Evaluating Software-based Hardening Techniques for General-Purpose Registers on a GPGPU

Evaluating Software-based Hardening Techniques for General-Purpose Registers on a GPGPU

Marcio M. Goncalves*, Jose Rodrigo Azambuja*, Josie E. Rodriguez Condia†, Matteo Sonza Reorda†, Luca Sterpone†
*Federal University of Rio Grande do Sul (UFRGS) - Institute of Informatics - PGMICRO
{mmgoncalves, jose.azambuja}@inf.ufrgs.br
†Politecnico di Torino - Department of Control and Computer Engineering DAUIN
{josie.rodriguez, matteo.sonzareorda, luca.sterpone}@polito.it

***Abstract*—Graphics Processing Units (GPUs) are considered a promising solution for high-performance safety-critical applications, such as self-driving cars. In this application domain, the use of fault tolerance techniques is mandatory to detect or correct faults, since they must work properly even in the presence of faults. GPUs are designed with aggressive technology scaling, which makes them susceptible to faults caused by radiation interference, such as the Single Event Upsets (SEUs), which can lead the system to a failure, and that is unacceptable in safety-critical applications. In this paper, we evaluate different software-based hardening techniques developed to detect SEUs in GPUs general-purpose registers and propose optimizations to improve performance and memory utilization. The techniques are implemented in four case-study applications and evaluated in a general-purpose soft-core GPU based on the NVIDIA G80 architecture. A fault injection campaign is performed at register transfer level to assess the fault detection potential of the implemented techniques. Results show that the proposed improvements can be tailored for different scenarios, helping engineers in navigating the design space of hardened GPGPU applications.**

***Index Terms*—Graphics Processing Units, fault tolerance, software-based hardening techniques**

I. Introduction

Graphics Processing Units (GPUs) have been originally designed for graphics applications, but soon evolved into general-purpose applications due to the high computing power and the advent of significant programming support. Over the last decade, the rapid proliferation of GPUs reached safety-critical applications, such as automotive. In this application domain, the use of fault tolerance techniques is mandatory to detect or correct faults, since safety-critical applications must continue to work correctly despite the existence of faults. However, the reliability of GPUs is still an open issue.

Increases in operating frequencies and transistor density in cutting-edge technology combined with the reduction of voltage supplies have made GPUs more susceptible to faults caused by radiation interference. Such faults, mainly caused by energized particles, make the newest GPUs prone to experience radiation-induced errors [1], [2], even on applications running at ground level, where neutrons are the primary source of soft errors [3], [4]. Among the most observed faults induced by radiation are the Single Event Upsets (SEUs), which affect memories and registers by causing bit-flips [5].

Modern GPUs are designed with a Reduced Instruction Set Computing (RISC) architecture that relies on large register files to perform high-performance computations over large blocks of data in parallel. In this sense, most instructions in a kernel operate directly on these registers, thus allowing a fault to propagate in the system easily. From a reliability point of view, the register file is a critical resource, as knowing the probability of an error in a register to propagate to the outputs may be sufficient to characterize the vulnerability of an application.

Software-based fault tolerance techniques have been proposed for GPUs in the past years, presenting high detection rates [6]. Their main goal is to protect the system against data flow errors, such as bit-flips in registers [7]. These techniques can be automatically applied to the source code of a program, thus simplifying the task for software developers: by protecting the system during software construction, the development costs can be reduced significantly [8].

This paper presents an evaluation of low-level software-based hardening techniques developed to detect SEUs in GPU register files and proposes optimizations to improve performance and memory footprint. Program codes are transformed at assembly-level and implemented in a soft-core General-Purpose GPU (GPGPU) based on the NVIDIA G80 architecture. Results in terms of execution time and program memory footprint are evaluated, as well as the fault detection rates.

II. Background

A. Related Work

Software-based hardening techniques used in GPUs are classified mainly in three classes: (1) naïve duplication, where the whole program is duplicated, (2) selective duplication, where only critical parts of the code are duplicated, and (3) Algorithm-Based Fault Tolerance (ABFT) [9], where particular types of algorithms can be protected.

A naïve duplication to detect faults in GPUs' general-purpose registers is proposed in [6], where the authors replicate the whole assembly code in an intertwined fashion and reach

This work has been partially supported by the European Commission through the Horizon 2020 RESCUE-ETN project under grant 722325, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES) - Finance Code 001, CNPq, and FAPERGS.

up to 99% error reduction at a performance cost of up to 78%. Selective duplication is able to lower costs in execution time and resource usage overheads by lowering the detection coverage, such as in [10], where a tool called Hauberk is able to reduce errors by 85% at a 15% execution time overhead, but only injected faults at protected high abstraction level variables. ABFT techniques are able to achieve high detection rates at low execution time overheads [11] but are limited to a specific group of applications.

This work extends [6] by proposing and evaluating improvements to save execution time and memory footprint. Also, the experiments presented in this work are performed in an optimized version of the FlexGrip, the soft-core GPGPU used in the previous work, which has been enhanced to correct some limitations and functional restrictions presented in the original version of the model [12].

B. FlexGrip Architecture

The FLEXible GRaphIcs Processor (FlexGrip) is an open-source soft-core GPGPU described in VHDL that implements the micro-architecture of the NVIDIA G80 Architecture [13]. FlexGrip is programmed using the CUDA programming environment and supports up to 27 instructions. The GPGPU consists of an array of Streaming Multiprocessors (SMs) that executes threads in parallel. Fig. 1 depicts the general organization of the SM in the FlexGrip architecture.

The SM can execute instructions in a parallel manner following the Single-Instruction Multiple-Thread (SIMT) paradigm [14]. The SM core is managed by a Block Scheduler Controller, which distributes the task workload to each SM available in the system. Internally, the SM is divided into a five-stage pipeline and includes one Warp Scheduler Controller managing and monitoring the concurrent execution of a group of 32 threads denoted as warp. Some pipeline registers are also present in the design and are located among the pipeline stages. The memory hierarchy in the GPGPGU is mainly composed of the General-Purpose Register File (GPRF), the Predicate Register File (PRF), the local memory, the constant memory, the shared memory, and the global memory.

The most relevant and used module in the memory hierarchy of a GPGPU is the GPRF, which contains the General-Purpose Registers (GPRs) and is used for every thread to store data operands and addresses during the program execution. The PRF has Predicate Registers (PRs) for every thread, which store the result of logic-arithmetic or comparison instructions, and each thread is assigned with four PRs. These registers are generally used by control-flow instructions to generate conditional or divergence paths. The local memory is mainly employed to store data arrays. Similarly, the constant memory is employed to store constant values for all threads during the execution of a program. The shared memory stores data operands that can be used among threads belonging to the same block. Finally, the global memory stores the initial inputs and the final results of a program kernel. These values are then retrieved by the host.

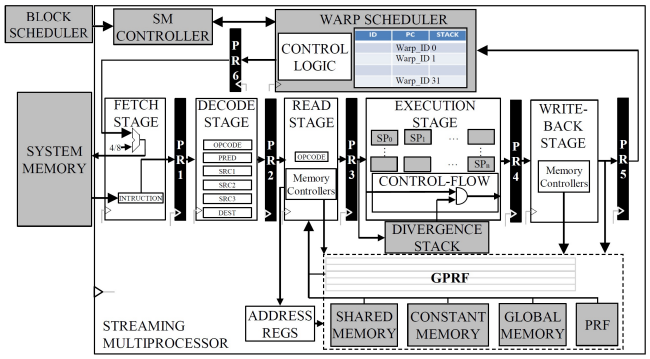

Fig. 1. A general scheme of the SM in FlexGrip.

III. Evaluated Fault Tolerance Techniques

The software-based fault tolerance techniques discussed in this paper aim to detect faults that affect specifically the GPRs and to inform the host when a fault is detected. Software-based techniques can be applied to the program code at different abstraction levels, such as at the CUDA level or at the assembly level. In our implementation, techniques are applied at the lowest level, in order to provide better control on the code transformation.

In order to detect a fault in the GPRF, static code analysis and three code transformations must be performed at the kernel assembly: (1) datapath instruction duplication, (2) consistency checking, and (3) host notification. The static code analysis is performed to find out which registers are being used by the application and which are not (spare registers). Then, a hash table is created, assigning a spare register as a copy register to each used register.

The first transformation (1) is responsible for duplicating all datapath instructions. This is the core of all discussed software-based techniques to detect faults in the GPRF, as it forces the hardware to execute twice the datapath in an intertwined fashion, being able to exploit Instruction Level Parallelism (ILP) from the GPGPU architecture much better than a simple duplication with comparison (which cannot take advantage of ILP). The replicated instructions also perform operations on the copy registers, completely separating the original and duplicated datapaths. In case there are not enough spare registers, selective hardening or register spilling must be performed.

The second transformation (2) is responsible for checking the consistency between the values of registers and their copies. To do so, it uses a comparison instruction, which then sets a PR in case of divergence. The main issue with checking register consistency is that it creates a dependency between both datapath flows, unifying both paths into the comparison instructions and therefore decreasing ILP gain. In this work, we evaluate the insertion of consistency checking after two classes of instructions: memory access and predicate register setting. The first directly affects the data being processed, while the latter affects the program's control flow.

The final transformation (3) notifies the host that an error has been detected. It is usually a memory write instruction, but could also be an exception signal to the host. These instructions are not executed on a correct application execution, and therefore do not usually cause performance degradation under normal circumstances.

In the following subsections, we will discuss two optimizations to code transformations and how we grouped them to evaluate the proposed techniques.

A. Move Optimization

Among all instructions in the FlexGrip ISA, the memory access ones (load and store) are the ones that require the most clock cycles to be executed. For example, a load instruction requires around four times more clock cycles than a move instruction. In this sense, aiming to decrease execution time and optimize performance, we propose the *Move Optimization*, which directly affects the datapath duplication (transformation 1) by replacing the replicated load instruction by a move instruction that copies the read value to the copy register.

While replicating a load instruction with a move instruction is able to reduce execution time, it inserts a point of failure on the software-based hardening technique. If a fault affects the register written by the load instruction before the move instruction can copy its value to the replicated registers, the corrupted value is also copied to the replicated register. In this case, both value and its replica would be corrupt, and the consistency checking transformation would not be able to signal a fault. It is important to notice that, even though the load instruction takes an increased amount of clock cycles to execute, only a fault that hits the instruction in its late write-to-register stage would actually upset the destination register, and therefore this point of failure is not as large as the time taken from fetch-to-fetch.

B. Conditional Optimization

As mentioned previously, the consistency check is done through a comparison instruction that sets a predicate register in case of divergence. The main issue is that it always overwrites the predicate registers and, therefore, the host notification must be performed immediately after the check. In this proposed *Conditional Optimization*, instead of performing a comparison, we propose to perform a conditional comparison (on the condition that a fault has not been detected). By doing so, the consistency checking will no longer overwrite the predicate register, and thus the host notification no longer has to be performed after each consistency checking. Thus, the *Conditional Optimization* aims to reduce the number of checks by postponing the host notification to some strategic point in the program. In this work, we choose the end of program execution as the point to notify the host if an error has been detected.

C. Implementation Groups

In order to evaluate the feasibility and effectiveness of the evaluated software-based hardening techniques, we implemented four groups of transformations: (I) No Optimization, without optimizations, (II) Conditional Optimization, with only the *Conditional Optimization*, (III) Move Optimization, with only the *Move Optimization*, and (IV) All Optimizations, with both *Move Optimization* and *Conditional Optimization*. Fig. 2 shows the four groups of transformations applied to a sample program code. The original instructions are represented in black, while replicated datapath instructions are presented in green, checking instructions in blue, and host notification instructions in red. The apostrophes on the registers of the replicated statements indicate that they are a register's replica.

The No Optimization group (I) enters the consistency checks and host notification for both comparison and memory access instructions just after these instructions. As one can see in Fig. 2, the original instruction 3, SETP, performs a comparison between registers R1 and R2, and stores the result in the predicate register P0. So, the two registers R1 and R2 must be checked to avoid an incorrect comparison result. In order to do so, a consistency check (instruction 4) is performed between register R1 and its replica R1', and the result is stored in the predicate register PE. The host notification instruction (instruction 5) is conditioned to the PE value, and will only be executed when the previous consistency check between R1 and R1' signals a divergence in PE. The same check and host notification are performed for register R2 and its replica R2' by the instructions 6 and 7, respectively.

In order to harden the memory access instructions, the No Optimization group (I) replicates the original load instruction (instruction 8) using replicated registers (instruction 9), and performs the consistency check (instruction 10) on these registers (R1 and its replica R1') while instruction 11 notifies the host in case of discrepancy. The original store instruction, GST, does not need to be replicated as we do not create replicas in memory. On the other hand, the consistency checks must be performed on its two registers R2 and R3, to avoid incorrect addressing and data. The consistency checks between the original registers R2 and R3 and their replicas R2' and R3' are, respectively, inserted through lines 13 and 15, while host notification instructions are inserted in lines 14 and 16.

The Conditional Optimization group (II) implements a conditional execution of the consistency checks to decrease the number of times that it is performed, possibly running it only once at the end of the program execution. In this case, the consistency checks are also placed at lines 4, 6, 10, 13, and 15, but the checks are conditioned by the negation of the result of a previous check (@!PE). Thus, if a divergence has already been detected previously, new consistency checks will not be performed, and the PE register will not be overwritten until the end of the program, where the host is notified. It is important to mention that the PE register must be initialized at the beginning of the program with a reset value.

The Move Optimization group (III), when compared to the No Optimization group, replaces the GLD replicas at line 9 by a move instruction, while the All Optimizations group (IV) implements both *Move Optimization* and *Conditional Optimization* into the same hardened code version.

Original Code	No Optimization	Conditional Optimization	Move Optimization	All Optimizations
1: ADD R1, R2, R3;	**1: ADD R1, R2, R3;**	**1: ADD R1, R2, R3;**	**1: ADD R1, R2, R3;**	**1: ADD R1, R2, R3;**
2:	2: ADD R1', R2', R3';	2: ADD R1', R2', R3';	2: ADD R1', R2', R3';	2: ADD R1', R2', R3';
3: SETP.EQ P0, R1, R2;	**3: SETP.EQ P0, R1, R2;**	**3: SETP.EQ P0, R1, R2;**	**3: SETP.EQ P0, R1, R2;**	**3: SETP.EQ P0, R1, R2;**
4:	4: SETP.NE PE, R1, R1';	4: @!PE SETP.NE PE, R1, R1';	4: SETP.NE PE, R1, R1';	4: @!PE SETP.NE PE, R1, R1';
5:	5: @PE MOV RE, 0x1;	5:	5: @PE MOV RE, 0x1;	5:
6:	6: SETP.NE PE, R2, R2';	6: @!PE SETP.NE PE, R2, R2';	6: SETP.NE PE, R2, R2';	6: @!PE SETP.NE PE, R2, R2';
7:	7: @PE MOV RE, 0x1;	7:	7: @PE MOV RE, 0x1;	7:
8: GLD R2, [R1];	**8: GLD R2, [R1];**	**8: GLD R2, [R1];**	**8: GLD R2, [R1];**	**8: GLD R2, [R1];**
9:	9: GLD R2', [R1'];	9: GLD R2', [R1'];	9: MOV R2', R2;	9: MOV R2', R2;
10:	10: SETP.NE PE, R1, R1';	10: @!PE SETP.NE PE, R1, R1';	10: SETP.NE PE, R1, R1';	10: @!PE SETP.NE PE, R1, R1';
11:	11: @PE MOV RE, 0x1;	11:	11: @PE MOV RE, 0x1;	11:
12: GST [R2], R3;	**12: GST [R2], R3;**	**12: GST [R2], R3;**	**12: GST [R2], R3;**	**12: GST [R2], R3;**
13:	13: SETP.NE PE, R2, R2';	13: @!PE SETP.NE PE, R2, R2';	13: ISETP.NE PE, R2, R2';	13: @!PE SETP.NE PE, R2, R2';
14:	14: @PE MOV RE, 0x1;	14:	14: @PE MOV RE, 0x1;	14:
15:	15: SETP.NE PE, R3, R3';	15: @!PE SETP.NE PE, R3, R3';	15: SETP.NE PE, R3, R3';	15: @!PE SETP.NE PE, R3, R3';
16:	16: @PE MOV RE, 0x1;	16:	16: @PE MOV RE, 0x1;	16:

Fig. 2. Implementation groups for the software-based hardening techniques.

IV. Implementation

Four case study algorithms have been chosen: matrix multiplication, Fast Fourier Transform (FFT), vector sum, and bitonic sort. All case studies are simple applications but differ in their use of the GPGPU control and data paths. In terms of data- and control-flow characteristics, matrix multiplication and vector sum are mostly data-flow oriented, with few conditional deviations, while FFT and bitonic sort are mostly control-flow oriented, with many conditional deviations.

The case-study algorithms were implemented in CUDA and compiled with the NVIDIA NVCC compiler. The compilation process generates the CUDA binary (cubin) file that contains the assembly code that is effectively executed by the NVIDIA GPU, as well as by the FlexGrip. This assembly code is called Source and Assembly (SASS) and can be extracted from the cubin file through the cuobjdump tool provided by NVIDIA's CUDA toolkit. In order to automatically apply the software-implemented techniques to the case-study applications, we used a tool called HPCT [15], which was upgraded to support the FlexGrip ISA as well as the proposed techniques. We input the SASS code to HPCT, which then automatically applies the code transformations and generates a hardened SASS file.

The performance and memory footprint costs of the proposed techniques are presented in Tables I and II, respectively. Results were organized in two categories (Datapath Duplication and Consistency Checking) that aim to individually analyze the costs involved in transformations performed by the proposed techniques, which are (1) datapath duplication, (2) consistency checking, and (3) host notification, as previously discussed in Section III. It is important to notice that we did not include a table for data memory footprint, as the datapath duplication duplicates all used data, and the consistency check requires one single predicate register.

Transformation 1 can be seen on the Datapath Duplication column, separated into No Optimization and *Move Optimization*. As one can see in Table I, results indicate significant performance improvements when *Move Optimization* is considered, where the average cost of runtime drops from 72.3% to 50.4%. The performance improvement observed in *Move Optimization* varies according to the case-study application due to the amount of memory access performed by each application. In terms of memory footprint, Table II shows that the *Move Optimization* does not reduce program memory usage when compared to non-optimized implementation, as both GLD and MOV are 64-bit instructions.

Transformations 2 and 3 can be seen on the Consistency Checking column. It shows both No Optimization and *Conditional Optimization* for the transformations to check memory accesses (Memory) and predicate setting (Predicate) instructions. To verify the individual costs of these transformations, we implemented them separately. When compared to the non-optimized techniques, the results show that the execution time overheads are almost halved when *Conditional Optimization* is used, where the average overhead dropped from 24.5% to 13% for memory access hardening and dropped from 16.4% to 8.4% for predicate setting hardening. These performance optimizations mainly happen due to the reduced quantity of host notifications, which also impacts on memory footprint overhead, where the cost is also halved when *Conditional Optimization* is used. The vector sum does not use PR and therefore has no data in the Predicate columns.

It is important to notice that, in order to implement a software-based fault tolerance technique, all three transformations are necessary, and the final cost is approximately their sum. We also performed experiments that proved that the final execution time and program memory footprint of the selected hardening technique is the sum of the chosen datapath duplication with the chosen checking transformations.

TABLE I
EXECUTION TIME (ns)

Application	Original Application [#]	Datapath Duplication [%]		Consistency Checking [%]			
				No Optimization		Conditional Optimization	
		No Optimization	Move Optimization	Memory	Predicate	Memory	Predicate
Matrix Mult.	320,340	77.1	49.9	21.1	10.7	10.7	5.6
FFT	963,730	78.3	66.7	16.5	11.5	8.3	5.8
VectorSum	140,640	75.9	46.8	28.0	-	16.7	-
Bitonic Sort	823,900	57.8	38.1	32.4	27.0	16.3	13.6
Average	562,153	72.3	50.4	24.5	16.4	13.0	8.4

TABLE II
PROGRAM MEMORY FOOTPRINT ($bytes$)

Application	Original Application [#]	Datapath Duplication [%]		Consistency Checking [%]			
				No Optimization		Conditional Optimization	
		No Optimization	Move Optimization	Memory	Predicate	Memory	Predicate
Matrix Mult.	264	78.8	78.8	16.9	6.8	8.5	3.4
FFT	1344	82.7	82.7	9.1	6.5	4.6	3.3
VectorSum	80	80.0	80.0	44.4	-	22.2	-
Bitonic Sort	288	66.7	66.7	20.0	16.7	10.0	8.3
Average	494	77.0	77.0	22.6	7.5	11.3	3.7

V. FAULT INJECTION RESULTS

Fault injection campaigns were automatically performed through simulation at RTL level in the ModelSim simulator in two steps, targeting only used registers. In the first step, the faults were injected during the execution of the original case-study algorithms to verify the susceptibility of the GPGPU to SEUs. In the second step, the faults were injected during the execution of the hardened applications to verify their effectiveness in detecting SEUs. For each fault injection campaign, 10,000 simulations were performed, and only one fault was injected in each simulation.

In order to evaluate the reliability of the system, we classified the injected faults according to their effect on the system: *Masked*, when the result is correct, *Detected Unrecoverable Error* (DUE), when the application crashes or the system hangs, or *Silent Data Corruption* (SDC), when the program finishes correctly, but the result is incorrect.

In order to analyze the impact in fault reduction of each proposed technique, we individually hardened memory access and predicate setting instructions. Thus, we divided the analysis into three categories to show the impact in error reduction according to the hardened instructions which are: (i) Memory access, (ii) Predicate setting, and both (iii) Memory + Predicate. Table **??** shows the fault injection results for the three categories without optimization (No Opt.) and with the *Move Optimization* (Move Opt.). The absolute number of errors for the original case-study applications is shown in the column Original Application.

Results for the category (i) show an average reduction in SDCs of 97% and 91.5%, respectively, for the non-optimized and for the *Move Optimization* techniques. It was already expected that the *Move Optimization* would decrease the error detection rate due to the point of failure inserted in the program by this optimization. Such SDC reduction occurs because data-flow corruptions are more prone to propagate to the memory, which was protected by this approach. Few reductions in DUE errors are observed because most of them occur when control-flow registers are affected, and these registers are no taken into account in this approach. On the other hand, the control-flow can be indirectly protected by hardening memory access instructions when variables that are used by control-flow instructions are loaded from memory.

Results for category (ii) show the opposite behavior observed in (i), presenting a high reduction in DUE errors and a small reduction in SDC errors. Such results are due to the fact that the predicate setting instructions are mainly used for the program's control-flow, thus, when the operands of these instructions are corrupted by a fault, the application is prone to stick in loops or perform incorrect deviations that cause thread desynchronization and lead the program to a hang state. Thus, when hardening these instructions, results show a reduction in DUE errors of up to 100% for both non-optimized and *Move Optimization* implementations. As the vector sum application does not use any PR and has not presented any DUE in its original form, we did not consider categories (ii) and (iii) nor DUE reduction.

TABLE III
FAULT INJECTION RESULTS - FAULTS AND PERCENTAGE REDUCTION

Application	Fault Classification	Original Application [#]	Memory [%]		Predicate [%]		Memory + Predicate [%]	
			No Opt.	Move Opt.	No Opt.	Move Opt.	No Opt.	Move Opt.
Matrix Mult.	SDC	3,522	99.8	95.5	0.8	0.3	100	96.2
	DUE	1,785	0.1	4.2	100	100	100	100
FFT	SDC	1,379	88.3	88.4	29.9	36.5	100	100
	DUE	2,323	32.7	32.0	100	100	100	100
VectorSum	SDC	3,096	100	82.3	-	-	-	-
	DUE	0	-	-	-	-	-	-
Sort	SDC	674	100	100	91.1	85.8	100	100
	DUE	1,171	70.5	72.6	0	0	74.3	73.0

The highest reduction rates have been achieved in the category (iii), where memory access and predicate setting instructions were hardened, thus, accumulating the advantages of data- and control-flow protection. The non-optimized hardening was capable of reducing SDCs and DUEs to zero in all applications but the Sort. Results for the *Move Optimization* show that not all errors could be mitigated, but the least SDC reduction observed was as large as 96.2% for the matrix multiplication.

VI. CONCLUSIONS AND FUTURE WORK

We presented an evaluation of software-based fault tolerance techniques designed to harden the GPRF of a GPGPU based on the G80 architecture against transient faults and proposed two optimizations to improve execution time and memory footprint. A fault injection campaign was performed through simulation to evaluate the GPGPU's susceptibility to SEUs.

The experimental results demonstrated that the hardening of memory access instructions achieved an average reduction in SDC errors of 97%, while the hardening of predicate setting instructions achieved a reduction in DUE errors of up to 100%. When both memory access and predicate setting instructions were hardened, the errors were reduced to zero. Such results indicate the high efficiency of software-based fault tolerance techniques to detect faults in the GPGPU register file, although the average cost to achieve these results has been 107.1% of program memory footprint and 109.8% of execution time. When we implemented the *Move Optimization* and the *Conditional Optimization* to mitigate these costs, the average overheads of program memory footprint and execution time dropped to 92% and 75.2%, respectively, at the average cost of 1% of SDC increase.

The results showed that the proposed improvements could be tailored for different scenarios, helping engineers in navigating the design space of hardened GPGPU applications. In the future, we intend to developed hardware-based fault tolerance techniques and compare the results with the techniques implemented in this work.

REFERENCES

[1] C. Slayman, "Soft errors—past history and recent discoveries," in *IEEE Int. Integrated Reliability Workshop Final Report*, 2010, pp. 25–30.

[2] A. Dixit and A. Wood, "The impact of new technology on soft error rates," in *International Reliability Physics Symposium*, 2011, pp. 1–7.

[3] P. Rech, C. Aguiar, C. Frost, and L. Carro, "An efficient and experimentally tuned software-based hardening strategy for matrix multiplication on gpus," *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2797–2804, 2013.

[4] J. R. Azambuja, G. Nazar, P. Rech, L. Carro, F. L. Kastensmidt, T. Fairbanks, and H. Quinn, "Evaluating neutron induced see in sram-based fpga protected by hardware- and software-based fault tolerant techniques," *IEEE Transactions on Nuclear Science*, vol. 60, no. 6, pp. 4243–4250, Dec 2013.

[5] P. E. Dodd and L. W. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Transactions on nuclear Science*, vol. 50, no. 3, pp. 583–602, 2003.

[6] M. Gonçalves, M. Saquetti, F. Kastensmidt, and J. R. Azambuja, "A low-level software-based fault tolerance approach to detect seus in gpus' register files," *Microelectronics Reliability*, vol. 76, pp. 665–669, 2017.

[7] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for gpu error detection," in *International Conference for High Performance Computing, Networking, Storage, and Analysis*, 2018, pp. 1–12.

[8] E. L. Rhod, C. A. L. Lisbôa, L. Carro, M. Sonza Reorda, and M. Violante, "Hardware and software transparency in the protection of programs against seus and sets," *Journal of Electronic Testing*, vol. 24, no. 1-3, pp. 45–56, 2008.

[9] K.-H. Huang and J. A. Abraham, "Algorithm-based fault tolerance for matrix operations," *IEEE transactions on computers*, vol. 100, no. 6, pp. 518–528, 1984.

[10] K. S. Yim, C. Pham, M. Saleheen, Z. Kalbarczyk, and R. Iyer, "Hauberk: Lightweight silent data corruption error detector for gpgpu," in *2011 IEEE International Parallel Distributed Processing Symposium*, May 2011, pp. 287–300.

[11] L. L. Pilla, P. Rech, F. Silvestri, C. Frost, P. O. A. Navaux, M. Sonza Reorda, and L. Carro, "Software-based hardening strategies for neutron sensitive fft algorithms on gpus," *IEEE Transactions on Nuclear Science*, vol. 61, no. 4, pp. 1874–1880, Aug 2014.

[12] B. Du, J. E. R. Condia, and M. Sonza Reorda, "An extended model to support detailed gpgpu reliability analysis," in *2019 14th International Conference on Design Technology of Integrated Systems In Nanoscale Era (DTIS)*, April 2019, pp. 1–6.

[13] K. Andryc, M. Merchant, and R. Tessier, "Flexgrip: A soft gpgpu for fpgas," in *2013 International Conference on Field-Programmable Technology (FPT)*, Dec 2013, pp. 230–237.

[14] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, no. 2, pp. 39–55, March 2008.

[15] J. R. Azambuja, A. Lapolli, L. Rosa, and F. L. Kastensmidt, "Detecting sees in microprocessors through a non-intrusive hybrid technique," *IEEE Transactions on Nuclear Science*, vol. 58, no. 3, pp. 993–1000, 2011.