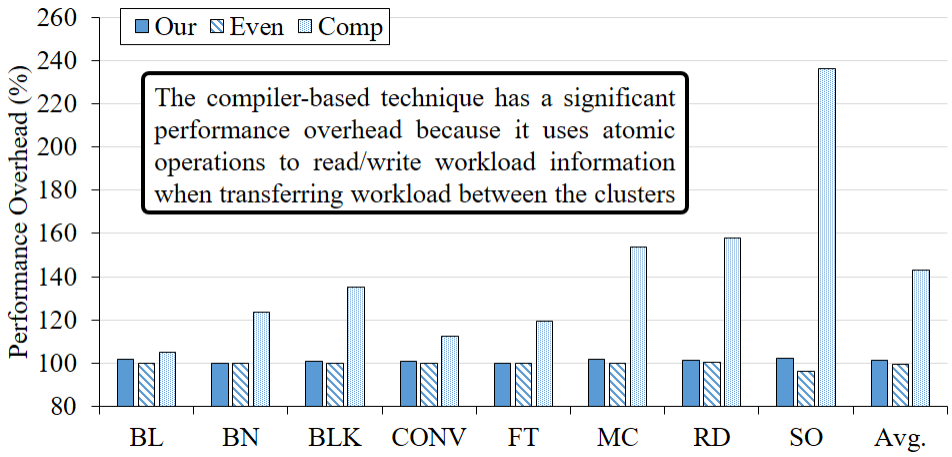


Figure 4.9: Normalized Average Performance Overhead for Our Technique, the Even Distribution [69], and the Compiler-based Technique [75].

Power Consumption Overhead Analysis:

Figure 4.10 shows the average power consumption overheads for our technique, the even distribution, and the compiler-based technique [75]. All the results are normalized to the

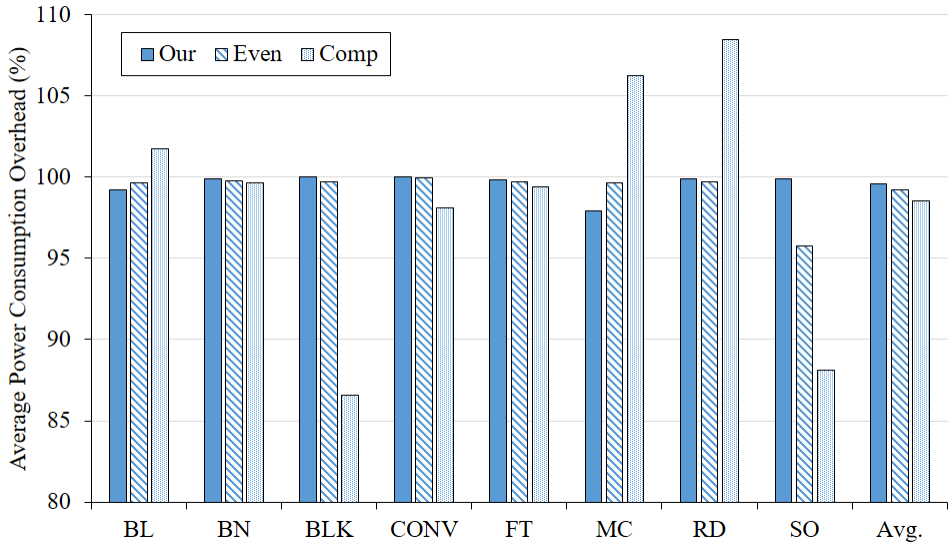


Figure 4.10: Normalized Average Power Consumption Overhead for Our Technique, the Even Distribution [69], the Compiler-based Technique [75], and Original Applications.

original applications' average power consumption. On average, our technique uses 0.5% less power than the original applications. This is because our technique needs additional time to balance the stress while maintaining the same throughput. Moreover, our technique does not require any code modification. The even distribution technique and the compiler-based technique consume 0.75% and 1.45% less power, respectively. This is because the compiler-based technique's atomic operations cause a noticeable amount of idle time to read/write workload information.

The aforementioned experimental results imply that our workload management technique can maximize the lifetime of the embedded system while satisfying its timing requirements. In addition, the compiler-based technology [75] would have increased soft-error susceptibility because its atomic operations cause noticeable performance overheads while it maintains the same throughput. However, since our approach does not require any additional workload, our approach will not exhibit this limitation.

4.5 Chapter Summary

In this chapter, we propose an aging-aware workload management unit for embedded GPUs in the presence of process variation. The proposed workload management unit incorporates the existing warp scheduler and instruction dispatcher to balance the stress across the embedded GPUs with process variation. The warp formation and workload distribution algorithms generate information to (re)configure the cluster and balance the stress across an embedded GPU. Then, the host configures the GPU with the results from the algorithms before it launches a kernel function. After that, the GPU distributes the instructions based on the instruction distribution ratio. The simulation results show that our technique improves the GPU aging over 95% of cases whereas the state-of-the-art compiler-based technique improves the GPU aging in 72.25% of cases. Moreover, compared to the compiler-based technique, our workload management unit reduces the performance overhead by 40% while achieving almost the same GPU aging improvement. These experimental results indicate that our technique may minimize the aging effect of embedded GPUs in the presence of process variation. Moreover, our workload management unit has less soft-error susceptibility than the state-of-the-art compiler-based technique because of the lower performance overhead.

Chapter 5

Run-time: Part II: Timing-aware GPU Workload Scheduling Framework

In this chapter, we propose a novel run-time scheduling framework to handle the dynamic behavior of the event-driven applications. The existing GPU workload scheduling frameworks do not have enough flexibility for increasing number of event-driven applications. This is because in the existing scheduling frameworks: 1) only temporal preemption is considered and 2) one application occupies the GPU at a time. In order to solve that problem and implement both temporal and spatial preemption, the proposed scheduling framework partitions the GPU workloads into sub-workloads and generates sub-workloads launch sequences to handle the dynamic behavior of the event-driven applications. We demonstrate the capability and novelty of our framework compared to the existing scheduling frameworks with realistic benchmark applications and with different execution scenarios.

5.1 System Model

Our target GPU-based embedded system has a GPU with a total of N_{tot} SMs. For a given set of event-driven applications, $\mathbf{A}=\{A_1, A_2, ..., A_i\}$, the execution time information for the applications, and the priority of the applications are provided. The goal of this work is to: 1) partition the application kernels to multiple sub-kernels and 2) generate sub-kernel launch sequences in such a way that satisfies $Res(A_i) < T_{dead(i)}$, where $Res(A_i)$ and $T_{dead(i)}$ represent the response time and the deadline of the application A_i, respectively.

Figure 5.1 shows an overview of the proposed application scheduling framework. During run-time, event-driven applications are launched by the user or the system itself. We assume that the system has no prior knowledge of the applications. The application includes the following information: 1) deadlines, 2) priority, and 3) execution time information (see Section 5.2 for more details). Based on the current status of the system and the applications, the proposed workload splitter partitions the application kernels into multiple sub-kernels. After that, sub-kernel launch sequences are generated by the proposed GPU execution schedule generator.

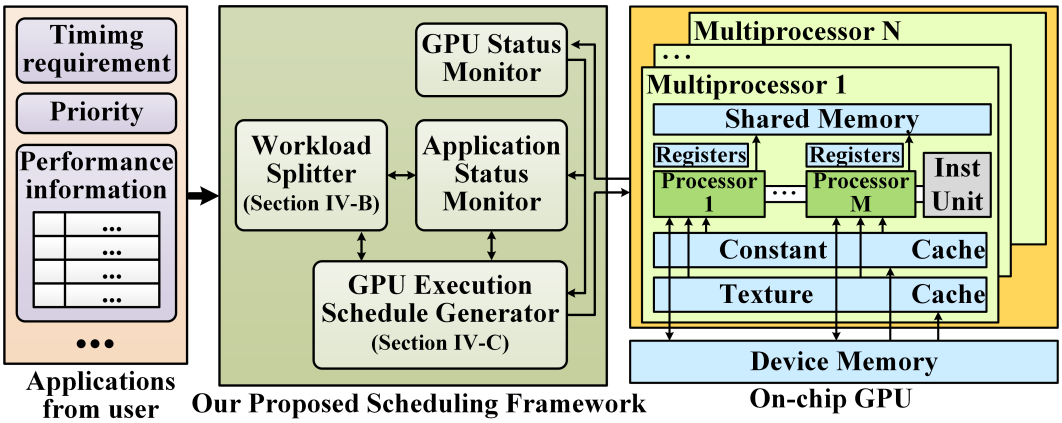


Figure 5.1: Overview of the Proposed Run-time Scheduling Framework on a GPU-based Embedded System.

5.2 Event-driven Application Model

A set of event-driven applications, $\mathbf{A}=\{A_1, A_2, ..., A_i\}$, will be randomly injected into the target GPU-based embedded system. The application processes a set of input data. The following two example scenarios describe the behavior of two event-driven applications in the system in Figure 1.1:

- When the system launches the traffic sign recognition application, each image sensor captures a video, with different frame rate, or image. The frame rate of each image sensor depends on several factors such as the image sensor specification, distance from the object, and driving direction of the vehicle. After that, the system launches the applications to process these frames. These frames are processed with exactly the same function such as *motion estimation*, *motion tracking*, and/or *object detection*.
- While driving, secure vehicular communication is required when the system communicates with various internal/external systems such as sub-systems of the vehicle, and other vehicles. Although the size of data varies, the exact same *data encryption/decryption* applications are used to encrypt/decrypt the set of input communication data in order to achieve secure communication.

The previously mentioned set of behaviors is very common among event-driven applications. An application needs to process a set of inputs, which is denoted by $\mathbf{I} = \bigcup_{n=1,...,N} \{I_n\}$ where N represents the total number of inputs. Note that the value of N may vary. If a CPU is used, the same function is launched N times to process a set of inputs $\mathbf{I}$. However, these multiple function launches are easily transformed into *data-level parallelism* which is the specialization of GPUs. Work in [76] has implemented throughput-oriented (data-parallel) GPGPU applications of the previously mentioned scenarios.

The application is a single-threaded application and has at least one throughput-oriented kernel function to process a set of input data. Moreover, since the execution time of the

application is dominated by these kernel functions, the kernel function is considered as a basic unit of execution[1].

Each kernel function is represented by a tuple, $<\mathbf{K}_i, P_i, T_{dead(i)}, \mathbf{E}_i(k, n_{SM})>$. $\mathbf{K}_i$ represents a set of kernel functions $\{K_{(i,1)}, K_{(i,2)}, \ldots, K_{(i,j)}\}$. P_i and $T_{dead(i)}$ represent the priority and the deadline of the application A_i, respectively. $\mathbf{E}_i(K, n_{SM})$ represents the execution time of the kernel K with n_{SM} SMs. In this dissertation, we assume that the execution of the kernels in one application is sequential. In addition, the internal dependencies that implement *task-level parallelism* are not considered. In other words, the host cannot launch the kernel $K_{(i,k)}$ until the previous kernel $K_{(i,k-1)}$ completes its operation. This sequential execution of the kernels is caused by the limitation of the GPU which is discussed in 3.1. We assume that the makespan of kernel functions is estimated during design-time by using a measurement based heuristic method and the estimations are provided as an input to our scheduling framework. The work in [12] is one of the estimation techniques that can be used to estimate the makespan of a kernel.

5.3 Run-time Scheduling Framework for GPU-based Real-time Embedded Systems

5.3.1 Temporal and Spatial Preemption

The proposed scheduling framework implements two types of preemptions: temporal and spatial preemption. As mentioned in Section 3.1, it may be impossible to suspend and resume the kernel function after the GPU starts executing the kernel function. In other

[1]Although the concept of the kernel is similar with the concept of the task, in GPGPU computing, the term *kernel* is the general name to refer a workload (or task) that is handled by the GPU. Moreover, since this dissertation is about to schedule the GPU workloads, we use the term *kernel* to represent the basic unit of execution.

words, the interruption of a kernel that is running on the GPU is not allowed. Therefore, temporal preemption of the GPU is related to the time that the host sends a kernel to the GPU. Since the GPUs are multi-core processor, it may be possible that each running application uses different amount of GPU resources, spatial preemption is related to the number of SMs for each running application. In this dissertation, the *temporal* and the *spatial* preemptions are defined as follows:

***Definition* 1.** An application A is **temporally preempted** by an application B if the following conditions are satisfied:

- Application A and B are waiting for the GPU resources in the host at the same time.
- $T^{B}_{init} < T^{A}_{init}$, where T^{App}_{init} represents the time that the host sends a kernel in an application App to the GPU to use the GPU resources.
- When the host sends the kernel to the GPU, the priority of the application B is higher than the priority of the application A.

***Definition* 2.** An application A is **spatially preempted** by an application B if the following conditions are satisfied:

- Application A and B are running on the GPU at the same time.
- For given time t, $n^{A}_{SM}(t) < n^{B}_{SM}(t)$, where $n^{App}_{SM}(t)$ represents the number of SMs for the application App at time t.
- At the time of GPU resource assignment, the priority of the application B is higher than the application A.

5.3.2 Workload Splitter

The proposed workload splitter partitions a kernel into multiple sub-kernels based on the current status of an event-driven application and a target GPU-based embedded system. In addition, the configurations for the sub-kernels are generated to control the mapping between

the kernel and the SMs in the GPU and represent the GPU resource (re)allocation. Thus, the workload splitter implements the *spatial preemption*.

The response time of an application $Resp(A_i)$ may be represented as:

$$Resp(A_i) = L_{init} + \sum_{n=0}^{N_k-1} \Big[L_{lnch} + E(K_{(i,n)}) \Big] + D_{tran} \tag{5.1}$$

where L_{init} represents the delay between the start of the application A_i and the assignment of the GPU resources to A_i. L_{lnch} represents the kernel launch overhead and $E(K_{(i,n)})$ represents the execution time of n^{th} kernel of the application A_i. D_{tran} represents the data transfer latency.

Since the kernels may be executed with different number of GPU resources, the execution time of each kernel would, therefore, depend on the number of assigned GPU resources. Moreover, since data transfer latency between the host and the device is approximately zero, Equation 5.1 is re-written as:

$$Resp(A_i) = L_{init} + \sum_{n=0}^{N_k-1} \Big[L_{lnch} + E(K_{(i,n)}, n_{SM(i)}) \Big] \tag{5.2}$$

Hence, to satisfy the deadline, the total response time of an event-driven application has to be less than, or equal to, the deadline of the application:

$$T_{dead(i)} \geq L_{init} + \sum_{n=0}^{N_k-1} \Big[L_{lnch} + E(K_{(i,n)}, n_{SM(i)}) \Big] \tag{5.3}$$

Since multiple applications may be concurrently running on the target GPU-based embedded system, it is critical to estimate the amount of GPU resources that satisfy Equation 5.3. Therefore, the amount of GPU resources can be estimated by:

$$T_{avail(i)} = T_{dead(i)} - T_{curr} \geq R(S(A_i), n_{SM}) \tag{5.4}$$

where T_{curr} represents the current time. $T_{dead(i)}$ represents the deadline of A_i (Equation 5.3), and $S(A_i)$ represents the current status of A_i. Equation 5.4 estimates the available time $T_{avail(i)}$, to run the application. The $R(S(A_i), n_{SM})$ represents the estimated remaining execution time of A_i with n_{SM} SMs based on the current status of A_i.

The current status of the application $S(A_i)$ is represented by the tuple $<n_{tb(i,k)}, n_{r(i)}>$, where $n_{tb(i,k)}$ describes the number of processed thread blocks in the current kernel and $n_{r(i)}$ describes the number of remaining kernels. Since the kernel workload granularity, on the target GPU-based embedded system, is a thread block, the remaining execution time of the current kernel with n_{SM} SMs is:

$$R_c(K_{(i,k)}, n_{SM}) = \frac{|K_{(i,k)}| - n_{tb(i,k)}}{|K_{(i,k)}|} \times E(K_{(i,k)}, n_{SM}) \tag{5.5}$$

The expected execution time of the remaining kernels is provided as:

$$R_r(K_{(i,k)}, n_{r(i)}, n_{SM}) = \sum_{n=n_{r(i)}}^{N_k-1} \Big[L_{lnch} + E(K_{(i,n)}, n_{SM}) \Big] \tag{5.6}$$

By using Equation 5.5 and 5.6, the remaining execution time of an event-driven application with n_{SM} SMs is derived as:

$$R(A_i, n_{SM}) = R_c(K_{(i,k)}, n_{SM}) + R_r(K_{(i,k)}, n_{r(i)}, n_{SM}) \tag{5.7}$$

The proposed workload splitter estimates the smallest n_{SM}, for each running application, that would satisfy Equation 5.4. Note that there may be additional time slot between the estimated completion time of the application and its deadline. After estimating n_{SM} for each running application, the proposed workload splitter will adjust n_{SM} based on the following three possible cases:

- **The GPU is idle**: In this case, there is no application kernel running on the GPU. Hence, the workload splitter partitions the application kernel into multiple sub-kernels,

based on n_{SM}, and sends them to the GPU execution schedule generator.

- **The GPU has available resources**: In this case, at least one application kernel is running on the GPU. The workload splitter determines whether the currently available GPU resources are enough to meet the deadlines. If the available GPU resources are not enough, the proposed workload splitter adjusts n_{SM}, for the running application kernel, and partitions the application kernel based on the adjusted n_{SM}. After that, the partitioned application kernels are sent to the GPU execution schedule generator.
- **The GPU is fully occupied by higher priority applications**: In this case, the workload splitter partitions the kernel into the smallest granularity possible and waits until the GPU has the available resources. Once the GPU has the available resources, then the rest of the process is the same as case 2).

After adjusting n_{SM}, the workload splitter generates the configurations for the sub-kernels by using the n_{SM}. Then the sub-kernels are generated with the configuration and the current status of the application which is obtained by the application status monitor (see Section 5.3.4 for a complete example).

Algorithm 6 shows the pseudo code of the workload splitter. In Lines 7-15, the workload splitter reallocates the GPU resource when there is a request for GPU resource reallocation. The loop in Lines 6-15 is the main body of the workload splitter. In Line 7 the workload splitter checks whether the previous sub-kernel is consumed or not. In Line 8 the amount of assigned GPU resources is retrieved. After that, sub-kernels are created and added to the list in Lines 10-13. Depending on the GPU resource status, the GPU resource reallocation request is sent in Lines 14-15. A set of sub-kernels is returned in Line 16.

The function *ResRealloc*() in Line 5 has a complexity of $O(nlog(n) + 2n)$. The function *GetrequiredRes*() in Line 8 has a complexity of $O(N_{tot})$, thus, the loop in Lines 6-15 has

Algorithm 6: Algorithm to Split the Application Kernels.

Input: A set of application kernels $\mathbf{K} = \{K_{(o,l)}, K_{(p,m)}, \ldots, K_{(q,n)}\}$;
Output: A set of splitted sub-kernels $\mathbf{k} = \{k'_{(o,l)}, k'_{(p,m)}, \ldots, k'_{(q,n)}\}$;

```
1  Function SplitKernel (K)
2  begin
       /* Variable initialization.                                          */
3      k ← ∅;
       /* Check the resource reallocation request flag.                     */
4      if ReallocReqStatus() = true then
5          ResRealloc() ;
6      foreach K ∈ K do
           /* Create new sub-kernel from the kernel K if it is required.    */
7          if K.NeedNewSubKernel() = true then
8              n = GetRequiredRes(K);
9              if n = 0 then
                   /* Create smallest sub-kernel if the kernel does not have any GPU
                      resource.                                              */
10                 k' ← SplitSmallestKernel(K) ;
11             else
                   /* Create sub-kernel with assigned GPU resource.         */
12                 k' ← SplitKernel(K, n) ;
13             k'.push(k') ;
               /* If the kernel does not have enough GPU resource, set the resource
                  reallocation flag.                                         */
14             if K.HasEnoughRes() = false then
15                 SetReallocReq();
16     return k' ;
```

a complexity of $O(N_{tot} \times n)$. Consequently, the overall complexity of the proposed workload splitter algorithm is given by $O(|nlog(n) + 2n + N_{tot} \times n|) = O(nlogn)$.

Algorithm 7 describes the procedure to reallocate the GPU resources. The variables are initialized in Lines 3-4. A list of application kernels is created and the amount of GPU resources is estimated for resource reallocation in Line 5. The loop in Lines 6-18 is the main part of the GPU resource reallocation. The target application kernel is fetched from the list in Line 7. The amount of GPU resources to meet the deadline (Equation 5.4) is estimated in Line 8. If there is enough amount of GPU resources, the workload splitter assigns the required amount of GPU resources in Lines 9-12. On the other hand, if the amount of GPU resources is not enough, the workload splitter allocates all the remaining GPU resources and

Algorithm 7: Algorithm to Reallocate GPU Resources for Each Running Application Kernels.

```
   Input: Application kernels for resource reallocation K;
1  Function ResRealloc (K)
2  begin
       /* Initialize the variables.                                          */
3      N_realloc ← ∅ ;
4      K_re ← ∅ ;
       /* Create resource reallocation list.                                 */
5      [K_re, N_realloc] ← CreateReallocList(K) ;
       /* Main loop for resource reallocation.                               */
6      while K_re.size() > 0 and N_realloc > 0 do
           /* Estimate the amount of GPU resources to meet the response time
              requirement.                                                   */
7          K ← K_re.front() ;
8          N_req = GetNewResource(K) ;
           /* Determine the amount of GPU resources for the application kernel based on
              the current available GPU resources and the required amount of GPU
              resources.                                                     */
9          if N_realloc ≥ N_req then
10             K.SetResource(N_req);
11             K.HasEnoughRes(true);
12             N_realloc = N_realloc - N_req;
13         else
               /* Use all available GPU resources.                           */
14             N_req = N_realloc;
15             K.SetResource(N_realloc);
16             K.HasEnoughRes(false);
17             N_realloc = N_realloc - N_req;
18         K_re.popfront() ;
```

marks the application kernel as not having enough GPU resource in Lines 13-17. After that, the target application kernel is removed from the list in Line 18.

The function $CreateReallocList()$ in Line 5 has a complexity of $O(nlog(n)+n)$ The function $GetNewResource()$ in Line 8 has a complexity of $O(N_{tot})$, thus, the outer-loop in Lines 6-18 has a complexity of $O(N_{tot}\times n)$. Consequently, the overall complexity of the GPU resource reallocation algorithm is given by $O(|nlog(n) + n + N_{tot}\times n|) = O(nlogn)$.

Algorithm 8 describes the procedure to create the list of the application kernels for the GPU resource reallocation. In Line 5, the running application kernels are sorted based on their priorities. If the applications have the same priority then they are sorted according to the

Algorithm 8: Algorithm to Create Reallocation List.

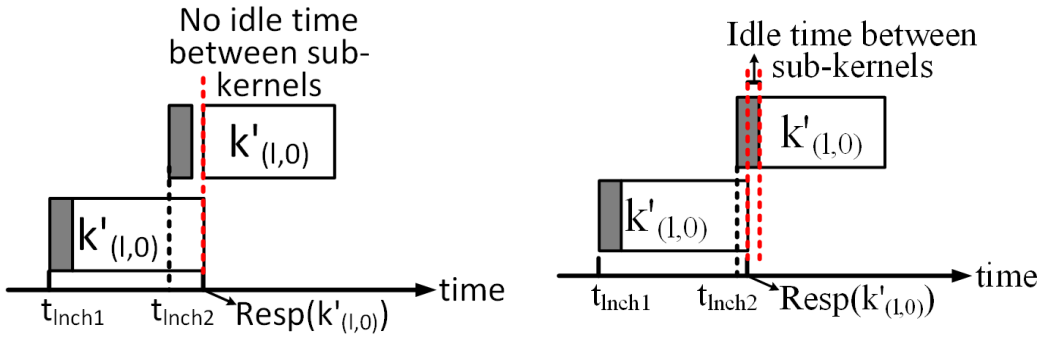
```
   Input: A set of application kernels K;
   Output: A set of kernels for GPU resource reallocation K_re,
           Amount of GPU resource for resource reallocation N_re;
1  Function CreateReallocList() begin
       /* Resource reallocation variable initialization.                    */
2      ForceLowerApp = false; /* Flag to include lower priority applications. */
3      N_re ← false;
4      K_re ← ∅;
       /* Sort applications based on the priority.                          */
5      K_sort = PrioritySort(K) ;
       /* Sort same priority with EDF algorithm.                            */
6      CreateOrderBasedOnEDF(K_sort) ;
7      foreach K ∈ K_sort do
8          if ForceLowerApp = false then
9              if K.HasEnoughRes() = false then
10                 ForceLowerApp ← true ;
11                 K_re.push(K);
12                 N_re = ReturnResource(K);
13         else
14             N_re = ReturnResource(K);
15             K_re.push(K)
16     return [K_re, N_re] ;
```

Earliest Deadline First (EDF) algorithm in Line 6. The loop in Lines 7-15 is the main body of the reallocation list creation. The current GPU resource status of the running application kernels is checked in Line 9-12. If the application kernel does not have enough resource, the application kernel and all the lower priority application kernels will be added to the GPU resource reallocation list in Lines 10-12 and Lines 13-15. The algorithm returns the GPU resource reallocation list and the amount of the GPU resource for reallocation in Line 16. The sort functions in Lines 5-6 have a complexity of $O(nlog(n))$ and the loop in Lines 7-15 has a complexity of $O(n)$. Consequently, the overall complexity of creation of the reallocation kernel list algorithm is given by $O(|nlog(n) + n|) = O(nlogn)$.

5.3.3 GPU Execution Schedule Generator

The statuses of the submitted sub-kernels are tracked by the GPU status monitor (see Figure 5.1). The GPU status monitor triggers the GPU execution schedule generator based on the GPU status. Then, the GPU execution schedule generator selects the sub-kernels, which are partitioned by the workload splitter, based on the priority and the deadline of the application. After that, the selected sub-kernels are submitted to the GPU hardware queue and the GPU resource (re)allocation is applied. Thus, the GPU execution schedule generator implements the *temporal preemption*.

During run-time, whenever the GPU has available resources, the GPU hardware selects one sub-kernel from its hardware queue and assigns the GPU resources to process the sub-kernel. Thus, in order to maximize the performance, and the number of applications that meet the deadlines, the sub-kernels must be submitted to the GPU hardware queue before the GPU has idle resources.



(a) Sub-kernel Launch Without the GPU Idle Time.

(b) Sub-kernel Launch With the GPU Idle Time.

Figure 5.2: Examples for Launching Sub-kernels.

Figure 5.2(a) shows two sub-kernels launch without any GPU idle time. In the figure, the shaded boxes represent the kernel launch overhead. The time difference between the response time of the first sub-kernel, $Resp(k'_{(l,0)})$, and the launch time of the second sub-kernel, t_{lnch2}, is greater than the kernel launch overhead. Therefore, the second sub-kernel waits in the

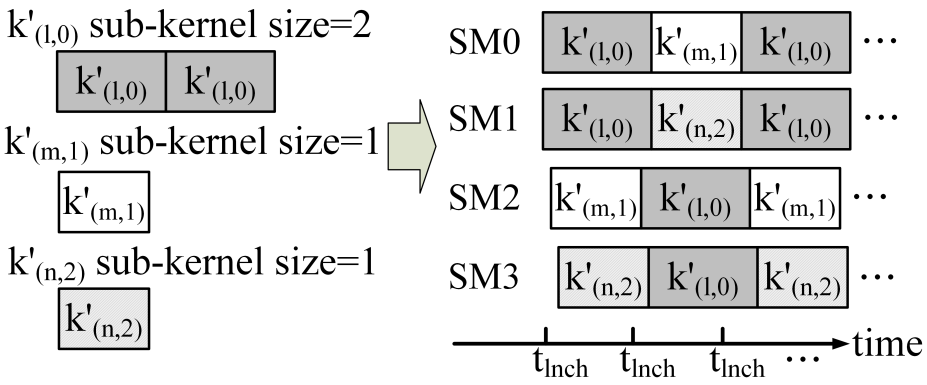


Figure 5.3: Example Sub-kernel Launches on the GPU Execution Schedule Generator.

GPU hardware queue and could start its operation right after the first sub-kernel completes its operation. On the other hand, in Figure 5.2(b), the kernel launch overhead is greater than the time difference between $Resp(k'_{(l,0)})$ and t_{lnch2}. In this case, the second sub-kernel could not wait in the GPU hardware queue. However, the overall response time of the application is larger than the case in Figure 5.2(a) because the GPU is idle between the sub-kernel executions. The sub-kernel launch time limit, T_{lnch}, is represented by the following equation:

$$t_{lnch} \leq Resp(k'_{(i,k)}) - L_{lnch} \tag{5.8}$$

Since multiple applications may be executed on our target GPU-based embedded system, t_{lnch} would need to be estimated based on the minimum response time among the currently running sub-kernels (Equation 5.9).

$$t_{lnch} \leq min(Resp_{k' \in S}(k')) - L_{lnch} \tag{5.9}$$

Figure 5.3 shows the example behavior of the GPU execution schedule generator. In this example, the target GPU has four SMs. Three different application sub-kernels are sent from the workload splitter. Note that Equation 5.9 estimates the sub-kernel launch time limit, t_{lnch}. During run-time, it is possible that the difference between the response time of the application and the deadline of the application is large enough to process starved low priority sub-kernels. As described in Section 5.3.2, when no GPU resource is assigned to the application kernel, the workload splitter partitions the application kernel into the smallest

Algorithm 9: Algorithm for GPU Execution Schedule Generator.

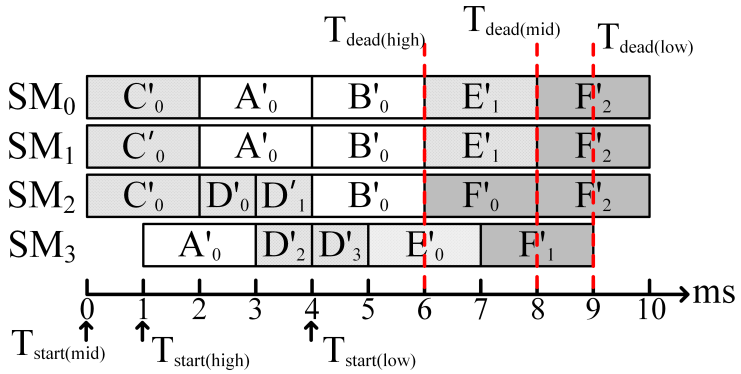
```
   Input: Application sub-kernels k',
          Starved application sub-kernels k'_starve;
1  Function GpuExecScheGen (k', k'_starve)
2  begin
       /* Sort applications based on the priority                    */
3      k'_sort = PrioritySort(k') ;
       /* Sort same priority with EDF algorithm                      */
4      CreateOrderBasedOnEDF(k'_sort) ;
5      k"_sort = PrioritySort(k'_starve) ;
6      CreateOrderBasedOnEDF(k"_sort) ;
       /* Main loop for resource reallocation                        */
7      foreach k' ∈ k'_sort do
8          T_dead ← GetDeadline(k');
9          T_resp ← GetRespTime(k');
10         T_avail ← T_dead - T_resp;
11         T_exec ← GetExecTime(k"_sort.front());
12         if T_avail > T_exec then
13             LaunchSubKernel(k"_sort.front());
14             k"_sort.popfront();
15         else
16             [k", T_exec ] ← GetMinExecKernel(k"_sort);
17             if T_avail > T_exec then
18                 LaunchSubKernel(k");
19                 k"_sort.remove(k");
20         LaunchSubKernel(k');
21         k'_sort.remove(k');
```

granularity possible.

Algorithm 9 shows the procedure of the GPU execution schedule generator. The GPU execution schedule generator sorts the sub-kernels based on the priority and the deadline of the application in Lines 3-4. After that, in Lines 5-6, the GPU execution schedule generator sorts the sub-kernels that do not have the GPU resources. The loop in Lines 7-21 is the main part of the GPU execution schedule generator. The available time for the low-priority sub-kernel T_{avail} is estimated in Lines 8-10. If there is a low priority sub-kernel that may complete its operation within T_{avail}, the GPU execution schedule generator launches the low priority sub-kernel in Lines 12-19. After that, the sub-kernel k' is launched in Line 20. The function $CreateOrderBasedOnEDF()$ in Line 4 and 6 has a complexity of $O(nlog(n))$. The loop in lines 7-21 has a complexity of $O(n \times N_{tot})$. Consequently, the overall complexity of the

App	$T_{dead(i)}$	P_i	|K|	Kernel info	
				Name(N_{TB})	Exec time
High	6ms	High	2	A(3)	2ms/TB
				B(3)	2ms/TB
Mid	8ms	Medium	3	C(3)	2ms/TB
				D(4)	1ms/TB
				E(3)	3ms/TB
Low	9ms	Low	1	F(5)	2ms/TB

(a) Input Application Information.



(b) Sub-kernel Execution Flow on the GPU.

Figure 5.4: Complete Working Example of the Proposed Scheduling Framework.

proposed GPU execution schedule generator algorithm is given by $O(|2nlog(n) + N_{tot} \times n| = O(nlogn)$.

5.3.4 Example of the Scheduling Framework

Figure 5.4 shows a complete working example of our scheduling framework. In this example, it is assumed that the GPU has four SMs. Figure 5.4(a) shows the information about the input applications while Figure 5.4(b) shows the behavior of the GPU. The three applications *High*, *Mid*, and *Low* are launched at 1 ms, 0 ms, and 4 ms, respectively.

At 0 ms, after the application *Mid* starts its operation, our scheduling framework generates and submits the sub-kernel C'_0 to the GPU. The size of C'_0 is the same as the original kernel

C, which is 3, due to the idle state of the GPU. At 1 ms, the application *High* starts its operation. Due to the higher priority than the application *Mid*, n_{SM}^{High} and n_{SM}^{Mid} are modified by the proposed workload splitter where n_{SM}^{High} and n_{SM}^{Mid} represent the number of SMs for the application *High* and *Mid*, respectively. The sub-kernel size for the application *High* is the same as original kernel due to the highest priority. The sub-kernel size for the application *Mid* is decreased to 1 which is the same as the number of remaining SMs in the GPU. After that, the sub-kernel A'_0 is submitted to the GPU. At this point, since only 1 SM is available, a part of the A'_0 starts running on the GPU.

At 2 ms, after the sub-kernel C'_0 completes its operations, the rest of the A'_0 use the GPU resources. At the same time, sub-kernels D'_0, D'_1, and D'_2 are generated and submitted to the GPU. At 4 ms, the workload splitter creates sub-kernels B'_0 and D'_3 from the kernel B and D, respectively. After that, the GPU execution schedule generator sends these sub-kernels according to the priority of the application. The application *Low* starts its operation, but it cannot get the GPU resources because all the GPU resources are occupied by the application *High* and *Mid*. At 5 ms, the sub-kernel D'_3 completes its operation, the sub-kernel E'_0 is generated and submitted to the GPU by our scheduling framework.

At 6 ms, the application *High* completes its entire operation. Since there are no higher priority applications, the application *Mid* may use more GPU resources, therefore n_{SM}^{Mid} is increased to 2. Since there is no higher priority applications, the application *Low* starts using the GPU resources. After that, sub-kernels E'_1 and F'_0 are generated and submitted to the GPU. The size of E'_1 and F'_0 are 2 and 1, respectively. At 7 ms, the sub-kernel F'_1 is generated from the kernel F and submitted to the GPU.

At 8 ms, the application *Mid* completes its entire operation. Since there is no other running applications, n_{SM}^{Low} is increased to 3. Then the sub-kernel F'_1 is generated and submitted to the GPU. After 2 ms, the application *Low* completes its entire operation at 10 ms. This example shows that the sub-kernel size is dynamically changed during run-time depending

on the status of the system (i.e. number of running applications, number of available GPU resources, and so on).

5.4 Evaluation

5.4.1 Experimental Setup

We have extensively evaluated our framework by comparing it to several state-of-the-art existing frameworks. We have built the simulator that represents our target GPU-based embedded system (as described in Section 3.1) and the simulator is assumed to resemble Nvidia's Tegra mobile embedded system. We also assumed that our target GPU has a total of 13 SMs and the off-chip memory is shared by the CPUs and the GPU. In order to evaluate our scheduling framework with realistic event-driven applications, during the experiments, we generate a set of workloads by randomly selecting applications from the Rodinia Benchmark Suite [18]. During the workload generation process, if the number of applications in the Rodinia Benchmark Suite is smaller than the number of applications for a workload, then same application has been selected multiple times. Table 5.1 shows the *dwarves*, the domains of the benchmark applications, and the problem sizes for the benchmark application. *Dwarves* are common computation and communication pattern of GPGPU applications [5].

Before the evaluation, we have assigned priorities to the benchmark applications based on the application domains and *dwarves*. Since most of the image processing applications, on an embedded system, require real-time behavior, image processing applications are classified as high priority application. In addition, a *HotSpot* application is also classified as a high priority application because applications like *HotSpot* could be used to estimate the current system status. The benchmark applications which have simple behavior (i.e. graph traversal,

Table 5.1: Rodinia Benchmark Suite [18].

Application Name	Dwarves	Domains	Problem Size
Leukocyte	Structured Grid	Medical Imaging	640x480 pixels/frame
Heart Wall	Structured Grid	Medical Imaging	656x744 pixels/frame
CFD Solver	Unstructured Grid	Fluid Dynamics	200k elements
HotSpot	Structured Grid	Physics Simulation	512512 data points
Back Propagation	Unstructured Grid	Pattern Recognition	65536 input nodes
Kmeans	Dense Linear Algebra	Data Mining	204800 data points, 34 features
Breadth-First Search	Graph Traversal	Graph Algorithms	1M nodes
SRAD	Structured Grid	Image Processing	20482048 data points
Streamcluster	Dense Linear Algebra	Data Mining	65536 points, 256 dimensions
PathFinder	Dynamic Programming	Grid Traversal	1k× 1k grid
Gaussian Elimination	Dense Linear Algebra	Linear Algebra	256×256 matrix
B+ Tree	Graph Traversal	Search	1M nodes

vector computation) are classified as low priority. Remaining benchmark applications are classified as medium priority. After classifying the priority of the application, we have assigned the deadlines to the benchmark applications based on the application domains, the *dwarves*, and the execution time of the applications. The classification results (and deadlines) for our experiments are as follows:

- **High priority**: Leukocyte (50ms), Heart Wall (50ms), HotSpot (1000ms), and SRAD (30ms);
- **Medium priority**: Back Propagation (30ms), PathFinder (20ms), Kmeans (100ms), and Streamcluster (2400ms);
- **Low priority**: Breadth-First Search (15ms), B+ Tree (400ms), Gaussian Elimination (1700ms), and CFD Solver (70000ms);

Note that the system may have a low-priority application that does not have a specific deadline. We have set the long deadline (70000ms) for the *CFD solver* application to represent the low-priority application that does not have a specific deadline.

For each application, the execution time information with different number of SMs needs to be provided as an input. Existing GPU-based embedded system does not have 13 SMs

and may be saturated with higher number of injected applications. However, the technology roadmap projects that even in a GPU-based mobile embedded system the number of SMs will increase. Moreover, we focus on the problem of scheduling throughput-oriented applications on the GPU. Therefore, in our simulator, we assumed that the system has enough computational capability to concurrently handle multiple applications. In order to obtain the execution time information with different number of SMs, we have used a NVIDIA Tesla K20m graphics card which has a similar GPU architecture (Kepler architecture) with NVIDIA Tegra K1 processor.

5.4.2 Number of the Applications Meeting Deadlines

In order to evaluate the proposed scheduling framework, we have randomly selected a number of applications and injected them into the simulator. Delays between the target applications are also randomly chosen within a 0.2 second window. The experimental results of our scheduling framework are compared to the state-of-the-art scheduling frameworks, *TimeGraph* [59], *k-exclusion locking protocol* [144], and *GPU-EvR* [68]. Figure 5.5 shows the number of applications that meet the deadlines. From the results, we observe that, with our scheduling framework, more applications meet their deadlines. The major reason for such an improvement is the fine-grained control of the application behavior through the temporal and the spatial preemptions. Since TimeGraph and k-Exclusive Locking Protocol implement only temporal preemption, these two scheduling framework have a limitation for controlling the GPU resources for multiple application. On the other hand, by implementing the temporal and the spatial preemptions together, our scheduling framework efficiently distributes the GPU resources to the multiple applications. Thus, the number of applications meeting deadlines is maximized with our scheduling framework (see Section 5.3.4 for example behavior of our scheduling framework).

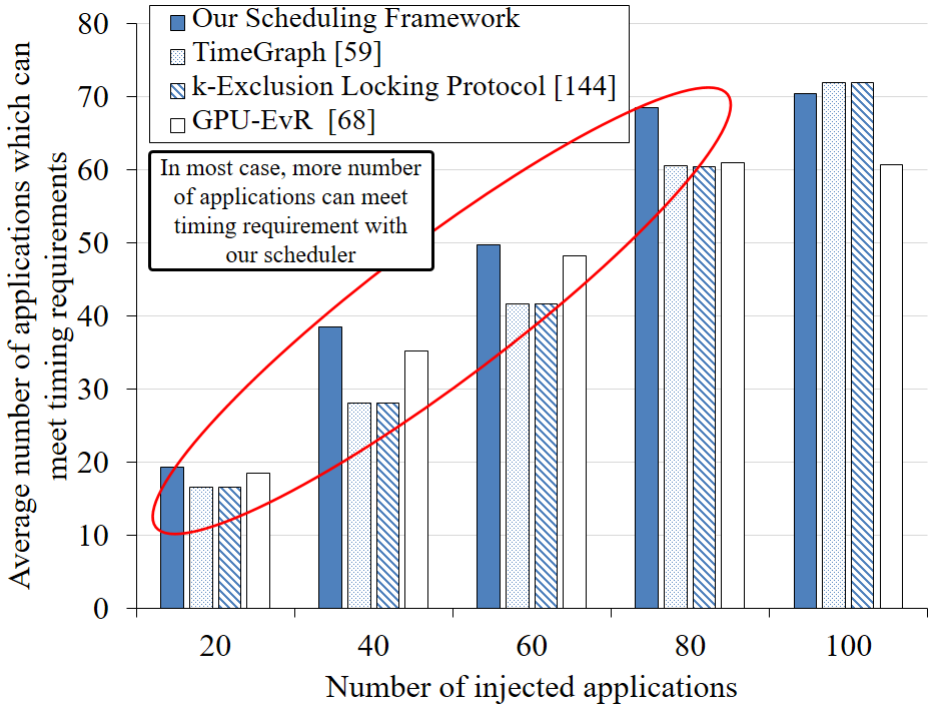


Figure 5.5: Number of Applications Which Meet Deadlines Compared to [59], [144], and [68]
.

However, our scheduling framework and *GPU-EvR* start saturating once 80 applications are injected. When the system is completely saturated with a large number of applications, the partitioning of GPU kernels may cause a negative effect on the applications. Therefore, when the system is saturated, our scheduling framework and *GPU-EvR* may be suffered by the performance bottleneck caused by their scheduling policy. Compared to our scheduling framework, *GPU-EvR* has more severe performance bottleneck due to its scheduling policy. *GPU-EvR* assigns the GPU resources to every running application, but our scheduling framework may not assign the GPU resources to some lower priority applications. Therefore, our scheduling framework may assign more GPU resources to the higher priority applications.

Table 5.2: Average Algorithm Execution Time Compared to [59], [144], and [68].

	Average Algorithm Exec. Time (us)	Implemented Preemptions
Our scheduling framework	9.21	Temporal & Spatial
TimeGraph [59]	2.32	Temporal
k-Exclusion Locking Protocol [144]	1.80	Temporal
GPU-EvR [68]	6.74	Temporal & Spatial

Table 5.2 shows the average algorithm execution time of each of the scheduling frameworks and the preemptions that are implemented by each of the scheduling frameworks. The average algorithm execution time is obtained using an Intel Core i7 Quad-core processor at 3.5 GHz. The results show that our scheduling framework and *GPU-EvR* have more performance overhead compared to other scheduling frameworks due to the implementation of the temporal and the spatial preemptions. In particular, compared to *GPU-EvR*, our scheduling framework has a greater performance overhead which is introduced by the improved scheduling policy. *TimeGraph* and *k-Exclusion Locking Protocol* have less performance overhead, because they do not implement the spatial preemption. However, the performance overhead of our scheduling framework is similar to the kernel launch overhead (see Section 1.2) which is very small compared to the kernel execution time. In addition, the execution of the workload splitter and the GPU execution schedule generator may be overlapped with the execution of other sub-kernels. In addition, the execution of the workload splitter and the GPU execution schedule generator may be overlapped with the execution of other sub-kernels. Therefore, the performance overhead of our preemption algorithms may not cause significant effect.

Figure 5.6 shows the ratio of applications that meet deadlines for each of the priority levels. For each priority level, we observe that more applications meet their deadlines with our scheduling framework (unless the system is completely saturated). From the figure, we can observe that the low priority applications have a larger ratio for meeting their deadlines. Since the system may have low-priority applications that do not have specific deadlines, low-priority applications, which are represented by the *CFD solver* application, may have a highest ratio for meeting deadline.

In addition, compared to *GPU-EvR*, more high priority applications meet their deadlines with our scheduling framework. As mentioned previously, our scheduling framework can assign more GPU resources to the higher priority applications, because our scheduling framework may decide not to assign the GPU resources to some lower priority applications. However,

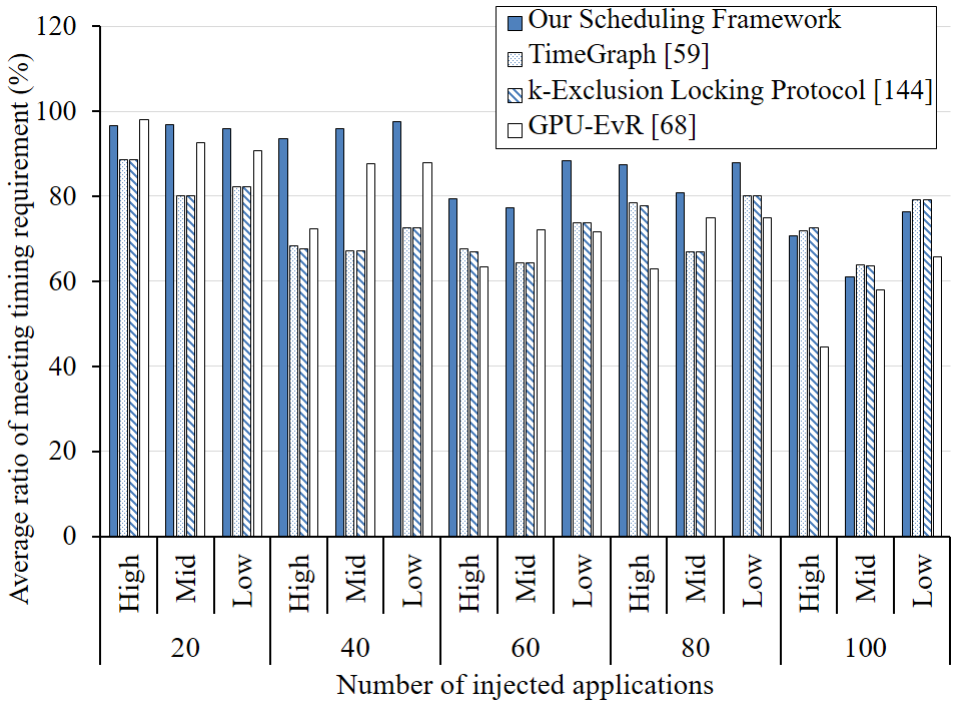


Figure 5.6: Priority Distribution of the Applications Which Meet Deadlines Compared to [59], [144], and [68].

when 20 applications are injected, the results show that slightly more high priority applications meet their deadline with GPU-EvR. Since our scheduler has the largest algorithm execution overhead (see Table 5.2), the implementation of the temporal and the spatial preemptions may cause performance bottleneck. Therefore, when a small number of applications are injected, high priority applications may not have highest ratio for meeting deadlines due to the algorithm execution overhead of our scheduling framework.

5.4.3 Controlling the Effect of Timing Violation

Figure 5.7 shows the total timing violation. In the figure, we observe that the amount of total timing violation is proportional to the number of injected applications. However, total timing violations, in our scheduling framework, is much less than in *TimeGraph* [59], *k-exclusion locking protocol* [144], and *GPU-EvR* [68]. It is observed, from the figure, that our scheduling

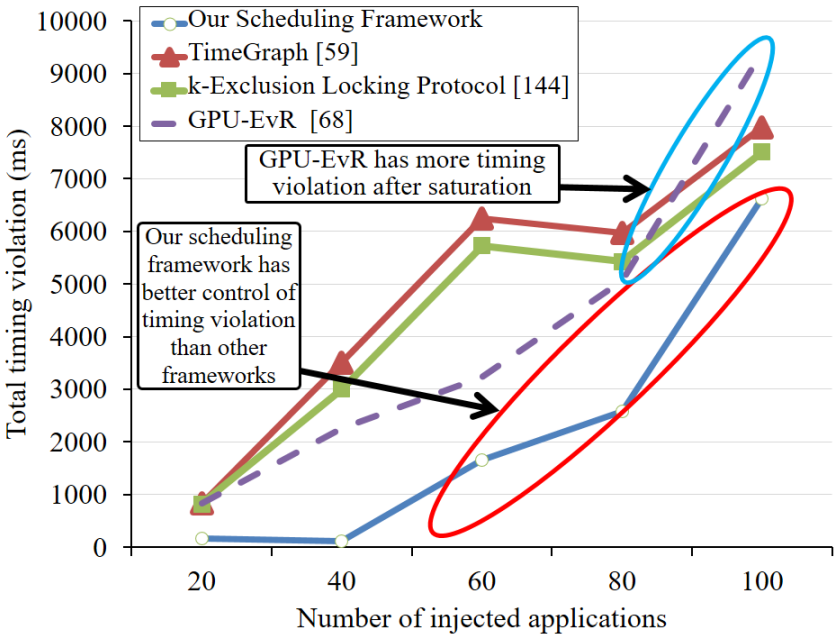


Figure 5.7: Average Total Timing Violation Compared to [59], [144], and [68].

framework has a better control of the timing violation. Since our scheduling framework allows concurrent execution of applications and has fine-grained control of the application behavior, our scheduling framework is able to minimize timing violations. We observe that the amount of total timing violation of our scheduling framework is significantly increased when the injected number of applications is greater than 80. However, our scheduling framework shows less total timing violations compared to [59], [144], and [68]. On the other hand, after the injection of 80 applications, the total timing violation of *GPU-EvR* is significantly increased and is greater than other two scheduling frameworks. Due to the performance bottleneck caused by the scheduling policy, *GPU-EvR* shows a greater amount of total timing violation. By using our scheduling framework, the average timing violation is decreased by 54.57%, 50.45%, and 46.39% compared to [59], [144], and [68], respectively. Therefore, we can conclude that our scheduling framework has, for the most part, a better control of timing violation than the other two state-of-the-art scheduling frameworks.

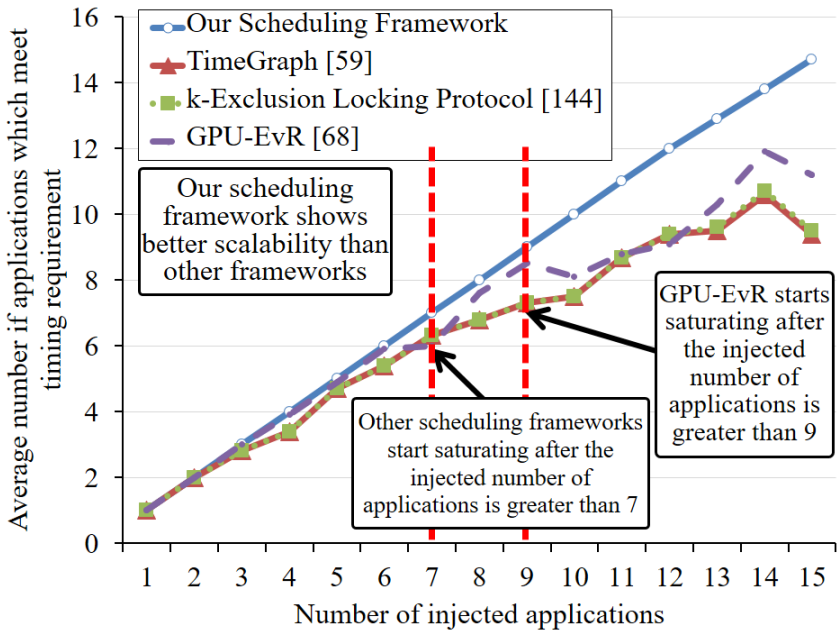


Figure 5.8: Scalability Comparison of Our Scheduling Framework Compared to [59], [144], and [68].

5.4.4 Scalability Analysis of the Scheduling Frameworks

We have also evaluated the scalability of our scheduling framework compared to *TimeGraph* [59], *k-exclusion locking protocol* [144], and *GPU-EvR* [68]. During our experiment, randomly selected applications are injected within a short period of time. The number of injected applications is in the range of 1 to 15. Figure 5.8 shows the average number of applications that meet their deadlines. When the number of the injected applications is increased to 7, all the scheduling frameworks scale in a similar manner. However, after the injected number of applications is greater than 7, *TimeGraph* and *k-exclusion locking protocol* start saturating. In addition, *GPU-EvR* starts saturating after the number of injected application is greater than 9. On the other hand, our scheduling framework scales for larger number of injected applications. The experimental results show that our scheduling framework is able to guarantee up to 1.56 times as many applications as [59], [144], and [68]. Since our scheduling framework handles multiple applications concurrently, and controls application behavior by partitioning the GPU kernels, our scheduling framework shows better scalability than

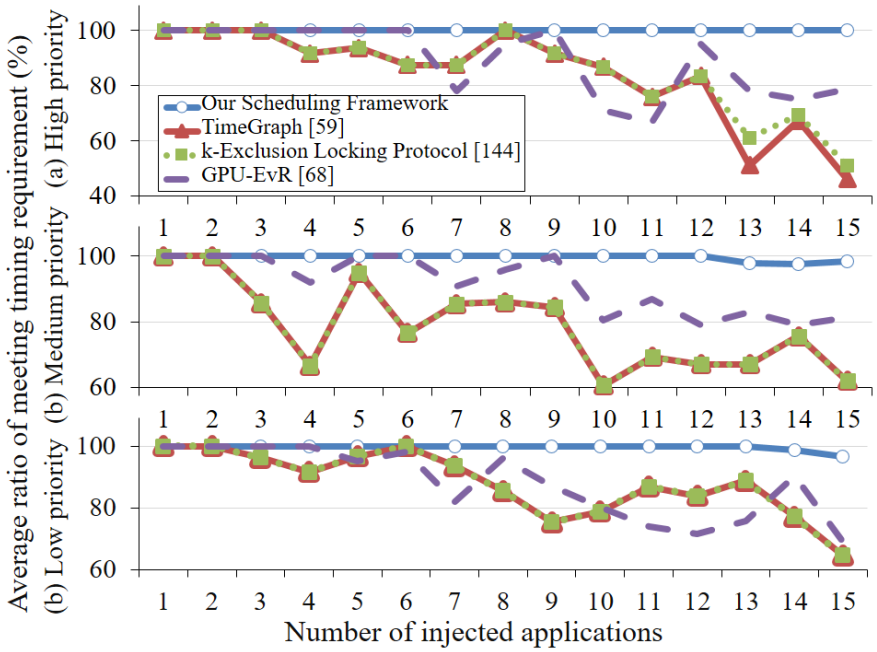


Figure 5.9: Priority Distribution Which Meet Deadlines During Scalability Analysis Compared to [59], [144], and [68].

the other two scheduling frameworks. *GPU-EvR* has a greater saturation limit than other two scheduling frameworks due to its fine-grained GPU resource management. However, since *GPU-EvR* assigns the GPU resources to every running application, *GPU-EvR* has less saturation limit than our scheduling framework.

Figure 5.9 shows the ratio of applications that meet deadlines for each of the priority levels during the scalability analysis. The results show that *TimeGraph* and *k-exclusion locking protocol* scheduling frameworks could not assign the GPU resources to medium priority applications. On the other hand, *GPU-EvR* scheduling framework could assign more GPU resources to medium and high priority applications due to the fine-grained GPU resource management. However, as shown in Figure 5.8, our scheduling framework more efficiently distributes the GPU resources to the applications. The major reason for this improvement is the scheduling policy that implements both temporal and spatial preemption.

5.5 Chapter Summary

In this chapter, we have presented a novel run-time scheduling algorithm for event-driven applications on a GPU-based embedded system. The presented scheduling framework consists of two modules: the workload splitter and the GPU execution schedule generator. The workload splitter partitions the GPU application kernels into multiple sub-kernels based on the current system status and the application requirement. By using our application model, the workload splitter may estimate the amount of required GPU resources in order to meet the application deadlines. The size of the sub-kernels and the number of sub-kernels are decided (spatial preemption) according to the above estimation. After the workload splitter creates sub-kernels, the GPU execution schedule generator generates sub-kernel launch sequences according to the priority and the deadline (temporal preemption) of the applications.

We have evaluated our scheduling framework by comparing with *TimeGraph* [59], and *k-exclusion locking protocol* [144]. Experimental results show that our scheduling framework is able to guarantee up to 1.37 times as many applications as [59] and [144]. In addition, our scheduling framework has a better control of timing violations. The results clearly show that our scheduling framework manages concurrent execution of multiple applications very efficiently. Moreover, the total amount of timing violation is decreased by up to 54.57%, compared to [59] and [144]. The above result implies that our scheduling framework provides better control of timing violation compared to other scheduling frameworks. Moreover, due to the fine-grained application behavior control, our scheduling framework shows better scalability than the other two scheduling frameworks. Our scheduling framework is able to guarantee up to 1.56 times as many applications compared to [59] and [144].

Chapter 6

Conclusion and Future Works

6.1 Conculstion

In recent years, GPUs have been employed in the critical path of applications in embedded systems due to their programmability, high-performance, and low power consumption. The embedded systems have been used in the increasing number of throughput-oriented applications and system events. At the same time, Moore's law has driven the semiconductor industry and the transistor size has been scaled down for decades.

Due to the increasing number of throughput-oriented applications and the nanoscale multi-core processors (including GPUs), the GPU-based embedded system have faced several challenges. Existing workload management techniques do not have enough flexibility to handle multiple applications while interacting with dynamic environment. In addition, due to the small feature size, the state-of-the-art nano-scale multi-core processors have faced several reliability challenges such as aging, soft-error, and process variation. In order to tackle these challenges, this dissertation proposes a reliability and timing aware workload management framework for GPU-based real-time embedded systems. The proposed framework includes

the following content:

Instruction scheduling for improving the soft-error reliability: As shown, the probability of having a soft-error on a single hardware component is proportional to the time that the hardware component is used [143]. In order to maximize the soft-error reliability, we propose an instruction scheduling algorithm. It takes a PTX code and application information as input, and then it generates an instruction schedule and a configuration that maximizes the soft-error reliability of GPU application.

Timing-aware workload scheduling for real-time embedded systems: Due to the increased number of throughput applications, the system should be able to manage its resource efficiently. However, the existing workload management techniques do not have enough flexibility to assign different amounts of resource to multiple applications. In order to tackle this problem, we propose a scheduling framework that partitions the GPU workload into small sub-workloads and generates a flexible schedule for them.

Aging-aware workload management on embedded GPUs: The state-of-the-art nano-scale multi-core processors, including GPUs, have faced several reliability challenges such Negative Bias Temperature Instability (NBTI), Hot Carrier Injection (HCI), and (die-to-die and with-in-die) process variation. The amount of transistor degradation caused by NBTI and HCI is proportional to the time a transistor is stressed or switched. Moreover, the process variation increases the randomness in the system status. The existing state-of-the-art GPU workload management techniques focus on maximizing the performance [69], thus the stress or switching activity is not evenly distributed across the GPU. Since the process variation is not considered, the existing techniques may not minimize the aging effect on embedded GPUs under process variation. In order to solve this problem, we proposed a workload management technique that evenly distributes the stress. The proposed technique takes the number of instructions and the critical path delay information to balance workload distribution on GPUs while considering the randomness of process variation.

In summary, our reliability and timing aware workload management framework is able to minimize the amount of aging effect on embedded GPUs and flexibly generate GPU workload schedules to handle the increasing number of throughput oriented applications.

6.2 Future Works

In addition to the research work we have presented in this dissertation, we would like to explore the following directions in the future. In this dissertation, we focus on minimizing aging effect and improving soft-error reliability on embedded GPUs. At the same time, our GPU workload scheduling framework generates schedule to maximize the number of applications that satisfy their deadlines. However, since performance and power are considered as overheads, our workload management framework may not be functioning properly if the performance and power are given as constraints. It is common to design the embedded systems with strict performance and power budget. Therefore, we will take performance and power into consideration.

We consider the process variation and core-level guradbanding to further improve the lifetime of the GPU. However, it is possible that the process variation causes asynchronous behavior and the arithmetic pipeline is not properly utilized. Figure 1.5(b) is such an example. The asynchronous behavior in the presence of the process variation need to be considered to further improve the lifetime of embedded GPUs.

Bibliography

[1] A. Agrawal, A. Ansari, and J. Torrellas. "Mosaic: Exploiting the spatial locality of process variation to reduce refresh energy in on-chip eDRAM modules". *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*, pages 84–95, 2014.

[2] P. Aguilera, J. Lee, A. Farmahini-Farahani, K. Morrow, M. Schulte, and N. S. Kim. "Process variation-aware workload partitioning algorithms for GPUs supporting spatial-multitasking". *Design, Automation Test in Europe Conference Exhibition (DATE'14)*, pages 1–6, March 2014.

[3] D. Ancajas, K. Chakraborty, and S. Roy. "Proactive Aging Management in Heterogeneous NoCs Through a Criticality-driven Routing Approach". *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'13)*, pages 32–37, 2013.

[4] A. Ansari, A. Mishra, J. Xu, and J. Torrellas. "Tangle: Route-oriented dynamic voltage minimization for variation-afflicted, energy-efficient on-chip networks". *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA'14)*, pages 440–451, 2014.

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. "The Landscape of Parallel Computing Research: A View from Berkeley". *EECS Department, University of California, Berkeley*, 2006.

[6] S. Baeg, S. Wen, and R. Wong. "SRAM Interleaving Distance Selection With a Soft Error Failure Model". *IEEE Transactions on Nuclear Science*, 56(4):2111–2118, 2009.

[7] S. Baghsorkhi, M. Delahaye, S. Patel, W. Gropp, and W. Hwu. "An Adaptive Performance Modeling Tool for GPU Architectures". *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'10)*, pages 105–114, 2010.

[8] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. "Analyzing CUDA workloads using a detailed GPU simulator". *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*, pages 163–174, 2009.

[9] M. Bandan, S. Bhattacharjee, R. Shafik, D. Pradhan, and J. Mathew. "Lifetime Reliability-Aware Checkpointing Mechanism: Modelling and Analysis". *2013 International Symposium on Electronic System Design (ISED'13)*, pages 128–132, 2013.

[10] C. Basaran and K.-D. Kang. "Supporting Preemptive Task Executions and Memory Copies in GPGPUs". *Euromicro Conference on Real-Time Systems (ECRTS'12)*, 2012.

[11] A. Benini, M. J. Rutherford, and K. P. Valavanis. "Real-time, GPU-based Pose Estimation of a UAV for Autonomous Takeoff and Landing". *IEEE International Conference on Robotics and Automation (ICRA'16)*, pages 63–70, May 2016.

[12] K. Berezovskyi, L. Santinelli, K. Bletsas, and E. Tovar. "WCET Measurement-based and Extreme Value Theory Characterisation of CUDA Kernels". *Proceedings of the 22Nd International Conference on Real-Time Networks and Systems (RTNS'14)*, pages 279–288, 2014.

[13] D. Berjon, C. Cuevas, F. Moran, and N. Garcia. "GPU-based implementation of an optimized nonparametric background modeling for real-time moving object detection". *IEEE Transactions on Consumer Electronics*, pages 361–369, May 2013.

[14] E. Cai and D. Marculescu. "Temperature Effect Inversion-Aware Power-Performance Optimization for FinFET-Based Multi-Core Systems". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2017.

[15] A. Calimera, E. Macii, and M. Poncino. "NBTI-aware Power Gating for Concurrent Leakage and Aging Optimization". *ACM/IEEE International Symposium on Low Power Electronics and Design (ISLPED'09)*, pages 127–132, 2009.

[16] C. Cao, T. Herault, G. Bosilca, and J. Dongarra. "Design for a Soft Error Resilient Dynamic Task-Based Runtime". *IEEE International Parallel and Distributed Processing Symposium*, pages 765–774, May 2015.

[17] A. Chakraborty, G. Ganesan, A. Rajaram, and D. Pan. "Analysis and Optimization of NBTI Induced Clock Skew in Gated Clock Trees". *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'09)*, pages 296–299, 2009.

[18] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron. "Rodinia: A benchmark suite for heterogeneous computing". *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC'09)*, pages 44–54, 2009.

[19] X. Chen, Y. Wang, Y. Liang, Y. Xie, and H. Yang. "Run-time technique for simultaneous aging and power optimization in GPGPUs". *Proceedings of the 51th Annual Design Automation Conference (DAC'14)*, pages 1–6, 2014.

[20] Y. Chen, A. Calimera, E. Macii, and M. Poncino. "Characterizing the Activity Factor in NBTI Aging Models for Embedded Cores". *Proceedings of the 25th Edition on Great Lakes Symposium on VLSI (GLSVLSI'15)*, pages 75–78, 2015.

[21] J. Cong and C. Yu. "Impact of loop transformations on software reliability". *IEEE/ACM International Conference on Computer-Aided Design (ICCAD'15)*, pages 278–285, 2015.

[22] C. Cuevas, D. Berjon, F. Moran, and N. Garcia. "Moving object detection for real-time augmented reality applications in a GPGPU". *IEEE Transactions on Consumer Electronics*, 58(1):117–125, 2012.

[23] A. Das, G. V. Merrett, and B. M. Al-Hashimi. "The slowdown or race-to-idle question: Workload-aware energy optimization of SMT multicore platforms under process variation". *Design, Automation Test in Europe Conference Exhibition (DATE'16)*, pages 535–538, 2016.

[24] D. A. G. de Oliveira, L. L. Pilla, T. Santini, and P. Rech. "Evaluation and Mitigation of Radiation-Induced Soft Errors in Graphics Processing Units". *IEEE Transactions on Computers*, 65(3):791–804, March 2016.

[25] D. Defour and E. Petit. "GPUburn: A system to test and mitigate GPU hardware failures". *2013 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII'13)*, pages 263–270, 2013.

[26] G. A. Elliott and J. H. Anderson. "Globally scheduled real-time multiprocessor systems with GPUs". *International Conference on Real-Time and Network Systems (RTNS'10)*, 2010.

[27] G. A. Elliott and J. H. Anderson. "An optimal k-exclusion real-time locking protocol motivated by multi-GPU systems". *International Conference on Real-Time Networks and Systems (RTNS'11)*, pages 15–24, 2011.

[28] G. A. Elliott and J. H. Anderson. "Real-World Constraints of GPUs in Real-Time Systems". *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'11)*, 2011.

[29] G. A. Elliott and J. H. Anderson. "Exploring the Multitude of Real-Time Multi-GPU Configurations". *IEEE Real-Time Systems Symposium*, pages 260–271, Dec 2014.

[30] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi. "GPU-Qin: A methodology for evaluating the error resilience of GPGPU applications". *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'14)*, pages 221–230, 2014.

[31] B. Fang, J. Wei, K. Pattabirama, and M. Ripeanu. "Evaluating Error Resiliency of GPGPU Applications". *High Performance Computing, Networking, Storage and Analysis (SCC'13)*, pages 1502–1503, 2012.

[32] B. Fang, J. Wei, K. Pattabiraman, and M. Ripeanu. "Towards Building Error Resilient GPGPU Applications". *3rd Workshop on Resilient Architecture (WRA'12)*, 2012.

[33] D. Flores-Tapia and S. Pistorius. "A real time Breast Microwave Radar imaging reconstruction technique using simt based interpolation". *IEEE International Conference on Image Processing (ICIP'10)*, pages 1389–1392, 2010.

[34] T. Furukawa, B. Lavis, and H. F. Durrant-Whyte. "Parallel grid-based recursive Bayesian estimation using GPU for real-time autonomous navigation". *IEEE International Conference on Robotics and Automation*, pages 316–321, May 2010.

[35] J. A. G.A. Elliott, B.C. Ward. "GPUSync: A Framework for Real-Time GPU Management". *2013 IEEE 34th Real-Time Systems Symposium (RTSS'13)*, pages 33–44, 2012.

[36] V. Ganapathi, C. Plagemann, D. Koller, and S. Thrun. "Real time motion capture using a single time-of-flight camera". *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'10)*, pages 755–762, 2010.

[37] G. Georgakos, U. Schlichtmann, R. Schneider, and S. Chakraborty. "Reliability Challenges for Electric Vehicles: From Devices to Architecture and Systems Software". *50th Annual Design Automation Conference (DAC'13)*, pages 1–9, 2013.

[38] D. Gnad, M. Shafique, F. Kriebel, S. Rehman, D. Sun, and J. Henkel. Hayat: Harnessing dark silicon and variability for aging deceleration and balancing. *Proceedings of the 52nd Annual Design Automation Conference (DAC'15)*, pages 1–6, 2015.

[39] L. B. Gomez, F. Cappello, L. Carro, N. Debardeleben, B. Fang, S. Gurumurthi, K. Pattabiraman, P. Rech, and M. S. Reorda. "GPGPUs: How to Combine High Computational Power with High Reliability". *Design, Automation and Test in Europe Conference and Exhibition (DATE'14)*, pages 1–9, 2014.

[40] I. Haque and V. Pande. "Hard Data on Soft Errors: A Large-Scale Assessment of Real-World Error Rates in GPGPU". *10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid'10)*, pages 691–696, 2010.

[41] M. Harris. How to implement performance metrics in cuda c/c+. *https://devblogs.nvidia.com/parallelforall/how-implement-performance-metrics-cuda-cc*, 2012.

[42] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn. "Reliable on-chip systems in the nano-era: Lessons learnt and future trends". *50th ACM/EDAC/IEEE Design Automation Conference (DAC'13)*, pages 1–10, May 2013.

[43] D. Honegger, H. Oleynikova, and M. Pollefeys. "Real-time and low latency embedded computer vision hardware based on a combination of FPGA and mobile CPU". *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 30–35, Sept 2014.

[44] S. Hong and H. Kim. "An Integrated GPU Power and Performance Model". *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA'10)*, pages 280–289, 2010.

[45] H. Huang and C.-P. Wen. "Layout-Based Soft Error Rate Estimation Framework considering Multiple Transient Faults - from Device to Circuit Level". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD'15)*, pages 1–1, 2015.

[46] L. Huang and Q. Xu. "Performance Yield-driven Task Allocation and Scheduling for MPSoCs Under Process Variation". *Design Automation Conference (DAC'10)*, pages 326–331, 2010.

[47] L. Huang, F. Yuan, and Q. Xu. "Lifetime reliability-aware task allocation and scheduling for MPSoC platforms". *Design, Automation Test in Europe Conference Exhibition (DATE'09)*, pages 51–56, 2009.

[48] W. Huang, K. Skadron, S. Gurumurthi, R. J. Ribando, and M. R. Stan. "Differentiating the roles of IR measurement and simulation for power and temperature-aware design". *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'09)*, pages 1–10, April 2009.

[49] S. i. Abe, R. Ogata, and Y. Watanabe. "Impact of Nuclear Reaction Models on Neutron-Induced Soft Error Rate Analysis". *IEEE Transactions on Nuclear Science*, pages 1806–1812, 2014.

[50] Y. Iwase, D. Abe, and T. Yakoh. "GPGPU aided method for real-time systems". *IEEE International Conference on Industrial Informatics (INDIN'12)*, pages 841–845, 2012.

[51] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison, and A. Fitzgibbon. "KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera". *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST'11)*, pages 559–568, 2011.

[52] J. Jablin, T. Jablin, O. Mutlu, and M. Herlihy. "Warp-aware Trace Scheduling for GPUs". *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*, pages 163–174, 2014.

[53] O. H. Jafari, D. Mitzel, and B. Leibe. "Real-time RGB-D based people detection and tracking for mobile robots and head-worn cameras". *IEEE International Conference on Robotics and Automation (ICRA'14)*, pages 36–43, May 2014.

[54] H. Jeon and M. Annavaram. "Warped-DMR: Light-weight Error Detection for GPGPU". *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*, pages 37–47, 2012.

[55] H. Jeon, M. Wilkening, V. Sridharan, S. Gurumurthi, and G. Loh. "Architectural Vulnerability Modeling and Analysis of Integrated Graphics Processors". *Workshop on Silicon Errors in Logic-System Effects (SELSE'13)*, pages 1–6, 2013.

[56] M. Joselli, M. Zamith, E. Clua, A. Montenegro, A. Conci, R. Leal-Toledo, L. Valente, B. Feijo, M. d'Ornellas, and C. Pozzer. "Automatic Dynamic Task Distribution between CPU and GPU for Real-Time Systems". *IEEE International Conference on Computational Science and Engineering (CSE'08)*, pages 48–55, 2008.

[57] N. Kapadia and S. Pasricha. "VARSHA: Variation and reliability-aware application scheduling with adaptive parallelism in the dark-silicon era". *Design, Automation Test in Europe Conference Exhibition (DATE'15)*, pages 1060–1065, 2015.

[58] S. Kato, K. Lakshmanan, A. Kumar, M. Kelkar, Y. Ishikawa, and R. Rajkumar. "RGEM: A Responsive GPGPU Execution Model for Runtime Engines". *Real-Time Systems Symposium (RTSS'11)*, pages 57–66, 2011.

[59] S. Kato, K. Lakshmanan, R. R. Rajkumar, and Y. Ishikawa. "TimeGraph: GPU scheduling for real-time multi-tasking environments". *USENIX Annual Technical Conference (USENIX ATC'11)*, 2011.

[60] J. Keane, T. H. Kim, and C. H. Kim. "An On-Chip NBTI Sensor for Measuring pMOS Threshold Voltage Degradation". *IEEE Transactions on Very Large Scale Integration Systems (TVLSI'10)*, 18(6):947–956, 2010.

[61] S. Kiamehr, F. Firouzi, and M. Tahoori. "Input and Transistor Reordering for NBTI and HCI Reduction in Complex CMOS Gates". *Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI'12)*, pages 201–206, 2012.

[62] S. Kiamehr, T. Osiecki, M. Tahoori, and S. Nassif. "Radiation-Induced Soft Error Analysis of SRAMs in SOI FinFET Technology: A Device to Circuit Approach". *Proceedings of the 51st Annual Design Automation Conference (DAC'14)*, 2014.

[63] H. Kim, A. Vitkovskiy, P. Gratz, and V. Soteriou. "Use it or lose it: wear-out and lifetime in future chip multiprocessors". *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*, pages 136–147, 2013.

[64] V. Kozhikkottu, A. Pan, V. Pai, S. Dey, and A. Raghunathan. Variation aware cache partitioning for multithreaded programs. *51st ACM/IEEE Design Automation Conference (DAC'14)*, pages 1–6, 2014.

[65] F. Kriebel, S. Rehman, M. Shafique, and J. Henkel. ageopt-rmt: Compiler-driven variation-aware aging optimization for redundant multithreading. *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC'16)*, pages 1–6, June 2016.

[66] L. Lai, V. Chandra, R. C. Aitken, and P. Gupta. "SlackProbe: A Flexible and Efficient In Situ Timing Slack Monitoring Methodology". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 33(8):1168–1179, Aug 2014.

[67] M. Lancaster and R. Green. "Enhanced real time facial detection and replacement using GPGPU". *International Conference of Image and Vision Computing New Zealand (IVCNZ'13)*, pages 276–281, 2013.

[68] H. Lee and M. A. A. Faruque. "GPU-EvR: Run-Time Event Based Real-Time Scheduling Framework on GPGPU Platform". *Design, Automation and Test in Europe Conference and Exhibition (DATE'14)*, pages 1–6, 2014.

[69] H. Lee, M. Shafique, and M. A. A. Faruque. "Low-overhead Aging-aware Resource Management on Embedded GPUs". *54th ACM/EDAC/IEEE Design Automation Conference (DAC'17)*, pages 1–6, 2017.

[70] J. Lee, P. P. Ajgaonkar, and N. S. Kim. "Analyzing throughput of GPGPUs exploiting within-die core-to-core frequency variation". *IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS'11)*, pages 237–246, 2011.

[71] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. Kim, T. Aamodt, and V. Reddi. "GPUWattch: Enabling Energy Optimizations in GPGPUs". *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA'13)*, pages 487–498, 2013.

[72] D. Li, J. Vetter, and W. Yu. "Classifying Soft Error Vulnerabilities in Extreme-scale Scientific Applications Using a Binary Instrumentation Tool". *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'12*, pages 1–11, 2012.

[73] S. Li, N. Farooqui, and S.Yalamanchili. "Software Reliability Enhancements for GPU Applications". *Sixth Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG'13)*, 2013.

[74] Q. Liu, C. Jung, D. Lee, and D. Tiwarit. "Low-cost soft error resilience with unified data verification and fine-grained recovery for acoustic sensor based detection". *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'16)*, pages 1–12, Oct 2016.

[75] A. Lotfi, A. Rahimi, L. Benini, and R. Gupta. "Aging-Aware Compilation for GPGPUs". *ACM Transactions on Architecture and Code Optimization (TACO'15)*, pages 1–20, 2015.

[76] A. Maghazeh, U. D. Bordoloi, A. Horga, P. Eles, and Z. Peng. "Saving Energy Eithout Defying Deadlines on Mobile GPU-Based Heterogeneous Systems". *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'14)*, pages 1–10, 2014.

[77] N. Maruyama, A. Nukada, and S. Matsuoka. "A High-Performance Fault-Tolerant Software Framework for Memory on Commodity GPUs". *IEEE International Symposium on Parallel & Distributed Processing (IPDPS'10)*, pages 1–12, 2010.

[78] D. Maturana and S. Scherer. Voxnet: A 3d convolutional neural network for real-time object recognition. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'15)*, pages 922–928, Sept 2015.

[79] M. McNaughton, C. Urmson, J. M. Dolan, and J. W. Lee. "Motion planning for autonomous driving with a conformal spatiotemporal lattice". *IEEE International Conference on Robotics and Automation*, pages 89–95, May 2011.

[80] R. Membarth, J. Lupp, F. Hannig, J. Teich, M. Korner, and W. Eckert. "Dynamic task-scheduling and resource management for GPU accelerators in medical imaging". *International Conference on Architecture of Computing Systems (ARCS'12)*, pages 147–159, 2012.

[81] P. Mercati, F. Paterna, A. Bartolini, L. Benini, and T. S. Rosing. "Dynamic variability management in mobile multicore processors under lifetime constraints". *IEEE 32nd International Conference on Computer Design (ICCD'14)*, pages 448–455, 2014.

[82] D. Mirzoyan, B. Akesson, and K. Goossens. "Process-variation-aware Mapping of Best-effort and Real-time Streaming Applications to MPSoCs". *ACM Transactions on Embedded Computing Systems*, 13(2s):1–24, Jan. 2014.

[83] N. Miskov-Zivanov and D. Marculescu. "MARS-C: Modeling and Reduction of Soft Errors in Combinational Circuits". *Proceedings of the 43rd Annual Design Automation Conference (DAC'06)*, pages 767–772, 2006.

[84] S. Mittal. "A Survey of Techniques for Architecting and Managing GPU Register File". *IEEE Transactions on Parallel and Distributed Systems*, 28(1):16–28, Jan 2017.

[85] S. Mu, C. Wang, M. Liu, D. Li, M. Zhu, X. Chen, X. Xie, and Y. Deng. "Evaluating the potential of graphics processors for high performance embedded computing". *Design, Automation Test in Europe Conference Exhibition (DATE'11)*, pages 1–6, 2011.

[86] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor". *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03)*, pages 29–40, 2003.

[87] L. Mussi, S. Cagnoni, and F. Daolio. "GPU-Based Road Sign Detection Using Particle Swarm Optimization". *International Conference on Intelligent Systems Design and Applications (ISDA'09)*, pages 152–157, 2009.

[88] M. Namaki-Shoushtari, A. Rahimi, N. Dutt, P. Gupta, and R. K. Gupta. "ARGO: Aging-aware GPGPU Register File Allocation". *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)*, pages 1–9, 2013.

[89] M. Nasro, U. Samee, W. Xiuli, and Y. Albert. "lowest priority first based feasibility analysis of real-time systems". *Journal of Parallel and Distributed Computing*, pages 1066–1075, 2013.

[90] B. Nie, D. Tiwari, S. Gupta, E. Smirni, and J. H. Rogers. A large-scale study of ssoft-errors on gpus in the field. *IEEE International Symposium on High Performance Computer Architecture (HPCA'16)*, pages 519–530, March 2016.

[91] A. Nukada, H. Takizawa, and S. Matsuoka. "NVCR: A Transparent Checkpoint-Restart Library for NVIDIA CUDA". *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW'11)*, pages 104–113, 2011.

[92] NVIDIA. "CUDA C Programming Guide", 2012.

[93] NVIDIA. "NVIDIA's next generation CUDA compute architecture: Kepler GK110". 2012.

[94] NVIDIA. "NVIDIA Jetson TK1 Development Kit Bringing GPU-accelerated computing to Embedded Systems". 2014.

[95] NVIDIA. "Pace Setter: Audi to Use Tegra X1, Accelerating Drive to Self-Piloted Cars". `http://blogs.nvidia.com/blog/2015/01/06/audi-tegra-x1/`, 2015.

[96] F. Oboril and M. B. Tahoori. "ExtraTime: Modeling and analysis of wearout due to transistor aging at microarchitecture-level". *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'12)*, pages 1–12, 2012.

[97] D. Oliveira, C. Lunardi, L. Pilla, P. Rech, P. Navaux, and L. Carro. "Radiation Sensitivity of High Performance Computing Applications on Kepler-Based GPGPUs". *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*, pages 732–737, 2014.

[98] D. A. G. Oliveira, P. Rech, H. M. Quinn, T. D. Fairbanks, L. Monroe, S. E. Michalak, C. Anderson-Cook, P. O. A. Navaux, and L. Carro. "Modern GPUs Radiation Sensitivity Evaluation and Mitigation Through Duplication With Comparison". *IEEE Transactions on Nuclear Science*, 61:3115–3122, Dec 2014.

[99] R. Otsuka, I. Sato, and R. Nakamura. "GPU based real-time surgical navigation system with three-dimensional ultrasound imaging for water-filled laparo-endoscope surgery". *Engineering in Medicine and Biology Society (EMBC'12), Annual International Conference of the IEEE*, pages 2800–2803, 2012.

[100] R. B. Parizi, R. Ferreira, L. Carro, and Á. Moreira. "Compiler Optimizations Do Impact the Reliability of Control-Flow Radiation Hardened Embedded Software". *Embedded Systems: Design, Analysis and Verification: 4th IFIP TC 10 International Embedded Systems Symposium (IESS'13)*, pages 49–60, 2013.

[101] J. Park, Q. Wang, D. Jayasinghe, J. Li, Y. Kanemasa, M. Matsubara, D. Yokoyama, M. Kitsuregawa, and C. Pu. "Variations in Performance Measurements of Multi-core Processors: A Study of n-Tier Applications". *2013 IEEE International Conference on Services Computing*, pages 336–343, June 2013.

[102] F. Paterna, A. Acquaviva, and L. Benini. "Aging-Aware Energy-Efficient Workload Allocation for Mobile Multimedia Platforms". *IEEE Transactions on Parallel and Distributed Systems*, 24(8):1489–1499, 2013.

[103] F. Paterna, A. Acquaviva, and L. Benini. "Aging-Aware Energy-Efficient Workload Allocation for Mobile Multimedia Platforms". *IEEE Transactions on Parallel and Distributed Systems*, pages 89–99, 2013.

[104] F. Paterna, A. Acquaviva, A. Caprara, F. Papariello, G. Desoli, and L. Benini. "Variability-Aware Task Allocation for Energy-Efficient Quality of Service Provisioning in Embedded Streaming Multimedia Applications". *IEEE Transactions on Computers*, 61(7):939–953, 2012.

[105] A. Pathania, Q. Jiao, A. Prakash, and T. Mitra. "Integrated CPU-GPU Power Management for 3D Mobile Games". *51st ACM/IEEE Design Automation Conference (DAC'14)*, pages 1–6, 2014.

[106] M. Pedersoli, J. Gonzlez, X. Hu, and X. Roca. "Toward Real-Time Pedestrian Detection Based on a Deformable Template Model". *IEEE Transactions on Intelligent Transportation Systems*, 15(1):355–364, Feb 2014.

[107] B. Raghunathan, Y. Turakhia, S. Garg, and D. Marculescu. "Cherry-picking: Exploiting process variations in dark-silicon homogeneous chip multi-processors". *Design, Automation Test in Europe Conference Exhibition (DATE'13)*, pages 39–44, 2013.

[108] A. Rahimi, L. Benini, and R. Gupta. "Aging-aware Compiler-directed VLIW Assignment for GPGPU Architectures". *Proceedings of the 50th Annual Design Automation Conference (DAC'13)*, pages 1–6, 2013.

[109] A. Rahimi, L. Benini, and R. K. Gupta. "Analysis of instruction-level vulnerability to dynamic voltage and temperature variations". *Design, Automation Test in Europe Conference Exhibition (DATE'12)*, pages 1102–1105, 2012.

[110] A. Rahimi, A. Marongiu, R. K. Gupta, and L. Benini. "A variability-aware OpenMP environment for efficient execution of accuracy-configurable computation on shared-FPU processor clusters". *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'13)*, pages 1–10, Sept 2013.

[111] P. Rech, L. Pilla, P. Navaux, and L. Carro. "Impact of GPUs Parallelism Management on Safety-Critical and HPC Applications Reliability". *44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'14)*, pages 455–466, 2014.

[112] J. Redmon and A. Angelova. "Real-time grasp detection using convolutional neural networks". *IEEE International Conference on Robotics and Automation (ICRA'15)*, pages 16–22, May 2015.

[113] S. Rehman, F. Kriebel, D. Sun, M. Shafique, and J. Henkel. "dTune: Leveraging Reliable Code Generation for Adaptive Dependability Tuning Under Process Variation and Aging-Induced Effects". *Proceedings of the 51st Annual Design Automation Conference (DAC'14)*, pages 1–6, 2014.

[114] S. Rehman, M. Shafique, and J. Henkel. "Instruction scheduling for reliability-aware compilation". *2012 49th ACM/EDAC/IEEE Design Automation Conference (DAC'12),*, pages 1288–1296, 2012.

[115] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel. "Reliable Software for Unreliable Hardware: Embedded Code Generation Aiming at Reliability". *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis (CODES+ISSS'11)*, pages 237–246, 2011.

[116] S. Rehman, M. Shafique, F. Kriebel, and J. Henkel. "RAISE: Reliability-Aware Instruction SchEduling for unreliable hardware". *17th Asia and South Pacific Design Automation Conference (ASP-DAC'12)*, pages 671–676, 2012.

[117] D. Sabena, L. Sterpone, L. Carro, and P. Rech. "Reliability Evaluation of Embedded GPGPUs for Safety Critical Applications". *IEEE Transactions on Nuclear Science*, pages 23–29, 2014.

[118] G. Sadowski. "Design Challenges Facing CPU-GPU-Accelerator Integrated Heterogeneous Systems". *Design Automation Conference (DAC'14)*, 2014.

[119] M. Salehi, M. K. Tavana, S. Rehman, F. Kriebel, M. Shafique, A. Ejlali, and J. Henkel. "DRVS: Power-efficient reliability management through Dynamic Redundancy and Voltage Scaling under variations". *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED'15)*, pages 225–230, July 2015.

[120] M. Shafique, D. Gnad, S. Garg, and J. Henkel. "Variability-aware dark silicon management in on-chip many-core systems". *Design, Automation Test in Europe Conference Exhibition (DATE'15)*, pages 387–392, 2015.

[121] J. W. Sheaffer, D. P. Luebke, and K. Skadron. "A Hardware Redundancy and Recovery Mechanism for Reliable Scientific Computation on Graphics Processors". *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, pages 55–64, 2007.

[122] G. Shi, J. Enos, M. Showerman, and V. Kindratenko. "On Testing GPU Memory for Hard and Soft Errors". *Proc. Symposium on Application Accelerators in High-Performance Computing (SAAHPC'09)*, pages 1–3, 2009.

[123] G. Shobaki, K. Wilken, and M. Heffernan. "Optimal Trace Scheduling Using Enumeration". *ACM Transactions on Architecture and Code Optimization (TACO'09)*, pages 1–32, 2009.

[124] P. Singh, E. Karl, D. Blaauw, and D. Sylvester. "Compact Degradation Sensors for Monitoring NBTI and Oxide Degradation". *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, pages 1645–1655, 2012.

[125] J. Sun, R. Lysecky, K. Shankar, A. Kodi, A. Louri, and J. Roveda. "Workload Assignment Considering NBTI Degradation in Multicore Systems". *ACM Journal on Emerging Technologies in Computing Systems (JETC'14)*, pages 1–22, 2014.

[126] J. Sun, R. Zheng, J. .Velamala, Y. Cao, R. Lysecky, K. Shankar, and J. Roveda. "A Self-tuning Design Methodology for Power-efficient Multi-core Systems". *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, pages 1–24, 2013.

[127] J. Tan, M. Chen, Y. Yi, and X. Fu. "Mitigating the Impact of Hardware Variability for GPGPUs Register File". *IEEE Transactions on Parallel and Distributed Systems*, 27(11):3283–3297, 2016.

[128] J. Tan and X. Fu. "RISE: Improving the Streaming Processors Reliability Against Soft Errors in GPGPUs". *Proceedings of the 21st international conference on Parallel architectures and compilation techniques (PACT'12)*, pages 191–200, 2012.

[129] J. Tan, Z. Li, M. Chen, and X. Fu. "Exploring Soft-Error Robust and Energy-Efficient Register File in GPGPUs Using Resistive Memory". *ACM Trans. Des. Autom. Electron. Syst.*, pages 1–25, jan 2016.

[130] J. Tan, Z. Li, and X. Fu. "Soft-error reliability and power co-optimization for GPGPUs register file using resistive memory". *Design, Automation Test in Europe Conference Exhibition (DATE'15)*, pages 369–374, 2015.

[131] H. Tashima, E. Yoshida, S. Kinouchi, F. Nishikido, N. Inadama, H. Murayama, M. Suga, H. Haneishi, and T. Yamaya. "Real-Time Imaging System for the Open-PET". *IEEE Transactions on Nuclear Science*, 59(1):40–46, 2012.

[132] M. K. Tavana, A. Kulkarni, A. Rahimi, T. Mohsenin, and H. Homayoun. "Energy-efficient mapping of biomedical applications on domain-specific accelerator under process variation". *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED'14)*, pages 275–278, Aug 2014.

[133] D. Tiwari, S. Gupta, J. Rogers, D. Maxwell, P. Rech, S. Vazhkudai, D. Oliveira, D. Londo, N. DeBardeleben, P. Navaux, L. Carro, and A. Bland. "Understanding GPU errors on large-scale HPC systems and the implications for system design and operation". *IEEE 21st International Symposium on High Performance Computer Architecture (HPCA'15)*, pages 331–342, 2015.

[134] U. Verner, A. Schuster, M. Silberstein, and A. Mendelson. "Scheduling Processing of Real-time Data Streams on Heterogeneous multi-GPU Systems". *Proceedings of the 5th Annual International Systems and Storage Conference (SYSTOR'12)*, pages 1–12, 2012.

[135] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron. "Real-world Design and Evaluation of Compiler-managed GPU Redundant Multithreading". *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA'14)*, pages 73–84, 2014.

[136] J. Wang, S. Zhong, L. Yan, and Z. Cao. "An Embedded System-on-Chip Architecture for Real-time Visual Detection and Matching". *IEEE Transactions on Circuits and Systems for Video Technology*, 24(3):525–538, March 2014.

[137] S. Wang, T. Jin, C. Zheng, and G. Duan. "Low Power Aging-Aware On-Chip Memory Structure Design by Duty Cycle Balancing". *Journal of Circuits, Systems and Computers*, 25(9):1–24, 2016.

[138] B. C. Ward, G. A. Elliott, and J. H. Anderson. "Replica-Request Priority Donation: A Real-Time Progress Mechanism for Global Locking Protocols". *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA'12)*, pages 280–289, 2012.

[139] R. Xiaoguang, X. Xinhai, W. Qian, C. Juan, W. Miao, and Y. Xuejun. "GS-DMR: Low-overhead soft error detection scheme for stencil-based computation". *Parallel Computing*, pages 50–65, 2015.

[140] J. Xiong, V. Zolotov, and L. He. "Robust Extraction of Spatial Correlation". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 26(4):619–631, 2007.

[141] Q. Xu and M. Annavaram. "PATS: Pattern Aware Scheduling and Power Gating for GPGPUs". *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT'14)*, pages 225–236, 2014.

[142] Y. Xu, R. Wang, T. Li, M. Song, L. Gao, Z. Luan, and D. Qian. "Scheduling Tasks with Mixed Timing Constraints in GPU-Powered Real-Time Systems". *Proceedings of the 2016 International Conference on Supercomputing (ICS'16)*, pages 1–13, 2016.

[143] J. Yan and W. Zhang. "Compiler-guided Register Reliability Improvement Against Soft Errors". *Proceedings of the 5th ACM International Conference on Embedded Software (EMSOFT '05)*, pages 203–209, 2005.

[144] M. Yang, H. Lei, Y. Liao, and F. Rabee. "PK-OMLP: An OMLP Based k-Exclusion Real-Time Locking Protocol for Multi-GPU Sharing under Partitioned Scheduling". *2013 IEEE 11th International Conference on Dependable, Autonomic and Secure Computing (DASC'13)*, pages 207–214, 2013.

[145] X. Yang, S. Deka, and R. Righetti. "A hybrid CPU-GPGPU approach for real-time elastography". *IEEE Transactions on Ultrasonics, Ferroelectrics and Frequency Control*, 58(12):2631–2645, 2011.

[146] K. Zhang, J. Hu, and B. Hua. "A Holistic Approach to Build Real-time Stream Processing System with GPU". *Journal of Parallel and Distributed Computing*, pages 44–57, 2015.

[147] Y. Zhang, S. Chen, L. Peng, and S. Chen. "NBTI alleviation on FinFET-made GPUs by utilizing device heterogeneity". *Integration, the VLSI Journal*, 51:10–20, 2015.

[148] Y. Zhu, Y. Deng, and Y. Chen. "Hermes: An integrated CPU/GPU microarchitecture for IP routing". *Design Automation Conference (DAC'11)*, pages 1044–1049, 2011.