electronics

MDPI

Article

Prediction-Based Error Correction for GPU Reliability with Low Overhead

Hyunyul Lim, Tae Hyun Kim and Sungho Kang *

Department of Electrical and Electronic Engineering, Yonsei University, Seoul 120749, Korea; lim8801@soc.yonsei.ac.kr (H.L.); incendio9@soc.yonsei.ac.kr (T.H.K.)

* Correspondence: shkang@yonsei.ac.kr; Tel.: +82-2-2123-2775

Received: 7 September 2020; Accepted: 30 October 2020; Published: 5 November 2020

Abstract: Scientific and simulation applications are continuously gaining importance in many fields of research and industries. These applications require massive amounts of memory and substantial arithmetic computation. Therefore, general-purpose computing on graphics processing units (GPGPU), which combines the computing power of graphics processing units (GPUs) and general CPUs, have been used for computationally intensive scientific and big data processing applications. Because current GPU architectures lack hardware support for error detection in computation logic, GPGPU has low reliability. Unlike graphics applications, errors in GPGPU can lead to serious problems in general-purpose computing applications. These applications are often intertwined with human life, meaning that errors can be life threatening. Therefore, this paper proposes a novel prediction-based error correction method called Prediction-based Error Correction (PRECOR) for GPU reliability, which detects and corrects errors in GPGPU platforms with a focus on errors in computational elements. The implementation of the proposed architecture needs a small number of checkpoint buffers in order to fix errors in computational logic. The PRECOR architecture has prediction buffers and controller units for predicting erroneous outputs before performing rollback. Following a rollback, the architecture confirms the accuracy of its predictions. The proposed method effectively reduces the hardware and time overheads required to correct errors. Experimental results confirm that PRECOR efficiently fixes errors with low hardware and time overheads.

Keywords: GPGPU; GPUs; error correction; GPU reliability; data hazard

1. Introduction

High-performance computing (HPC) applications typically require massive amounts of memory and a huge number of arithmetic computations. Special accelerators and processors have been proposed to achieve massive parallel computing power [1–4]. However, these accelerators and processors are very expensive to manufacture and cannot be used for general purposes. On the other hand, graphics processing units (GPUs) contain a huge number of computation and memory units. Due to their highly parallel structure, recent GPU researches have focused on general-purpose applications in the high-performance computing (HPC) field [5]. Therefore, GPUs are widely used as parallel computing accelerators in big data processing applications currently. Because big data processing applications have become increasingly intertwined with humans, the reliability of GPUs designed for general applications has become increasingly important, and such approaches are referred to as general-purpose computing on graphics processing units (GPGPU).

However, because current GPUs are designed for creating images to be output to a display device, they lack hardware support for detecting and correcting errors in combinational logic. Therefore, incorrect results can be easily generated by GPUs, particularly in combinational logic. In GPU architectures, memory storage is protected by error correction codes (ECCs), but combinational logic is

not protected. This is so because storage structures have regular patterns that can be protected via parity and ECCs, whereas combinational structures exhibit irregular patterns. Therefore, combinational logic outcomes cannot be easily protected via parity and ECC approaches.

Furthermore, errors on GPUs are less critical in the control flow than these on CPUs. Because of the functionality of GPUs, the area portion of the control flow units on GPUs is smaller than that on CPUs. Moreover, the number of the instruction cache which can affect the control flow does not increase on GPUs even though the problem size increases [6]. Therefore, the functional interruption (FI) rate on GPUs is lower than that on CPUs and the SDC rate on GPUs is much higher than that on CPUs [6,7]. Additionally, the increases in the SDC rate on the GPGPU system cause the higher probability of obtaining incorrect outcomes in scientific applications. Therefore, methods for correcting SDC errors in GPGPU architectures are becoming increasingly important.

Error detection and correction methods have always been imperative in space and nuclear applications, but they have recently become more important in a wide variety of fields. As technological devices continue to shrink, circuits are becoming more susceptible to radiation effects and electromagnetic emissions owing to reduced node capacitance [8]. Furthermore, to meet low-power requirements, the frequency and voltage settings of modern systems on chips (SoCs) are designed to reach the edges of their performance limits. Because SoCs have become increasingly common in various fields affecting daily life, errors in SoCs have in turn become increasingly dangerous for humans. Thus, appropriate error detection and correction methods are essential for meeting the reliability demands of modern SoCs.

This paper proposes the approach called the Prediction-based Error Correction (PRECOR) for GPU reliability which predicts errors, checks prediction accuracy by re-executing instructions, and corrects faulty prediction results. To test the efficacy of the proposed method, simulated PRECOR solutions are compared with solutions obtained via dual and triple modular redundancy (DMR and TMR, respectively). The area and time overheads required for correcting errors are compared to prove the efficiency of the proposed method. The area overhead is reduced by 7% compared with the DMR methods. In addition, the time overhead is also reduced up to 10%.

The remainder of this paper is organized as follows. Our major motivations are discussed in Section 2. Section 3 presents the methodology and hardware details of the PRECOR method. Experimental setups and results are discussed in Section 4. Section 5 summarizes our conclusions.

2. Motivation

In this section, demands of soft error resilience techniques for GPUs is explained. First, GPU architecture is described in Section 2.1. Then, the present error resilience support in GPGPUs is presented in Section 2.2. In Section 2.3, related works are explained.

2.1. GPU Architecture

Figure 1a presents the Fermi GPU architecture [9]. A GPU architecture consists of a scalable number of streaming multi-processors (SMs). An SM consists of streaming processors (SPs) for arithmetic calculations, special function units (SFUs) for sine, cosine, and square root functions, load/store (LD/ST) units for memory operations, and several register file banks for caches.

SMs are designed to implement the single instruction, multiple threads (SIMT) execution model. A warp consisting of 32 individual threads is executed within a single SM block. SIMT execution is a lockstep execution mechanism that executes a given set of operations simultaneously. In the pipeline stage, the 32 threads of a warp run simultaneously with the same instructions. Each warp is scheduled by a two-level scheduler that checks the warp ID, the active mask, and a single program counter. This two-level scheduler deploys ready warps in the fetch stage. Each thread in a warp is executed in lockstep in each SP. There are two types of thread barrier instructions for preventing the data race condition. One is used for individual warps and the other is used for all threads.

These instructions ensure that a thread in a barrier will not pass the barrier until all concurrent threads have been completed.

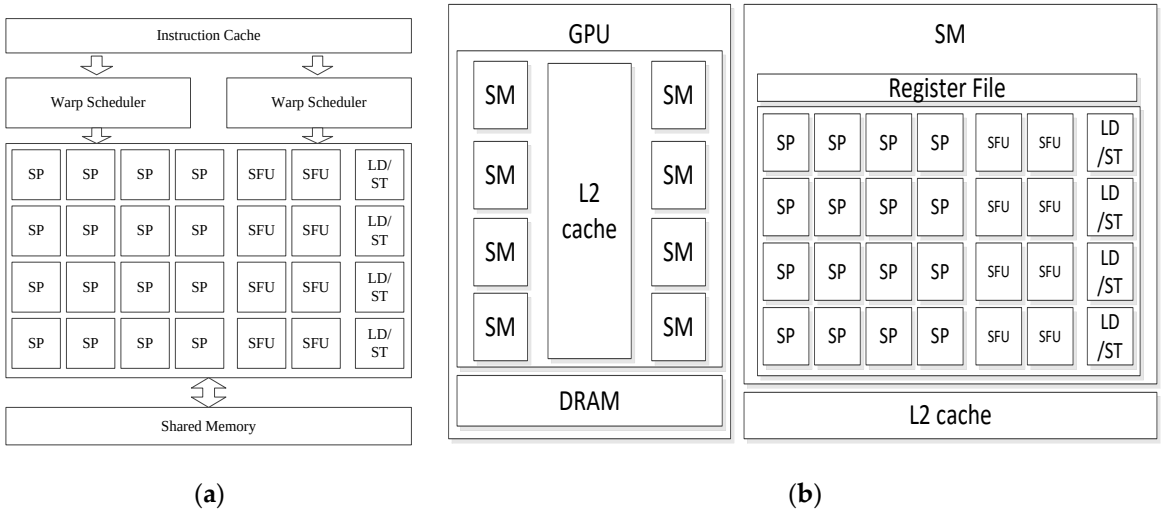


Figure 1. Overview of the Fermi graphics processing unit (GPU) architecture and memory scheme: (**a**) GPU architecture; (**b**) Memory scheme.

Caches consist of hierarchical structures. Figure 1b presents a typical cache structure. The global memory of a GPGPU system is deployed in dynamic random-access memory. Every SM has access to all global memory. L2 cache is shared memory, which is shared by the SPs in the SMs. Each SM executes read and write instructions at the L2 cache level. The L2 cache size is 768 KB. The register file is accessible to all SPs in an SM. This register file is mapped in the SMs to improve computational performance by caching data for the threads running on each SM.

2.2. Resilience Support in GPU Architecture

Soft error resilience is becoming more and more important in GPU than in CPU. GPUs have a relatively high error rate. For example, 1.8% of commodity GPU devices have a least one permanent fault [10], and some GPUs were determined to have experienced transient memory faults at a rate of 66% during evaluations in an HPC cluster environment [11]. Most analyses of GPGPU failures have focused on memory faults, and combinational errors have not yet been extensively studied. Because a large portion of the total GPGPU silicon area is used for GPU cores [12], a non-negligible number of errors can be expected to occur in the cores of GPUs. However, unlike memory errors, which can be detected via ECCs, it is difficult to find errors that occur in GPU cores (e.g., floating-point units, arithmetic logic units (ALUs), local memory, or registers). Furthermore, miniaturization has increased error rates in hardware, especially those of transient errors [12]. Thus, it is necessary to develop a method to increase hardware reliability to avoid data corruption. In a previous research, a detailed survey of GPU errors in the Titan supercomputer was presented to analyze the reliability of the GPGPU architecture [13]. The authors gather the error history of Titan using simple event correlators on software management workstations and could highlight many GPU memory errors and GPU software/firmware-related errors, but they are unable to collect silent data corruption (SDC) errors that occurred in the processor. Therefore, analysis of SDC by the architecture-level fault injection has been studied [14]. In addition, error propagation and its effects on processing cores and memory have been studied [15]. SDC is widespread in GPGPU kernels and propagates through memory states via data corruption.

G. Li et al. [15] measures SDCs and benign errors by comparing the output memory of fault injection run with that of the golden run. This corresponds to data recorded in output memory (OM) after the programs finish their executions. On average, crashes comprise 17.52%, SDCs comprise

18.98% and Benign errors comprise 63.35% of all injections. Finally, more than half of fault injections except benign fault injections are SDCs. However, on CPUs, the ratio of SDCs and crash is 36.11% (13% of SDCs and 23% of crash) [7]. SDC is dependent on the application, but such SDC errors certainly affect the output results. Nearly 50% of the errors that occur in a GPU corrupt output data via SDC.

However, GPUs only have resilience support for memory storage. A single-error-correction, double-error-detection (SECDED) ECC is included in the GPU device memory, L2 cache, instruction cache, register files, shared memory, and L1 cache regions [9]. However, not all the blocks in GPUs are protected because SECDED only tracks memory reads and writes. Therefore, other blocks, such as logic blocks, queues, the two-level scheduler, the thread block scheduler, the instruction dispatch unit, and the interconnecting network are vulnerable to errors. Such errors can have different types of effects on circuits: (1) no effect on the program output (the failure does not affect any of the outputs), (2) program crash, or (3) SDC error (incorrect output, but the program does not crash).

One of the keys to the resilience of some applications is that soft errors may not affect output states. At the microarchitecture level, there are two factors that determine the soft error rate of a hardware structure [16]: the failures-in-time (FIT) and the architecture vulnerability factor (AVF) [17]. The FIT rate is the raw soft error rate (SER) per bit. It is dependent on process technology and circuit design. The AVF represents the effects of the SER, meaning that a soft error that affects the output data and damages a process can lead to crashes and SDC. Therefore, the AVF is dependent on the applications and instructions in which errors occur and, thus, we will not focus on such errors and their effects.

However, the other issues discussed above are crucial. Regardless of the error resilience problems in GPGPU systems, there may not be hardware support for detecting and correcting errors without ECCs. However, many recent applications on GPGPU systems are business-critical, long-running, and financially sensitive. Therefore, a single error can have a serious impact on users.

2.3. Related Works

Error detection and correction in combinational logic structures are traditionally performed using modular redundancy, by which the same instructions are executed several times and errors are detected by comparing the results.

Dual Modular Redundancy (DMR) is usually based on neighboring dual cores that make synchronization, transfer, and comparison of the results upon error detection. The dual cores are synchronized by special link and the results of the dual cores are compared upon error detection [18]. DMR only detects errors in GPU lanes and requires correction schemes for recovering any identified errors.

Triple Modular Redundancy (TMR) is usually based on the neighboring triple cores that make synchronization, transfer, and comparison of the results upon error detection and collection [19]. If an error occurs, the three results are compared with each other and two same results are selected to the correct result. In a TMR design, each logic element is designed in triplicate and majority voters are inserted after each register stage to remove logic upsets.

DMR threads are deployed with branch divergence, meaning that any remaining cores are idle. Warped-DMR [20] is a DMR-based technique that exploits these underutilized GPGPU resources (i.e., the many idle cores) for duplicating threads and detecting errors.

Warped-RE [21] is a fault-tolerant technique based on both DMR and TMR. DMR is an opportunistic approach that checks results using native redundancy instructions in the GPU lane. After error detection via DMR, TMR is used for error correction. Therefore, the DMR and TMR logics are also required. Additionally, opportunistic DMR threads are difficult to detect and can only be obtained by deploying additional control logic, which requires more than 1.5% hardware overhead and performance overhead by two additional pipeline stages only for searching opportunistic thread within total threads.

The Clover scheme [22] introduced fault detection and correction techniques for sensing the waves created by particle strikes. These techniques detect errors using sensors that react to particle strikes and restart erroneous processes in idempotent regions. However, idempotent-region checkpoint regions

(CRs) have a critical drawback. If errors occur at the end of these regions, the CR cannot restart at the proper point because of the latency of the sensing process. Clover employs a method called Tail-DMR to compensate for this drawback by resetting the program to the beginning of the code region where the error was detected. However, this method only detects errors related to particle strikes, and the hardware overhead required for particle strike sensors is unsustainable.

Argus-G [23] is another error detection architecture for GPGPU cores. It is an extension of the Argus [24] architecture for CPU cores. Argus detects errors in three basic invariants, namely control flow, computation, and dataflow, based on signatures to detect errors in GPGPU cores. However, significant area overhead is required for generating signatures and comparing them with optimal values.

The proposed PRECOR method detects errors by combining partial and temporal redundancies and corrects them using an error prediction buffer. This approach is based on warp-level DMR, which duplicates and verifies instructions. The use of this DMR scheme improves the performance overhead of PRECOR by nearly 100%. Additionally, the controller and checkpoint buffer are much simpler in PRECOR compared with other competitive schemes.

3. Proposed Methodology

This section presents the main concepts of PRECOR. The proposed method exploits space and time redundancies in GPU systems. Sections 3.1 and 3.2 describe the PRECOR algorithm and present the architecture of PRECOR in detail, respectively.

3.1. The PRECOR Approach

In most approaches, erroneous results are corrected via traditional DMR using a checkpoint method. The checkpoint buffers that are deployed in the pipeline stages store the previous state of the data and recover that state after an error is detected. These checkpoint buffers require massive overhead for duplicating all the data in the pipeline buffers. Furthermore, when restoring a state during the recovery process, the entire system must be halted, which increases execution time.

To reduce execution time and the overhead of the checkpoint buffers, PRECOR implements an error prediction method based on historical data. Figure 2 illustrates the detection and correction flow of PRECOR. A thread is fetched at two cores, an original core and a redundancy core. Then cores decode and execute the thread. After execution, a comparator compares the two outcomes. This comparison is for checking an error occurrence. If outcomes are not the same, an error occurs to one of two cores. So, if outcomes are the same, the process keeps continuing. If not, the prediction controller anticipates the outcomes with a history of the cores. Then, the process keeps continuing with anticipated outcomes and the error occurrence outcome is saved in the buffer of prediction controller. During this process, the error occurrence instruction is issued to the core. After the execution of the re-issued instruction, the re-executed outcome is compared with the outcome in the prediction controller. If these are the same, the output value is changed to the re-executed outcome. If not, the process is continued. However, since the proposed method corrects an error by comparing values generated from the same two cores, permanent errors that continuously output a specific error value at any moment cannot be corrected, therefore, these error values cannot be corrected through the proposed method. So, only transient errors are targeted on the proposed method.

To better explain the PRECOR method and avoid confusion, the following four terminologies are introduced to make the approach easier to understand.

Anticipated incorrect output (AIO): The output that is assumed to be incorrect by the prediction controller and kept in the buffer.

Anticipated correct output (ACO): The output that is assumed to be correct by the prediction controller and executed on the pipeline.

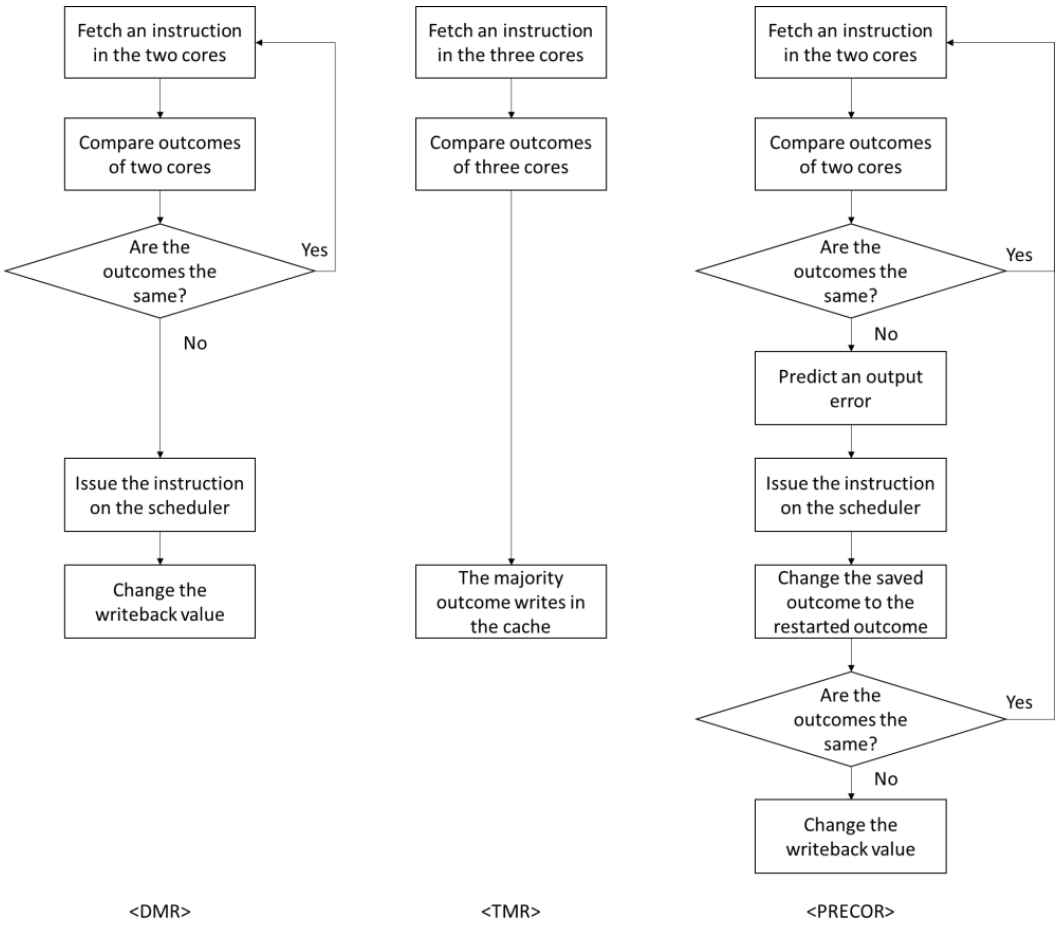


Figure 2. Flowchart of the proposed method.

Prediction success: The pipeline executes the next instruction with the correct/incorrect values. It does not matter whether the values are correct or not.

Prediction failure: The pipeline returns to the process and changes the values. It does not matter whether the values are correct or not.

PRECOR performs error detection on real GPUs by an efficient error correction using roll-back with DMR. Thus, PRECOR is the same as DMR in detecting an error. In correction, it acts like TMR by keeping the result in the buffer. In the first stage, the same instruction is fetched into two cores, an original core and a redundancy core, similar to the initial steps of DMR. After the instruction is executed, the two resulting values from the two cores are compared to check for errors. If the results are mismatched, an error has occurred. To correct an error, PRECOR uses a restart scheme, meaning that the instruction that resulted in the error is re-issued in the pipeline stage. Meanwhile, the controller identifies the core that can cause the error via historical prediction and the next instruction continues with the ACO value which means that the pipeline continues despite an error occurrence. The cores in which errors occur are marked with a latch and the controller chooses the core based on the state of a latch. However, this ACO must be verified.

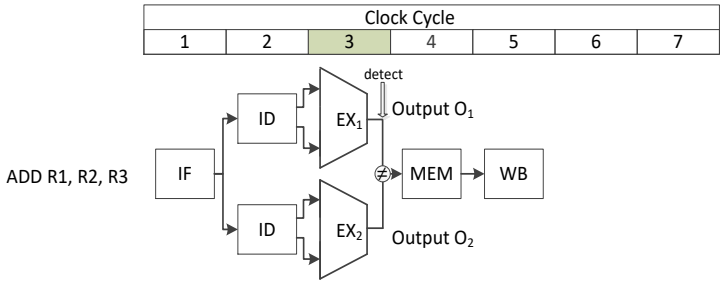
To verify the output, the method must keep the output value in buffer. In the proposed method, there are two ways to select the output value for verification. The first way, called ACO strategy, is that the ACO value is kept in the buffer and the next instruction continues despite an error occurrence. Thus, the next instruction continues with the ACO value, chosen by the control unit using the latches of the two modules. The ACO is also stored in the buffer in the ACO strategy if the ACO value and the new output value from re-executed instruction are matched in the ACO strategy. However, if they are mismatched, the ACO value is the incorrect output value. In contrast, the AIO strategy is that the

AIO value is kept in the buffer and the next instruction continues with the ACO value. If the AIO value and the new output value from re-executed instruction are mismatched, the ACO value is the correct output value. If the ACO value is found to be incorrect after comparison with the new output value from re-executed instruction, the new output value from re-executed instruction is written to the register in the WB stage and the pipeline stalls.

In the case of GPU, instructions can be classified into FP arithmetic instructions, Integer operations, and memory operations. Therefore, the rollback operation is performed differently depending on the type of instruction. First of all, the error detection and rollback operation are performed only for integer instructions and FP instructions. Because memory instructions are protected by ECC. In the case of branch instructions, when an error occurs, it affects the process flow, that is, the progress of the next instruction, and unlike the FP instructions or integer arithmetic instructions, the operation must be stopped and rollback must be performed. Moreover, there is a rollback operation already for branch misprediction, so if an error occurs during branch operation, it can be operated again as if it is misprediction.

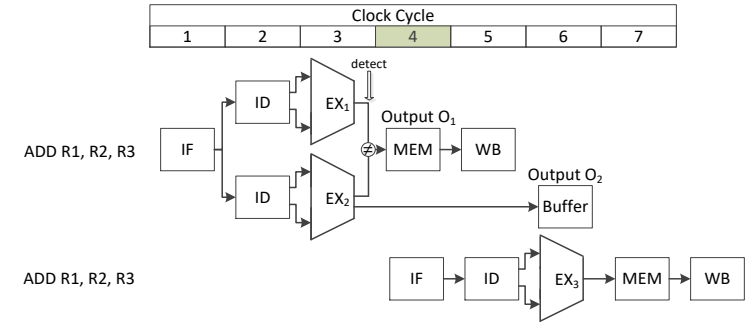
As mentioned in the explanation of terminologies, the final output value of the prediction success may be also incorrect. If two of the three results are incorrect, these errors cannot be corrected exactly in TMR strategy. However, these cases rarely happen. If the probability of an error occurrence in a module is P_m, the probability of more than two errored results of three results is $3{P_m}^2 + {P_m}^3$. Additionally, it rarely occurs that the erroneous output values have the same value, except stuck-at fault cases. In addition, the case that all outputs from two modules have errors rarely happens. As studied in [6], the SDC rate in GPUs is about $(1.80 + 0.39) \times 10^1 - (1.04 + 0.32) \times 10^3$ FIT (errors/10^9 h). Therefore, it is highly unlikely to have more than one corruption during a single execution in the natural radioactive environment. Thus, PRECOR chooses the AIO strategy because retaining the AIO is faster than retaining the ACO.

The example of PRECOR is demonstrated in the general single-instruction multiple-data (SIMD) pipeline stage, which consists of five sub-stages, namely instruction fetching, instruction decoding (ID), execution (EX), memory (MEM), and write-back (WB). Figure 3 illustrates the detection and correction example of PRECOR. In the example, it is also assumed that there is a precedent that an error occurred in 1000 instructions before on EX_2 and no error occurred on EX_1. Errors are detected in the EX stage, which outputs values as shown in Figure 3a. In an error-free case, the same instruction is fetched and executed in two units. The output values O_1 and O_2 from execution units EX_1 and EX_2, respectively, are compared for error detection.

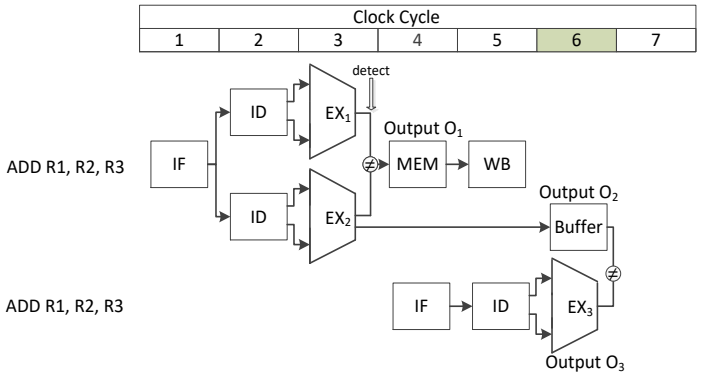


(**a**)

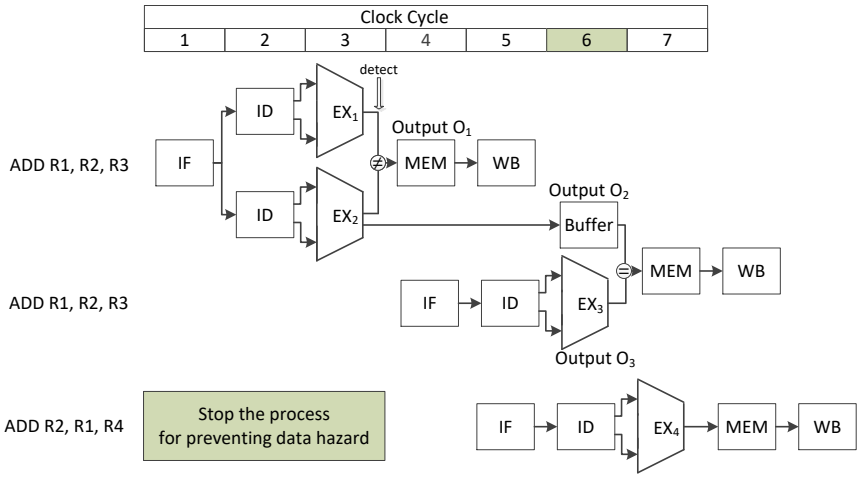
Figure 3. *Cont.*



(**b**)



(**c**)



(**d**)

Figure 3. Example of a PRECOR execution. (**a**) The error detection step of PRECOR in pipeline stage. (**b**) The roll-back step of PRECOR in pipeline stage. (**c**) The recovery step of PRECOR in pipeline stage, prediction success case. (**d**) The recovery step of PRECOR in pipeline stage, prediction failure case.

If the two outputs are mismatched, the result is rechecked in one pipeline while another pipeline executes the next instruction with the predicted correct results as described in Figure 3b. Therefore, the same instruction is fetched for rechecking and is executed in the execution unit, EX_1, which generates the ACO. PRECOR assumes that the value on the right side is incorrect the first time this happens. After that, the value that comes from the last error-causing core (EX_2) is assumed to be the

AIO. Because the output values O_1 and O_2 are different, the same instruction is fetched and executed in a different execution unit, EX_3 as described in Figure 3b. In the proposed correction method, O_2 is picked as AIO and kept in the buffer while the other pipelines keep running using the output value O_1.

After the execution of the re-fetched instruction (EX_3), the output value (O_3) from the re-fetched instruction and the output value in the buffer (O_2) are compared. If O_2 and O_3 are different, the output values O_1 and O_2 are correct as shown in Figure 3c. Consequently, the pipeline continues. Since O_2 and O_3 are the same, the output values O_2 and O_3 are correct as shown in Figure 3d. Because the prediction fails, the latches of each module are toggled by the prediction control unit. Meanwhile, the pipeline stops until the new value (O_3) is written in the memory.

PRECOR chooses the AIO strategy because retaining the AIO is faster than retaining the ACO. Because the AIO strategy has fewer pipeline stalls than the ACO strategy without any reliability loss, all erroneous output cases of each method are shown in the Table 1. For the erroneous output values in the first column of Table 1, the second, third, and fourth columns show the output, the prediction result, and the correctness of the output in the TMR, ACO and AIO strategies, respectively. The "Output", "Prediction" and "Result" columns of each strategy show the final output written to the register, the result of the prediction and the correctness of the final output, respectively. As shown in Table 1, the ACO strategy fails in five cases, whereas the AIO strategy fails only once. Therefore, AIO strategy is faster than the ACO strategy because AIO strategy is less stalling the pipeline during the rollback. Moreover, the reliability of the AIO strategy in the PRECOR method is the same as that of the PRECOR ACO strategy. Therefore, the AIO strategy is more efficient than the ACO strategy.

Table 1. All error cases in the triple modular redundancy (TMR) and Prediction-based Error Correction (PRECOR) methods when retaining different outputs.

Erroneous Output	TMR		Anticipated Correct Output (ACO)			Anticipated Incorrect Output (AIO)		
	Output	Result	Output	Prediction	Result	Output	Prediction	Result
O_1	O_2	Correct	O_3	Failure	Correct	O_3	Failure	Correct
O_2	O_1	Correct	O_1	Success	Correct	O_1	Success	Correct
O_3	O_1	Correct	O_1	Success	Correct	O_1	Success	Correct
O_1, O_2	X	Incorrect	O_3	Failure	Correct	O_1	Success	Incorrect
O_1, O_2	X	Incorrect	O_3	Failure	Incorrect	O_1	Success	Incorrect
O_2, O_3	X	Incorrect	O_3	Failure	Incorrect	O_1	Success	Correct
O_1, O_2, O_3	X	Incorrect	O_3	Failure	Incorrect	O_1	Success	Incorrect

Table 1 shows all error cases in the TMR and PRECOR. EX_1, EX_2, and EX_3 are assumed to be independent and identical execution units. Therefore, each module has the same error occurrence rate p and the reliability can be calculated using the binomial theorem as follows:

$$P(r;n,p) = \binom{n}{r} p^r (1-p)^{n-r} \quad (1)$$

r is the number of the erroneous outputs, n is the number of total outputs, and p is the error occurrence rate, respectively. For example, the cases of two erroneous outputs out of three outputs are (O_1, O_2), (O_1, O_3), and (O_2, O_3). Thus, the probability of the cases is $R = \binom{3}{2} p^2 (1-p)^1$. According to Table 1, zero erroneous output or one erroneous output out of three outputs are correct in TMR. Additionally, the reliability of the voter (R_V) must be considered for the right selection in TMR. It is also

assumed that all voters are independent and identical, and each voter has the same error occurrence rate p_v. Therefore, the reliability of TMR [19] is:

$$\begin{aligned} R_{TMR} &= \binom{3}{1} p^1 (1-p)^2 p_V^3 + \binom{3}{0} p^0 (1-p)^3 p_V^3 \\ &= (1-p)^3 \, (1-p_v)^3 + 3 \times p(1-p)^2 (1-p_v)^3 \end{aligned} \tag{2}$$

Compared with TMR, erroneous output cases (O_1, O_2) and (O_2, O_3) are also correct in PRECOR ACO and PRECOR AIO, respectively. In addition, the reliability of the voters is not considered because the voters are not required in PRECOR. Therefore, the reliability of PRECOR (ACO and AIO) is

$$\begin{aligned} R_{PRC} &= \binom{3}{3} p^0 (1-p)^3 + \binom{3}{2} p^1 (1-p)^2 + p^2 (1-p)^1 \\ &= (1-p)^3 + 3 \times p(1-p)^2 + p^2 (1-p)^1 \end{aligned} \tag{3}$$

PRECOR assesses whether or not an error occurs in a core based on historical prediction results. As technological devices continue to shrink, cores become more vulnerable to outside influences. Technology scaling has also increased process variation [11], meaning that some cores are more affected by single-particle strikes or other environmental effects than others. Soundararajan et al. [16] demonstrated that errors occur more frequently in more sensitive cores.

To avoid writing an erroneous result on a shared cache when using this method, a data-hazard prevention scheme is implemented because similar situations occur in data hazard cases. By providing microarchitectural and compiler support for data hazards, PRECOR can effectively fix errors. If an error-affected cache is read from or written to, the re-launched instruction is terminated, and the read/write instruction is halted by the detector. Once the correct value is obtained for the re-launched instruction, the value in the cache is changed and the instruction restarts.

3.2. Microarchitectural Support

Figure 4 illustrates the microarchitectural support for PRECOR. This SM architecture includes a prediction controller that manages the prediction and correction flows. In conventional instruction pipelining, the throughput speed of the cores is increased by implementing instruction-level parallelism. The states of the stages for establishing a pipeline are maintained in pipeline registers. If an error occurs, the corrupted thread is re-executed to correct the result. In traditional implementations, the state of an instruction is restored by checkpoint pipeline buffers. However, checkpoint buffers incur high area overhead in pipeline registers. To overcome this drawback, PRECOR implements a prediction controller that maintains a relatively simple instruction and position list rather than all thread contexts. GPUs issue parallel thread execution (PTX) instructions that execute many threads simultaneously on multiple GPU cores. Each PTX instruction is decomposed into 32 thread contexts for executing 32 threads in each SP. After the ID stage, the thread contexts are scheduled in each SP. Each SP is controlled by the operand collector. Therefore, the thread context which has to be executed on each SP is received from the decoder and is kept on the buffer of the operand collector until the thread context is executed on each SP as shown in Figure 4.

In the general DMR architecture, these thread contexts are maintained in an extra cache for error correction. Rather than the thread contexts themselves, the PRECOR architecture retains only the PTX instructions and the positions of thread contexts (i.e., pointers to thread contexts that find specific threads in decoded PTX instructions). After decoding a PTX instruction, the thread contexts that should be re-executed are selected by specifying the positions of the corrupted threads.

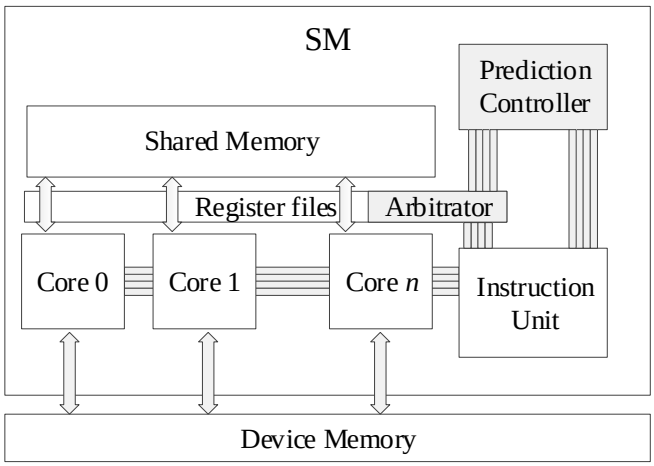


Figure 4. Overview of the proposed microarchitectural hardware support.

Figure 5 presents a decoder and a scheduler with a prediction controller. The prediction controller consists of the copied instruction cache and a position list. The copied instruction cache saves the PTX instructions for re-executing and the position list identifies thread contexts containing errors in their PTX instructions. At the marked positions of the corrupted thread contexts, the mask bits in the position list are set to one. Therefore, the scheduler can selectively launch threads whose bits are set to one.

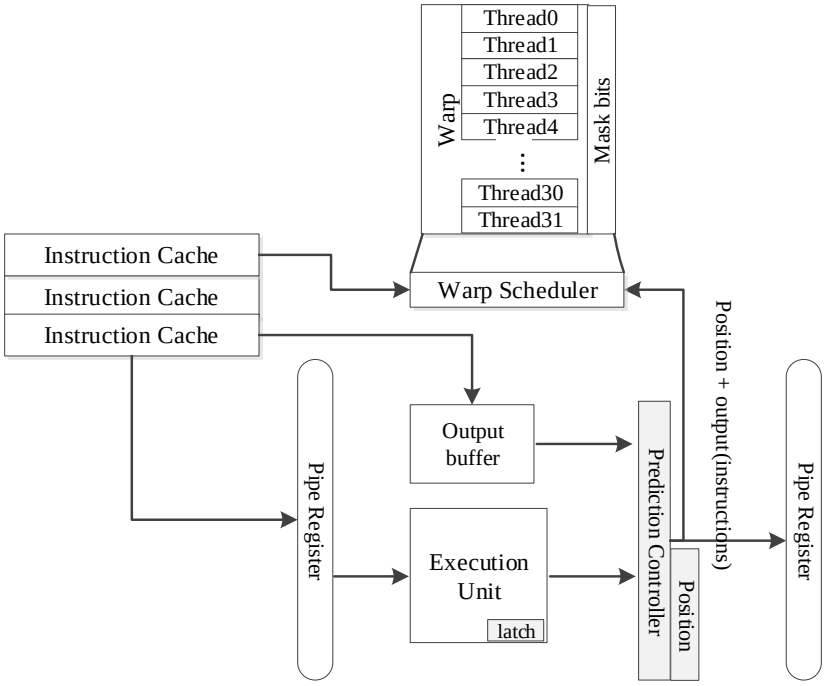


Figure 5. Prediction flow with pipeline registers.

Therefore, the correction flow of the PRECOR architecture proceeds as follows. When an error occurs, the prediction controller turns on the correction flow process. The erroneous instruction which has to be re-executed is re-fetched and decoded by the prediction controller using the fetch and decode units. Before scheduling the thread contexts, the prediction controller matches the mask bits using the position list. The mask bits control whether or not specific thread contexts are scheduled. Accordingly, the prediction controller masks 31 threads to transmit only previously erroneous thread context to the operand collector. Meanwhile, the other thread contexts on queues of the operand collectors in the GPU lane continue executing despite the re-execution of the erroneous instruction. The difference

between the error occurrence and the non-error situation is that only previously erroneous thread context is added on the queue of the operand collector in the specific SP which generates the ACO without halting the GPU lane. After the output of the re-executed instruction is compared with the buffer, the selected data is written in the register. During the correction flow, the PRECOR architecture retains the address of the destination register to prevent data hazards.

Figure 6 presents the flow for preventing data hazards in the GPU. The GPU uses the scoreboard algorithm to check for write-after-read and read-after-write dependency hazards [25]. PRECOR uses the scoreboard algorithm to remedy data-hazard dependency problems. The destination register address is obtained from the pipeline by the prediction controller and sent to the scoreboard. This address prevents the stalling of a specific thread that uses the destination register as its source register. Specific threads are selected to be re-executed and the destination registers are indicated for the scoreboard while other threads continue their pipelining processes. In contrast, existing methods prevent data hazards by simply stalling all threads during the correction flow.

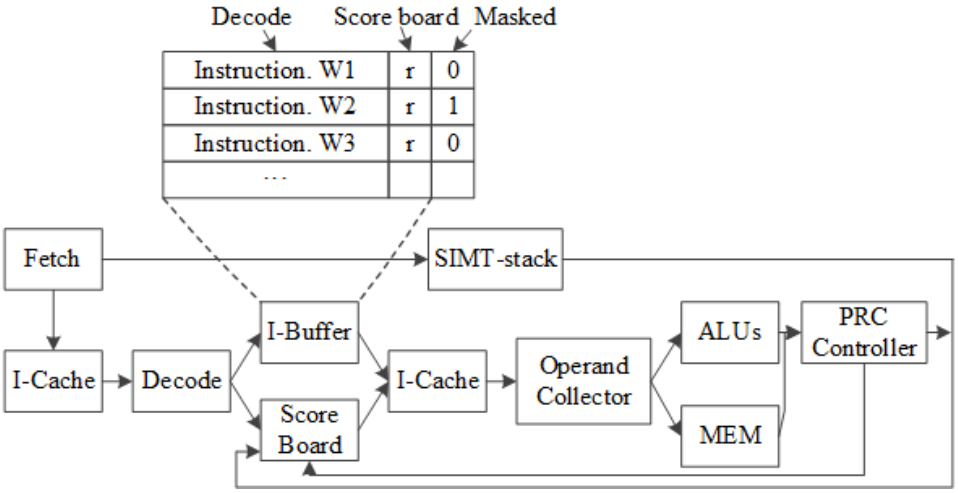


Figure 6. Single instruction, multiple threads (SIMT) core pipeline flow with a scoreboard data-hazard preventer.

4. Experimental Results and Analysis

The experimental results are obtained to evaluate the performance overhead, area overhead, and fault coverage of PRECOR. The experimental results are described in this section in comparison to previous methods: (i) DMR: full duplication re-execution of GPU instructions and correction via checkpoint buffers [18], (ii) TMR: full triplication re-execution of GPU instructions and correction via majority-voter rules [19].

4.1. Experimental Setup

To evaluate the performance of PRECOR, two simulation setup environments were employed. The GPGPU-sim was generally used for evaluating the performance overhead of methods. This application was modeled from commercial NVIDIA GPU units, Fermi architecture. GPGPU-sim simulation was not updated after Fermi architecture. Therefore, GPGPU-sim simulation only supports Fermi architecture. It can easily simulate the benchmarks of the simulation and analyze their results. However, it cannot check for fault coverage. GPGPU-sim v3.2.2 [25] was used to compare PRECOR with other methods. In this simulation, the GPGPU had 30 SMs, each consisting of 32 SIMT lanes. The SIMT lanes were grouped into four SIMT-lane clusters: four SPs, four SFUs, four LD/ST units, and four register banks. Several applications from rodinia_3.1 were selected as benchmarks [26]. As mentioned previously, our target applications (scientific computing and financial applications) demand very high accuracy.

To complement the GPGPU-sim simulation, Nyami open-source architecture was also used to build our experimental setup [27]. The DMR, TMR, and PRECOR architectures were designed using the Nyami architecture to evaluate the efficiency of the PRECOR architecture. Testbenches with the hash, dhrystone, and membench applications were built using this architecture. Figure 7 presents the Nyami GPGPU architecture. The Nyami architecture has four FIFO fetch units and a program counter (PC). Each thread is controlled by the mask signal and the thread-select stage. The 16 float and integer units are grouped and execute the same instruction at the same time as the vector units. The results of each instruction are stored in the same register file. Instructions for loading and storing 8-, 16-, and 32-bit scalars are supported. The Nyami architecture was implemented via SystemVerilog [28]. Several applications in the NyuziToolChain were selected as benchmarks for the testbenches.

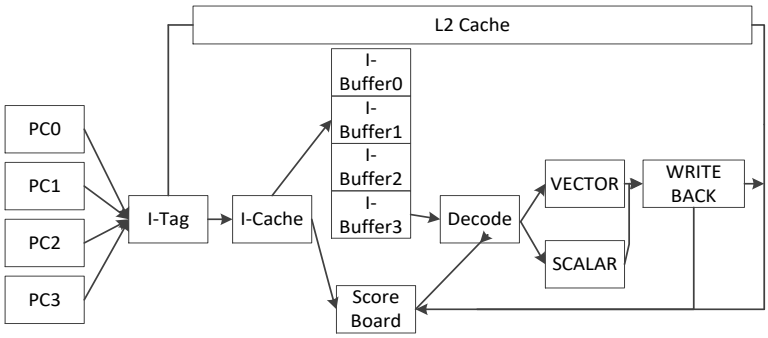


Figure 7. Nyami processor architecture.

4.2. Fault Coverage

To evaluate the fault coverage of PRECOR and the previous methods, they were designed to match a physical architecture using Nyami emulators [27]. The fault coverage of the methods cannot be checked on GPGPU-sim. Therefore, Nyami architecture was used for the fault-coverage simulations. The benchmark applications were executed on the fault-coverage test architecture using Nyami Processor and NyuziToolChain. Then, the faults were injected by modifying the emulators. To select instructions in the testbenches for the injection of transient faults, a fault-injection program was designed on the emulator using C++. An error was injected into the compiled thread at the frequency of the mean instructions between failures (MIBF).

For the simulations, the ptx instructions from the unchanged simulation have parsed twice or three times in ptx parse file except for the control flow instruction and the memory operations. Then, the application without faults is executed for obtaining the full thread logs in the application. Therefore, the full execution logs are gathered from the second run. Next, the erroneous threads are selected using the number of the threads and MIBF by the fault injection random function. The predictions of success and failure have been checked using the weighted random function in the erroneous thread selection. Then, the rollback operation is executed for each erroneous thread during the next simulation.

As shown in Figure 8, the fault coverage of PRECOR was almost the same as that of TMR because the number of masked threads increases when PRECOR re-executes instructions. Moreover, the fault coverage for the dhrystone benchmark using the DMR method was lower than for the other benchmarks. This is so because dhrystone contains much more arithmetic instructions than the other benchmarks, which can generate more faults in the simulations. Therefore, more simultaneous transient faults occur in the simulations, and simultaneous transient faults cause errors when using DMR methods. Thus, the dhrystone benchmark resulted in less fault coverage than the other benchmarks.

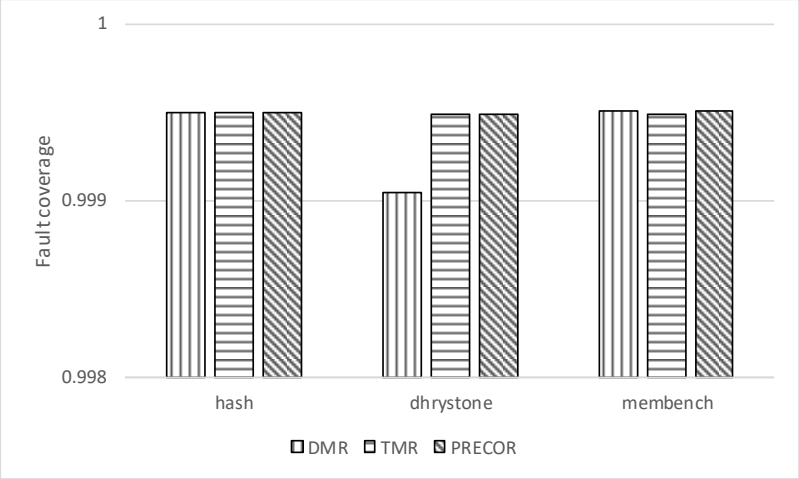
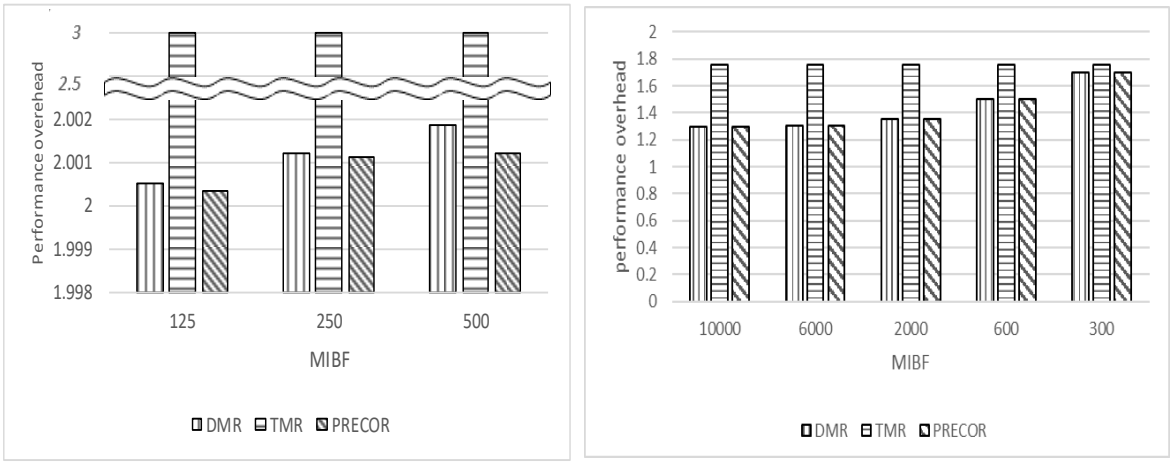


Figure 8. Comparison of fault coverage.

4.3. Comparison of Performance Overheads

Figure 9a shows the performance overheads of TMR, DMR, and PRECOR. The performance overheads of PRECOR and DMR increase as MIBF decreases, whereas the performance overhead of TMR is not changed regardless of the MIBF since TMR does not need the rollback operation when errors occur. Therefore, TMR is not affected by MIBF. However, DMR and PRECOR are affected by MIBF since the rollback operation is executed when errors occur. The performance overheads of PRECOR and DMR do not show much difference when the MIBF is large. Because the difference between DMR and PRECOR is the rollback overhead when an error occurs. Therefore, if MIBF is large and the number of the rollback operations is small, the difference in performance overhead between DMR and PRECOR also is small. However, as the MIBF decreases, the rollback overhead of PRECOR increases less than that of DMR.



(**a**) (**b**)

Figure 9. Performance overhead of Dual Modular Redundancy (DMR), TMR and PRECOR: (**a**) hash benchmark; (**b**) breadth first search benchmark.

Figure 9b describes the performance overheads of TMR, DMR, and PRECOR depending on the prediction success rate. Since the pipeline is stalled if the prediction fails, the simulation cycle decreases as the probability of prediction success increases. Therefore, the simulation cycle for the case of high prediction success rate is less than that of low prediction success rate.

The performance overhead depending on forecasting accuracy was evaluated based on an experiment on error injection and prediction simulated using C code. The target benchmark is the

breadth first search application on the GPGPU simulator. As shown in Figure 10, as the forecasting accuracy increases, the performance overhead decreases. Therefore, accurate prediction is important for reducing the time overheads of the error correction. However, forecasting data in real-world cases cannot be achieved, one can only assume the accuracy of the prediction. However, the prediction failure only happens in O_1 case on the proposed methodology as shown in Table 1. Therefore, the probability of the prediction success is $(1-p \times (1-p) \times (1-p))$ even if the results are not correct. Therefore, the proposed methodology can reduce the performance overhead by increasing the probability of the prediction success.

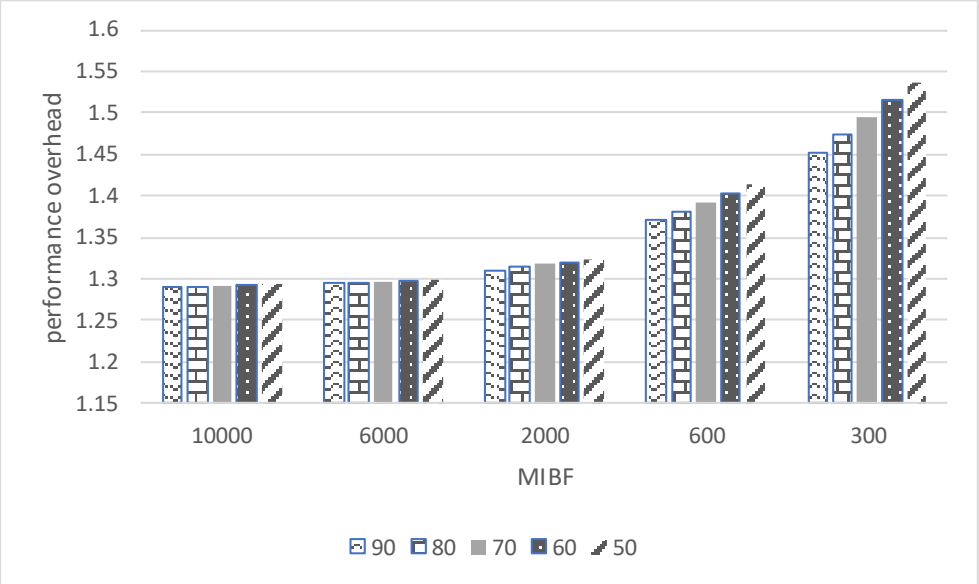


Figure 10. Performance overhead on breadth first search benchmark with various prediction probability.

Simulations on the Nyami architecture were implemented via the Nyami emulator. PRECOR selects results using history buffers to speed up the correction procedure. Therefore, the prediction accuracy when using history buffers was selected to make a performance comparison. In the simulation, prediction accuracy was set as the value that was referred to in the forecasting accuracy simulation.

Hash benchmark simulations were also performed for each MIBF 3000 times. As shown in Figure 9, the difference between DMR and PRECOR increases as the MIBF increases.

4.4. Comparison of Area Overheads

Hardware overheads were calculated based on a one-SM layout in the Synopsys CAD software. To synthesize additional hardware, Nyami architecture was implemented using the Synopsys Design Compiler and the saedEDK32.28 nm library (saed32rvt_tt0p85v25c.db). We do not add the redundant floating-point units and integer units for duplication with comparison. By combing the existing cores with comparators and voters, DMR and TMR have been generated. Therefore, original, DMR, and TMR cores have 18 lanes, 9 lanes and 6 lanes in the core, respectively. The results are listed in Table 2. Additional area overhead indicates that the area exceeded the available space in the original Nyami architecture. The total area is the entire area synthesized in the behavioral-level Nyami architecture using systemverilog code.

TMR needs six voters, DMR needs the pipeline state buffer and nine comparators, and PRECOR needs instruction buffer, history latch, and nine comparators. Since the pipeline buffer which saves the thread context is large, DMR architecture is required large area overhead. On the other hand, PRECOR architecture needs a 32-bit register to save the instruction, and the nine latches to keep history data. The area overheads were 7% lower for PRECOR than for the DMR method, but are larger than those of the TMR method. However, TMR has longer execution times than PRECOR.

Table 2. Area overhead comparison (nm^2).

	Additional Area Overhead	Total Area	Overhead (%)
Normal	0	768,824	0.00%
DMR [18]	56,388	825,213	6.83%
TMR	42,588	811,412	5.24%
PRECOR	52,974	821,798	6.44%

5. Conclusions

Recently, GPGPU systems have emerged in devices with strong parallel computing power for high-end applications. However, GPGPU systems are very vulnerable to soft errors. Streaming processors, which play important roles in data parallelism, critically determine the reliability of GPUs.

This paper proposes PRECOR as a low-cost error correction method for GPGPU architectures. PRECOR accelerates the correction process by choosing a correct value before additional instructions proceed, thereby avoiding additional errors. Therefore, the buffers for saving the outcome, and the prediction controller is the area overhead of the proposed method. However, the buffers for the rollback operation are reduced by the position buffers in the proposed method. Therefore, the total area overhead of the proposed method is less than DMR. Additionally, the rollback performance overhead is less than the conventional DMR because the entire process continues when the rollback operation executes in the proposed method, unlike DMR, which stops the entire process.

The experimental results show that PRECOR can more reliably correct soft errors compared with traditional error correction approaches. It also reduces the required time overhead by improving prediction accuracy, allowing processes to continue instead of halting. Finally, it reduces hardware overhead by 7% compared with DMR method.

Author Contributions: Conceptualization, S.K.; Funding acquisition, S.K.; Investigation, H.L. and T.H.K.; Software, H.L.; Supervision, S.K.; Validation, T.H.K.; Writing—original draft, H.L.; Writing—review & editing, S.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research was supported by Multi-Ministry Collaborative R&D Program (R&D program for complex cognitive technology) through the National Research Foundation of Korea (NRF) funded by MOTIE (2018M3E3A1057248).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Liu, L.; Zhu, W.; Yin, S.; Wei, S. A Binary-Feature-Based Object Recognition Accelerator With 22 M-Vector/s Throughput and 0.68 G-Vector/J Energy-Efficiency for Full-HD Resolution. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2018**, *38*, 1265–1277. [CrossRef]
2. Cousins, D.B.; Rohloff, K.; Sumorok, D. Designing an FPGA-accelerated homomorphic encryption co-processor. *IEEE Trans. Emerg. Top. Comput.* **2016**, *5*, 193–206. [CrossRef]
3. Wang, C.; Gong, L.; Yu, Q.; Li, X.; Xie, Y.; Zhou, X. DLAU: A scalable deep learning accelerator unit on FPGA. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* **2016**, *36*, 513–517. [CrossRef]
4. De Donno, D.; Esposito, A.; Monti, G.; Tarricone, L. GPU-based acceleration of MPIE/MoM matrix calculation for the analysis of microstrip circuits. In Proceedings of the 5th European Conference on Antennas and Propagation (EUCAP), Rome, Italy, 11–15 April 2011; pp. 3921–3924.
5. Owens, J.D.; Houston, M.; Luebke, D.; Green, S.; Stone, J.E.; Phillips, J.C. GPU computing. *Proc. IEEE* **2008**, *96*, 879–899. [CrossRef]
6. De Oliveira, D.A.G.G.; Pilla, L.L.; Santini, T.; Rech, P. Evaluation and mitigation of radiation-induced soft errors in graphics processing units. *IEEE Trans. Comput.* **2015**, *65*, 791–804. [CrossRef]
7. Saggese, G.P.; Wang, N.J.; Kalbarczyk, Z.T.; Patel, S.J.; Iyer, R.K. An experimental study of soft errors in microprocessors. *IEEE Micro* **2005**, *25*, 30–39. [CrossRef]
8. Constantinescu, C. Trends and challenges in VLSI circuit reliability. *IEEE Micro* **2003**, *23*, 14–19. [CrossRef]

9. Nvidia, C. Nvidia's next generation cuda compute architecture: Fermi. *Comput. Syst* **2009**, *26*, 63–72.
10. Shi, G.; Enos, J.; Showerman, M.; Kindratenko, V. On testing GPU memory for hard and soft errors. In Proceedings of the Symposium on Application Accelerators in High-Performance Computing, Urbana, IL, USA, 28–30 July 2009.
11. Haque, I.S.; Pande, V.S. Hard data on soft errors: A large-scale assessment of real-world error rates in gpgpu. In Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, Melbourne, Australia, 17–20 May 2010; pp. 691–696.
12. Borkar, S. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro* **2005**, *25*, 10–16. [CrossRef]
13. Tiwari, D.; Gupta, S.; Gallarno, G.; Rogers, J.; Maxwell, D. Reliability lessons learned from gpu experience with the titan supercomputer at oak ridge leadership computing facility. In Proceedings of the SC'15: International Conference for High Performance Computing, Networking, Storage and Analysis, Austin, TX, USA, 15–20 November 2015; pp. 1–12.
14. Santos, F.F.d.; Rech, P. Analyzing the criticality of transient faults-induced SDCS on GPU applications. In Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems, Denver, CO, USA, 12–17 November 2017; pp. 1–7.
15. Li, G.; Pattabiraman, K.; Cher, C.-Y.; Bose, P. Understanding error propagation in GPGPU applications. In Proceedings of the SC'16: International Conference for High Performance Computing, Networking, Storage and Analysis, Salt Lake City, UT, USA, 13–18 November 2016; pp. 240–251.
16. Soundararajan, N.K.; Parashar, A.; Sivasubramaniam, A. Mechanisms for bounding vulnerabilities of processor structures. In Proceedings of the 34th International Symposium on Computer Architecture (ISCA 2007), San Diego, CA, USA, 9–13 June 2007; pp. 506–515.
17. Biswas, A.; Racunas, P.; Cheveresan, R.; Emer, J.; Mukherjee, S.S.; Rangan, R. Computing architectural vulnerability factors for address-based structures. In Proceedings of the 32nd International Symposium on Computer Architecture (ISCA'05), Madison, WI, USA, 4–8 June 2005; pp. 532–543.
18. Sheaffer, J.W.; Luebke, D.P.; Skadron, K. A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors. In Proceedings of the Graphics Hardware, San Diego, CA, USA, 4–5 August 2007; pp. 55–64.
19. Mitra, S.; McCluskey, E.J. Word-voter: A new voter design for triple modular redundant systems. In Proceedings of the 18th IEEE VLSI Test Symposium, Montreal, QC, Canada, 30 April–4 May 2000; pp. 465–470.
20. Jeon, H.; Annavaram, M. Warped-dmr: Light-weight error detection for gpgpu. In Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture, Vancouver, BC, Canada, 1–5 December 2012; pp. 37–47.
21. Abdel-Majeed, M.; Dweik, W.; Jeon, H.; Annavaram, M. Warped-re: Low-cost error detection and correction in gpus. In Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Rio de Janeiro, Brazil, 22–25 June 2015; pp. 331–342.
22. Liu, Q.; Jung, C.; Lee, D.; Tiwari, D. Clover: Compiler directed lightweight soft error resilience. *ACM Sigplan Not.* **2015**, *50*, 1–10.
23. Nathan, R.; Sorin, D.J. Argus-G: Comprehensive, Low-Cost Error Detection for GPGPU Cores. *IEEE Comput. Archit. Lett.* **2014**, *14*, 13–16. [CrossRef]
24. Meixner, A.; Bauer, M.E.; Sorin, D. Argus: Low-cost, comprehensive error detection in simple cores. In Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007), Chicago, IL, USA, 1–5 December 2007; pp. 210–222.
25. Aamodt, T.M.; Fung, W.W.; Singh, I.; El-Shafiey, A.; Kwa, J.; Hetherington, T.; Gubran, A.; Boktor, A.; Rogers, T.; Bakhoda, A. GPGPU-Sim 3. X Manual. 2012. Available online: http://gpgpu-sim.org/manual/index.php/Main_Page (accessed on 4 November 2020).
26. Che, S.; Boyer, M.; Meng, J.; Tarjan, D.; Sheaffer, J.W.; Lee, S.-H.; Skadron, K. Rodinia: A benchmark suite for heterogeneous computing. In Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC), Austin, TX, USA, 4–6 October 2009; pp. 44–54.

27. Bush, J.; Dexter, P.; Miller, T.N.; Carpenter, A. Nyami: A synthesizable GPU architectural model for general-purpose and graphics-specific workloads. In Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, USA, 29–31 March 2015; pp. 173–182.
28. Nikhil, R. Bluespec System Verilog: Efficient, correct RTL from high-level specifications. In Proceedings of the Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004, MEMOCODE'04, San Diego, CA, USA, 23–25 June 2004; pp. 69–70.