HAL
open science

An Effective Method to Identify Microarchitectural Vulnerabilities in GPUs

Josie Rodriguez Condia, Paolo Rech, Fernando Fernandes dos Santos, Luigi Carro, Matteo Sonza Reorda

To cite this version:

Josie Rodriguez Condia, Paolo Rech, Fernando Fernandes dos Santos, Luigi Carro, Matteo Sonza Reorda. An Effective Method to Identify Microarchitectural Vulnerabilities in GPUs. IEEE Transactions on Device and Materials Reliability, 2022, pp.1-14. 10.1109/TDMR.2022.3166260 . hal-03669439

HAL Id: hal-03669439

https://inria.hal.science/hal-03669439v1

Submitted on 16 May 2022

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

An Effective Method to Identify Microarchitectural Vulnerabilities in GPUs

Josie E. Rodriguez Condia, *Member, IEEE,* Paolo Rech, *Senior Member, IEEE*
Fernando Fernandes dos Santos, *Member, IEEE* Luigi Carro, *Member, IEEE*
and Matteo Sonza Reorda, *Fellow, IEEE*

***Abstract*—Graphics Processing Units (GPUs) are increasingly adopted in several domains where reliability is fundamental, such as self-driving cars and autonomous systems. Unfortunately, GPU devices have been shown to have a high error rate, while the constraints imposed by real-time safety-critical applications make traditional (and costly) replication-based hardening solutions inadequate.**

This work proposes an effective methodology to identify the architectural vulnerable sites in GPUs modules, i.e. the locations that, if corrupted, most affect the correct instructions execution. We first identify, through an innovative method based on Register-Transfer Level (RTL) fault injection experiments, the architectural vulnerabilities of a GPU model. Then, we mitigate the fault impact via selective hardening applied to the flip-flops that have been identified as critical. We evaluate three hardening strategies: Triple Modular Redundancy (TMR), Triple Modular Redundancy against SETs (ΔTMR), and Dual Interlocked Storage Cells (Dice flip-flops). The results gathered on a publicly available GPU Model (FlexGripPlus) considering functional units, pipeline registers, and warp scheduler controller show that our method can tolerate from 85% to 99% of faults in the pipeline registers, from 50% to 100% of faults in the functional units and up to 10% of faults in the warp scheduler, with a reduced hardware overhead (in the range of 58 % to 94% when compared with traditional TMR).

Finally, we adapt the methodology to perform a complementary evaluation targeting permanent faults and identify critical sites prone to propagate fault effects across the GPU. We found that a considerable percentage (65% to 98%) of flip-flops that are critical for transient faults are also critical for permanent faults.

***Index Terms*—Graphics Processing Unit (GPU), Reliability, Selective Hardening**

I. Introduction

Josie E. Rodriguez Condia and Matteo Sonza Reorda are with the Department of Department of Control and Computer Engineering (DAUIN), Politecnico di Torino, Turin, Italy, e-mail: josie.rodriguez@polito.it, matteo.sonzareorda@polito.it.

Paolo Rech is with the Department of Industrial Engineering, University of Trento, Trento, Italy, e-mail: paolo.rech@unitn.it.

Fernando Fernandes dos Santos is with the Institut National de Recherche en Sciences et Technologies du Numérique (INRIA), Grenoble, France, e-mail: fernando.fernandes-dos-santos@inria.fr.

Luigi Carro is with the Department of applied informatics, Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, e-mail: carro@inf.ufrgs.br.

This work has been partially supported by the European Commission through the Horizon 2020 RESCUE-ETN project under grant 722325 and PERIOD under the grant agreement No 886202.

Manuscript received month XX, 20XX; revised month XX, 20XX.

THE computational power and the programming flexibility of modern Graphics Processing Units (GPUs) have boosted their adoption in the embedded and High Performance Computing (HPC) domains, especially when Artificial Intelligence (AI) algorithms based on Deep and Convolutional Neural Networks (DNNs and CNNs) are exploited. Fields that benefit from GPUs efficiency in executing AI algorithms, such as autonomous systems for automotive, robotics, and space exploration, are also characterized by strict requirements in terms of dependability. The recent market shift, from consumer to safety-critical applications, has gradually raised the interest in GPU's reliability, asking for effective solutions for fault mitigation.

An intense research effort is currently being carried out to characterize and mitigate the effects of *transient* faults in GPUs. On the one hand, GPU manufacturers employ several solutions to improve their devices' reliability, for example, by designing more robust memory cells [1], introducing error detection and correction structures in their memories (ECCs) [2], [3], replicating complete units [4] or components to increase the manufacturing yield (i.e., about 75% of the connecting nodes are duplicated or multiplied [5]), or developing (software) hardening solutions targeting the application domains where dependability and functional safety are crucial [6], [7]. On the other hand, the research community has been extensively studying GPU reliability through simulation-based fault injection [8]–[10] or beam experiments [11], [12]. The effort towards improving GPUs reliability also requires the availability of effective methods and tools to identify the most critical modules in the hardware and to estimate the improved reliability or safety, as described in widely adopted reliability standards, such as the ISO26262 [7].

Unlike transient faults, *permanent* faults in GPUs received a low attention so far in the literature, mainly due to the lack of appropriate GPU models and to their complexity. Unfortunately, permanent faults can become a significant issue due to aging, as GPUs life expectation shifts from 1-2 years of consumer electronics to 10s of years of automotive applications. As part of our contribution we will also show that the proposed method allows identifying the most critical effects of permanent faults in the modules of a GPU as well.

The identification of the primary sources of failures in the architecture of GPUs (*critical sites*) provides essential information to design and develop efficient solutions, such as

*Copyright © 2022 IEEE. Personal use of this material is permitted. However, permission to use this material for any other purposes must be obtained by sending a request to pubs-permissions@ieee.org

selective hardening. Reliability design teams can also exploit this information to adjust designs in early stages. Unfortunately, the current methods based on extensive fault injection campaigns at the Register-Transfer Level (RTL) and using complex algorithms in a GPU are impractical and unaffordable since they take too long (i.e., the characterization of *one* GPU module executing a simple five-layers CNN takes more than 740 *hours*) [13].

Several previous works focused on analyzing the impact of faults on conventional CPUs with limited scalability into GPUs due to their high hardware complexity and the intrinsic software parallelism. This paper extends our preliminary work [14], in which we introduced a method to identify critical sites in a GPU and evaluated the effectiveness of selectively protecting them via TMR. In this paper, we generalize and extend the method to identify architectural vulnerability sites (critical flip-flops) in any module of a GPU affected by *transient* faults. Moreover, the same method is used to identify those critical sites prone to propagate *permanent* fault effects (i.e., caused by aging). The proposed method exploits the combination of a set of fine-grain RTL fault injection campaigns and a set of compact but carefully selected input patterns *(micro-benchmark)* designed to stimulate the modules in the GPU. These micro-benchmarks are composed of a specific set of GPU ISA instructions activating a single target module in the GPU. A complementary set of applications provides different coding styles when exciting units in the GPU. This stimulus strategy, adopted from previous works [15], [16], aims at characterizing the fault effects on a given unit by analyzing the operation of compacted sets of assembly instructions in combination with some specific complex codes.

In order to assess the effectiveness of the proposed approach we resorted to the FlexGripPlus GPU model. We identify five modules that are more likely to impact the code execution when affected by faults: the Integer cores (INT), the Floating-Point core (FP32), the Special Function Unit core (SFU), the Pipeline Registers (PRs), and the Warp Scheduler controller (WS). For each module, we first identified the most critical sites using a set of dedicated micro-benchmarks. Then, we analyze the effect at the application level and determined the associated internal structures per critical site (i.e., registers, state machines, etc). Finally, we apply and evaluate three selective hardening strategies for the previously identified highly sensitive FFs to transient faults. The hardening strategies are based on: *i)* Triplicating the FFs (TMR), *ii)* Triplicating FFs against Single Event Transients (ΔTMR) [17], and *iii)* using Dual Interlocked Storage Cell FFs (Dice_FFs), further details in Section V. We choose three correction strategies rather than simple detection, since current GPU products must be compliant with real-time constraints (i.e., the processing of 40 frames per second). Thus, in the event of a fault, the system often does not have time to re-process the data before the next frame.

Results show that with the proposed approach we can *tolerate* from 85% to 99% of faults in the pipeline registers and from 50% to 100% of faults in the execution units with an overhead *lower than the one of traditional full TMR*.

The main contributions of this work can be summarized as:

- A viable method to analyze the microarchitectural fault effects produced by transient or permanent faults arising in different GPU modules, allowing the identification of the most vulnerable *(critical)* sites (FFs) in a GPU by employing a set of micro-benchmarks and application codes;
- The quantitative assessment of the proposed method and the resulting selective approach on a GPU model considering five units and three selective hardening strategies applied to FFs vulnerable to transient faults.

The remainder of the manuscript is organized as follows. Section II provides some background information and overviews the related works about methodologies of reliability evaluation and hardening mechanisms for parallel devices. Section III describes the proposed methodology to characterize the modules and identify vulnerable sites to transient faults. Section IV reports the analysis of the transient fault effects. Section V illustrates and reports the evaluation of the proposed selective hardening strategies for transient fault effects. Section VI provides an analysis of the vulnerable sites to propagate permanent fault effects in the GPU. Moreover, this section compares the vulnerable sites to permanent and transient faults to support the development of design improvements in the device. Finally, Section VII draws some conclusions and proposes future works.

II. Background and Related Work

The reliability evaluation of GPUs follows two main targets: *i)* the evaluation of applications, and *ii)* the evaluation of vulnerable sites in the hardware modules. In both cases, the task is challenging due to the massive parallelism of GPUs, which makes it difficult to produce accurate results in acceptable times. In the literature, several methodologies have been proposed for reliability assessment in GPUs. Most methods are based on massive fault injection campaigns using representative workloads with later analyzes of the outputs.

A. Methodologies for Reliability Evaluation

Prior works [9], [10] are based on high-level reliability evaluation using several fault injections campaigns modifying the source code of a program (i.e., mutations). Unfortunately, this coarse-grained level of injection can only represent at application level few effects coming from hardware faults. Moreover, there are several hardware modules that cannot be analyzed at these levels (e.g., schedulers and controllers). Authors in [18] proposed a systematic methodology to reduce the number of fault sites in software-based fault injection campaigns by analysing the number of executed pseudo-instructions in a program. This method only evaluates a representative subset of threads, so reducing the execution time. Unfortunately, the coarse-grain analyzes do not allow to correlate errors caused by hardware modules in a GPUs.

Other works [8], [19] evaluated the reliability of GPUs by performing extensive fault campaigns at two abstraction levels (*application* and *architectural*). The application analysis provides some details about the vulnerability of a GPU without

direct correlation with the fault effect caused in internal modules. In contrast, the low-level micro-architectural evaluation can provide detailed information about the identification of vulnerable locations on most internal modules. In [20], [21], and [22], the authors proposed some reliability evaluation methodologies based on microarchitectural evaluation on RT-level descriptions. In these works, fine-grain evaluation mainly uses two approaches to identify vulnerable locations: *i)* exhaustive evaluation of all sites in the modules, and *ii)* statistical samples of the sites in the modules. In the first case, the exhaustive evaluation requires high computational power and extensive simulation times, which are reduced by using derating factors based on functional usage. On the other hand, the statistical samples are good enough to provide representative reliability evaluations.

Other reliability evaluation methodologies are based on exposing real GPUs to beam experiments [12], [23], [24], which are accurate when targeting application characterizations and consist on executing the target workloads on real devices in the presence of external sources able to produce faults in the GPU (i.e., radiation). The analyzes are performed on the output of the applications, which can be used to identify sensitive and vulnerable sites in general parts of a GPU. Unfortunately, these techniques are not accurate when identifying specific vulnerable submodules and sites on the internal modules. Moreover, beam experiments are not easily available in most of the cases.

Finally, alternative methodologies exploit the combination of several abstraction levels in the reliability evaluation, so taking advantage of the main benefits from all levels. Authors in [25] introduced a methodology based on a framework to inject faults at the assembly level. Then, real GPU hardware boost the fault injection phase, so reducing latency. The tool employs an architectural model to profile applications, which are then evaluated in the real GPU. Thus, the framework provides a tunable and efficient balance between the representativeness and the cost of a fault-injection campaign. In [26], the authors proposed a methodology to evaluate the reliability of CNNs running on GPUs by combining low-level microarchitectural and software-based fault injection to determine microarchitectural faults in functional units and propagate the error effects at the instruction levels. This method reduces the profiling and fault injection campaigns by selecting a set of applications on a few operative data ranges, which are representative enough of the operation of functional units and some control modules in a GPU. In this work, we adopt a similar philosophy to perform the reliability characterization of the internal units in a GPU. In our approach, we employ a low-level microarchitectural abstraction to identify vulnerable structures inside a unit of the GPU and we exploit a set of special functional programs to excite and propagate any possible fault effect. Then, we employ those results to analyze and implement selective hardening mechanism. It is worth noting that for the purpose of this work, we did not use any evaluation using software-based fault injection strategies.

B. Mitigation Strategies

Several techniques have been proposed to mitigate the effects of transient faults in computing devices. The available solutions can be grouped in hardware, software, or hybrid techniques. The hardware solutions are devoted mainly to applications with heavy requirements in terms of functional safety and reliability. In this scenario, the additional costs are justified by the improved features and capabilities. The most classical hardware strategies include *Double and Triple Modular Redundancy* (DMR, TMR), and *Error-Correcting Codes* (ECCs). Other alternatives are selective hardening [27] and custom-optimized [28] techniques. For GPUs, the adoption of these solutions requires a careful evaluation of the introduced area and power consumption overhead. Their higher complexity and the huge amount of computing units make GPU hardening more challenging than for CPUs. Some GPU mitigation solutions based on Built-In Self-Repair (BISR), exploiting spare modules to replace faulty units, have also been proposed [29]–[31]. Furthermore, some authors proposed the reconfiguration of computational modules [32], [33] and memories [34] in GPUs once a fault is detected. Lately, GPU redundant mixed-precision hardware has been exploited for low-cost error detection [35].

Software mitigation solutions can also be adopted in GPUs. These solutions are based on code adaptations to mitigate fault effects using functional [36] and algorithmic [37], [38] methods. Nevertheless, the performance degradation and memory overhead can be relevant, compromising the real-time constraints of some GPU applications. In [39], the authors introduced a software-based redundant multithreading mechanism multiplying the threads to be executed. However, the performance overhead depends on the workload and the method cannot always be generalized.

This work aims at identifying the most critical elements in a GPU, considering mainly transient faults. Our methodology allows a fast yet effective flow for determining microarchitectural vulnerable sites in the modules of a GPU. Then, this information is used for fault mitigation purposes by implementing selective hardening strategies in any module used for parallel execution, including the functional units, the pipeline registers and scheduling controllers. To quantitatively validate the proposed method, we resorted to the FlexGripPlus RTL model [40] of an NVIDIA GPU architecture.

III. Proposed methodology

In this section we describe the proposed idea, giving a general overview and then describing in detail each step. We focus first on transient faults evaluation and the identification of vulnerable sites. Then, in Section VI, we extend and apply the proposed evaluation methodology for the evaluation of permanent faults.

A. Overview

The proposed methodology to identify the most sensitive locations to transient faults in GPU modules and to harden them is divided in four steps: **(1)** Design and selection of input patterns, **(2)** Microarchitectural Fault injection campaigns, **(3)**

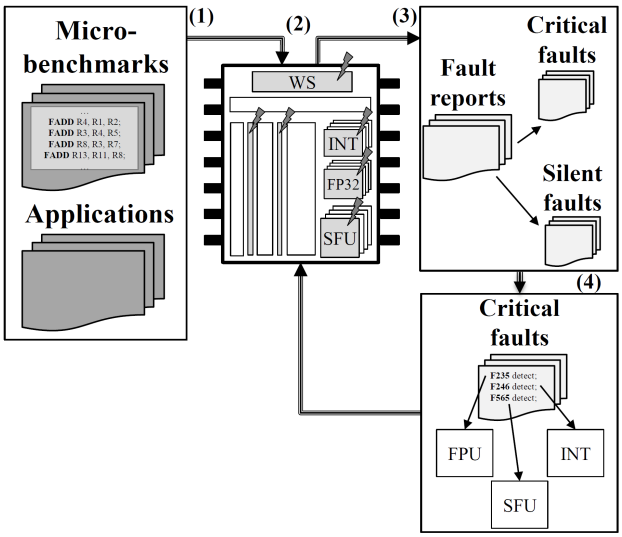

Fig. 1. A general scheme of the proposed methodology to evaluate the microarchitectural reliability, determine the fault impacts and identify vulnerable locations in the modules of a GPU core.

Fault classification, and **(4)** Feedback evaluation, as depicted in Figure 1.

In the first step, we design a set of special programs (micro-benchmarks) to stimulate each target module to the most commonly used set of assembly instructions and consider the functional modules they use for their execution (details in Section III-B). Moreover, we consider several common applications as complementary information input to observe the fault distribution dependence on the software and identify the critical sites. Then, in the second step, several **Low-level microarchitectural RTL fault injection** campaigns are performed on a microarchitectural GPU model. In case of the transient faults injections, each fault campaign starts by selecting one site of a main computing elements of the GPU to inject a single bit-flip. Once the program starts its execution, the fault is injected in a randomly selected time and then, the program continues the operation with the purpose of propagating any fault effect.

For the purpose of this paper, we consider five modules: the Warp Scheduler controller, the Pipeline Registers (PRs), the FP32 and INT functional units, and the Special Function Units (SFUs), and its local controller. It is worth noting that we do not inject faults in memory resources (caches and register file), since we assume that GPUs employed in safety-critical applications include ECC mechanisms. Moreover, faults in memory cells may affect any software visible state directly, propagating corrupted values with no further operations (its behavior is already well known (single/double bit-flip) and depends just on the memory technology [41]). On the other hand, understanding the impact of transient faults inside a computing module during the in-field operation of a device, on the application output is much harder [42].

The third step (**fault classification**) computes the probability for each fault to reach any visible state in the instruction outputs (i.e., to compute the Architectural Vulnerability Factor, AVF [43]). Moreover, the classification determines the type of impact on the instruction output values (critical or masked).

The **feedback evaluation** is intended to extract the most critical and common errors observed. Then, these errors are tracked-back to the original micro-structural source, so locating the specific hardware unit and site causing a failure. These locations are the main candidates to implement any protection mechanism, such as hardening. The identified critical locations can be then targeted for protection using any feasible mitigation mechanism according to the structure of the unit, type of resource in the GPU, and vulnerability.

To better describe the method we will refer to the RT-level GPU model (FlexGripPlus) [40], which will also be used to evaluate and validate the proposed approach. FlexGripPlus is an open-source VHDL-based model, which implements the Nvidia G80 architecture [44], with structural details for the most representative modules, and compatible with the commercial CUDA programming environment.

The next sub-sections explain in details the procedures used in the four steps of the proposed methodology. Although the FlexGripPlus GPU model was employed, the proposed methodology can be adopted for any other GPU architecture as well.

B. Design and Selection of Input Patterns

In the proposed method, a set of parallel programs (*kernels*) stimulates and excites the functional units and propagates the incidence of any fault effect to any observable point, so characterising the fault effects on the target GPU modules. The observable points consist of the register file and the output memory used during the execution of a kernel.

We defined the inputs patterns to characterize the microarchitectural (RTL) effects considering the incidence of a real GPU and several workloads with the purpose of profiling and identifying the most frequently employed instructions from the ISA of the GPU. These workloads are taken from universally adopted benchmark suites for HPC and safety-critical applications (e.g., Rodinia [45], NVIDIA SDK, CNNs [46], [47]).

The identified assembly instructions represent only a small part of all the ≈200 different opcodes in a typical ISA of a GPU, but they account for more than 70% of the instructions that compose the analyzed applications. These instructions composing each micro-benchmark are:

- Floating point operations (FADD, FMUL, FFMA - Fused MUL and ADD)
- Integer operations (IADD, IMUL, IMAD - MUL and ADD)
- Trascendental functions (SIN, EXP)
- Load/Store (GLD, GST)
- Branch (BRA)
- Integer set predicate/register (ISET).

The programs using **arithmetic instructions** (*IADD*, *IMAD*, *IMUL*, *FADD*, *FMAD*, *FMUL*, *FSIN* and *FEXP*) accept different operands from memory, so executing a sequence of instructions with different input values. On the one hand, these programs execute 64 threads (2 warps), each executing the same instruction, when targeting functional units in the GPU, since the evaluation of the available execution units in

a GPU does not depend on the degree of parallelism of the kernel but in the quality of the inputs. On the other hand, the programs are configured with 1024 threads (32 warps) when evaluating the warp scheduler controller to reach the maximum occupancy and workload in the module. In this case, the degree of parallelism in the programs directly affects the occupancy and the fault effects in the module. For both targets (functional units and scheduler), each program executes the same instruction without interactions between threads.

An ideal evaluation of each instruction requires the execution of the same or equivalent values used during the operation of the device. Other approaches make use of random patterns. Unfortunately, both cases require extensive simulation times, which is clearly unfeasible in most scenarios (i.e., more than 2^{64} possible operands for a 32 bits operands adder).

In case of the evaluation of transient faults, we limit the analysis to three input ranges in the functional units (*Small*, *Medium*, *Large*) identified from the preliminary profiling. This predefined input ranges are: *Small* (S, both inputs in the range $6.8\text{x}10^{-6}$ to $7.3\text{x}10^{-6}$), *Medium* (M, in the range 1.8 to 59.4), and *Large* (L, in the range $3.8\text{x}10^{9}$ to $12.5\text{x}10^{9}$). For the instructions using the SFU (*FSIN* and *FEXP*), we selected three inputs according to their operational constraints (in the range 0 to $\pi/2$), so bypassing the operation of any internal range reduction procedure. To avoid the bias in the experimental results, each micro-benchmark used in the fault injection campaigns considers 4 different randomly selected values for each input range.

We also consider **memory movements** (*GLD* and *GST*) and **control-flow instructions** (*BRA*, *ISET*). In the first case, these programs perform four sequences of one load operation followed by a store operation. A fault can be detected when either the load or the store operation fail. On the other hand, for the control-flow operation, we allocate a limited number of setting instructions (two) before one conditional branch operation followed by an unconditional branch operation. A fault is detected when a set register is not correctly assigned or when the intended branch condition fails. We anticipate that (not surprisingly) in most cases faults affecting control-flow instructions severely affect the execution of the GPU leading to a hang.

A complementary set of applications are selected to excite the modules in the GPU with typical workloads, which include different parallel coding styles.

C. RTL Fault Injection

A custom RT-level fault injection framework [48] was developed, using one general controller to manage the *ModelSim* environment hosting the model. This controller injects one transient fault (Single Even Upset) at a time in the target GPU module, according to a previously generated fault list, so it is possible to observe the fine-grain effect of a fault corrupting several threads as the effect of one single fault propagation and not by multiple fault injections.

For the proposed microarchitectural analysis in the GPU, we inject faults in the WS, PRs, INT cores, the Single Precision FP32s, the SFUs and their control logic (SFU controller) (see Figure 2). Table I describes the main features of the targeted modules, including their size, and instructions used to stimulate each module.

TABLE I
EVALUATED MODULES, SIZES AND INSTRUCTIONS USED PER MODULE

Module	RTL Size (Flip-Flops)	Type	Instructions
FP32	4,451	Execution/ Data	FADD, FMUL, FFMA
INT	1,542	Execution/ Data	IADD, IMUL, IMAD
SFU	3,133	Execution/ Data	FSIN, FEXP
SFU controller	288	Control	FSIN, FEXP
PRs	3,007	Control/Data	ALL
Warp Scheduler	942	Control	ALL

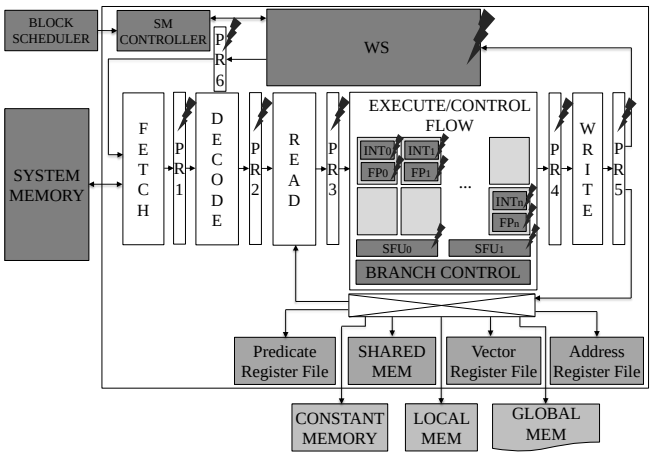

Fig. 2. A general scheme of the microarchitectural organization of the FlexGripPlus model and the internal modules targeted for the experiments. The SFU controllers are not shown in the scheme.

In the fault campaigns, the framework performs an exhaustive injection of all fault sites on a given module. Moreover, the framework can be used to target specific sites in a module as well. It is worth noting that for the evaluation of transient faults on each injection site, several injection times are randomly selected considering the total execution time of each micro-benchmark or application.

D. Fault classification

Once the fault is propagated to any of the available outputs of the target module (instruction output register, memories, or control signals), its effect is classified by comparing the output values and signals with the golden ones obtained in a fault-free simulation, as critical (i.e., corrupting the results of the program) or as *Masked* (i.e., no effect). The critical faults are classified as *Silent Data Corruption* (SDC) when producing output values mismatches, and as *Detected Unrecoverable Errors* (DUE) when hanging the device.

The classification stage generates one report after finishing each fault campaign, which includes the effect (*SDC*, *DUE*,

Masked) of each injected fault based on *(i)* the used instruction, *(ii)* the input value range *(iii)* the target module (where the fault is injected). The general report classifies the fault effect of each fault and allows to measure the AVF and the Fault Coverage (FC) for each module and instruction as the ratio between the number of observed fault effects (SDCs/DUEs) and the total number of the injected faults and injection sites, respectively.

E. Feedback Evaluation Analysis

The filtered reports (containing the critical faults, only) are used to carefully track all propagated faults to any of the observation points in the GPU and then we proceed to determine the original site of the fault causing the observed SDC or DUE. For this purpose, we combine the filtered reports and the output results of each micro-benchmark and application. We classify and group the locations, causing the fault propagation, and structures per module. These identified sensitive locations per module (e.g., FFs) are the main candidates to implement a protection or mitigation mechanism. Moreover, this information can be used to update or modify vulnerable structures in the units of a GPU.

F. Main benefits and possible drawbacks of the method

The main advantage of the proposed methodology lies on exploiting the functional operation of the units in the GPU to reduce the computational effort involved during the fault simulation campaigns for low-level microarchitectural hardware modules in a GPU. In detail, an ideal microarchitectural evaluation require the exhaustive evaluation of all possible input values for a module or equivalent statistical fault injection campaigns. Both cases (statistical and exhaustive) imposes enormous simulation times. Moreover, it is possible that a considerable amount of input patterns would be unrealistic for the operation of the complete system (i.e., lack of available instructions to generate such input patterns to a hardware unit). In contrast, the proposed methodology considers the functional operation of the modules (i.e., *serial* or *parallel*), the available instructions to activate each module, and the operational ranges to select the best combination between the description of the micro-benchmarks and the inputs used during the fault injection campaigns.

The evaluation and characterization correlate the vulnerable sites in a module and the output effects. This information can be used to mitigate fault effects or modify vulnerable sites in the modules of the GPU. Furthermore, the characterization and correlation can support the development of high abstraction level fault models for the reliability evaluation in GPUs.

However, the development of effective micro-benchmarks imposes several constraints, including the need to describe the routines at low-level assembly (which is not always feasible for GPUs). In addition, it is required to skip optimization and the effect on changing instructions by the compiler's operation. Finally, a deep understanding of the architectural operation of the GPU's internal modules is required to select the most feasible set of inputs patterns for the micro-benchmarks, so still providing representative figures in the characterization of vulnerable sites in the modules of a GPU.

IV. Analyzing the effect of transient faults

In this Section we detail the results of the RTL characterization of fault effect for transient faults in the targeted GPU modules. Then, we identify the most critical locations inside the modules. The fault injection campaigns for transient faults were performed on a server of 12 Intel Xeon CPUs running at 2.5 GHz and with 256 GB of RAM.

In the experiments, 5 GPU modules (PRs, FP32/INT/SFU functional units, and the warp scheduler controller) are evaluated and characterized using the fault injection described in Section III-C. Each fault injection campaign considers one micro-benchmark using one of the 12 selected assembly instructions and, for each micro-benchmark, we characterize four random values for each of the three input ranges (S/M/L). A similar procedure, although using only the standard workload, is followed for the fault-injection campaigns in the six applications (*Redu*, *Edge*, *FFT*, *MxM*, *nn*, and *Vadd*). Further details on the applications we used can be found in [49]. In total, 180 fault-injection campaigns evaluate more than 10,000 faults in the five targeted modules of a GPU for the characterization of transient faults. Overall we present data from more than 1.89×10^6 fault injections. This guarantees a statistical margin error lower than 3%. It is worth noting that micro-benchmarks are configured to execute more threads (1,024) when performing the fault-injections campaigns on the warp scheduler, so reaching the maximum occupancy of the module during the operation of each program.

To have a fine grain evaluation, we divide the injection locations in each module into two groups. For the functional units, we consider the injections in flip-flops close to the module's input and output registers (IO) as the first evaluation group. The second group is composed of the internal sites (IS). Similarly, for the PRs, we divide faults injected in the data path (Data) and control path (Control). Finally, for the warp scheduler controller, we divide the faults injected on the internal memories (M) and those on the internal logic blocks (L).

Figure 3 depicts the Architectural Vulnerability Factor (AVF) [43] distributions for the Functional Units (INT, FP32, and SFU), the PRs and the WS Controller. A preliminary evaluation on the data-movement and control-flow micro-benchmarks (*GLD*, *GST*, *BRA*, and *ISET*) reveals that the fault effect in functional units is negligible, since these modules remain idle. Thus, we have not performed the complete fault-injection campaigns when targeting those units with the previous micro-benchmarks. A similar approach was applied to the fault-injection campaigns in the applications, so we consider the existing instructions per program to characterize the modules activated by these instructions only.

A general overview of the results shows that, for all custom micro-benchmarks (*IADD*, *IMUL*, *IMAD*, *FADD*, *FMUL*, and *FMAD*), faults in the functional units are more likely to produce SDCs than DUEs. On the contrary, for most applications (*Redu*, *FFT*, *Edge*, *nn*, *MxM*, *Vadd*) injections are more likely to lead to DUEs. This different trend can be explained considering that, in the applications, the INT module is also used to calculate indices and addresses to memory resources,

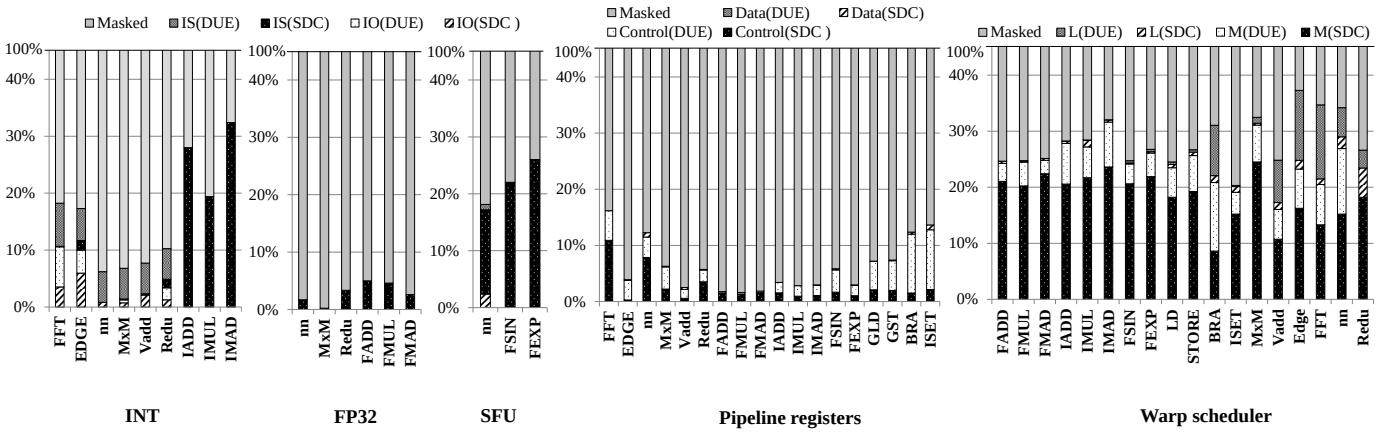

Fig. 3. AVF on the functional units (FP32, INT, SFU), PRs and WS for the different applications and micro-benchmarks.

so hanging the application when one index addresses incorrect memory locations. In contrast, the micro-benchmarks were designed to excite as many internal sites as possible in the functional units during the experiments. Thus, results shows that these micro-benchmarks can stimulate the functional units and report the effect of individual workloads. Interestingly, there is the negligible percentage of DUEs caused in the FP32 and SFU modules. In detail, a minimal percentage (not visible in Fig. 3) of faults in the SFU caused DUEs mainly generated by the impact of faults on a local controller. Once a fault corrupts the operation of the controller, it hangs the execution.

Analyzing the fault injection results, we found that most faults in the INT and FP32 affect only one of the instantiated threads. On the contrary, considering the nn, FSIN, and FEXP benchmarks, we found that faults in the special function units (SFUs) affect several threads. This behavior can be explained considering that the GPU cores use dedicated functional unit cores (INT and FP32) to operate each related instruction (*ADD*, *MUL*, and *MAD*). On the contrary, the GPU core only includes a few (two) SFUs which are shared among the different threads. Multiple SDCs are caused by faults in the control units of the SFUs. The effect of a fault occurring when managing a thread is propagated also to other threads.

As observed in Figure 3, each functional unit has an AVF that depends on the executed instruction (or application). This is because each assembly instruction, involving functional procedures, uses a certain number of submodules in a given structure (e.g., the IADD instruction uses a parallel adder structure instead of a multiplier in the INT module). Several submodules may then remain inactive during the execution of a specific instruction and be activated for others.

The case of the INT module is particularly interesting, since it has a low SDCs AVF for the micro-benchmarks but a high DUEs AVF for applications (>60% of the observed faults). This should not surprise, as the applications mostly employ integer instructions for control flow or parallelism management, such as the assignation of thread pointers and memory addressing. A (data) fault in these operations is likely to collapse the GPU operation.

The functional units AVF for the floating point instructions (*FADD*, *FMUL*, *FMAD*) is much smaller than for the integer instructions (*IADD*, *IMUL*, *IMAD*). This is caused by the higher complexity and area of the floating point units, that are more than 3x larger than the integer units (see Table I). A larger area increases the number of injection sites, thus reducing the probability to hit a critical resource for computation. However, the distribution shows that the fault effect is mainly dominated by SDCs in the internal modules of both functional units. Similarly, the applications using INT and FP32 modules (nn and MxM) present the same behavior.

Data in Figure 3 attests that the PRs are more likely to be corrupted by the execution of control-flow (*BRA*, *ISET*) or data movement (*GLD*, *GST*) operations. Applications that include these kind of operations (*Redu*, *FFT*, *nn*, *MxM*) are also the ones with higher AVF for the PRs. Interestingly, all instructions (arithmetical, control-flow, and memory movement) use uniformly about 47% of the registers in the PRs. It is worth noting that the stress of this cross-structural unit (PRs) in the GPU does not directly depend on the instruction, and cannot be the main cause for the particularly high AVF for control-flow and data movement instructions. The higher AVF is then only related to the type of instruction and the effect of the fault propagation on the computation.

In detail, most SDCs and DUEs caused by faults in the PRs are originated by control path corruption (about an average of 97.6% of the observed faults). Most of the observed DUEs are caused by corruptions of pipeline control registers that, despite being few (about 16%), are highly critical. Additionally, we found that faults in these pipeline control registers can corrupt multiple threads (up to 18 threads per warp). On the average, ≈5% of SDCs caused by faults in the pipeline affect multiple threads. While most PRs (≈84%) store operands for each parallel core and might produce single thread corruptions, registers devoted to control signals are critical as they manage the operation of several threads and cause multiple threads corruptions. Moreover, the evaluation of the memory movement and control flow micro-benchmarks (*GLD*, *GST*, *BRA*, *ISET*) reveals that faults in the data-path of the PRs can also corrupt the control flow.

Interestingly, results on the WS for both (micro-benchmarks and applications) support and confirm the idea that controllers

are highly sensitive modules in the structure of a GPU. Results show that for all analyzed programs more than 25% of faults can corrupt or hang the operation of an application. A detailed observation, reveals that most faults are product of failures in the internal memories of the scheduler (M) devoted to control and trace the execution of the operative threads. Nevertheless, a considerable percentage of "Logic" faults (those in the logic unit (L) of the WS) are highly sensitive for the applications and micro-benchmarks mostly employing control-flow operations. In fact, these (control-flow) applications are highly sensitive to DUEs caused by faults on the memory and the logic of the WS. For other applications and micro-benchmarks, faults in the WS are likely to cause more SDCs than DUEs mainly located in the memory of the WS.

A set of preliminary results (not shown) employing the original micro-benchmarks configured with a limited number of threads (64) revealed that a few percentage of faults (<5%) affected the WS by the limited percentage of occupancy of the module. On the other hand, results in Figure 3 (For maximum occupancy in the WS) demonstrate that the parallel configuration and the percentage of occupancy play an important role in the fault sensitivity of control modules in a GPU.

V. Protecting the GPUs modules against transient faults via selective hardening

In this Section, we explore and evaluate several selective hardening strategies to reduce the effect of transient faults in the identified critical sites for the evaluated modules of a GPU.

In detail, we evaluate three selective hardening strategies applied to those sensitive locations on each module of a GPU. The first strategy (Triple Modular Redundancy or TMR) triplicates only the most vulnerable sites *(critical sites)* in a given module and includes a voter circuit. The second strategy (ΔTMR) also triplicates the most sensitive sites, includes one voter circuit and also adds multiplexers and delay elements (Δ), which consist of a set of NOT gates organized sequentially. The ΔTMR strategy can be used to harden both transient fault effects (SEUs and Single Event Transients or SETs). In case of SETs, the delay elements and the multiplexers are used to capture data at different times [17]). In the experiments, we select delays elements providing up to 50% of the clock cycle of the GPU. Finally, the third strategy replaces the most vulnerable locations by using a Dual Interlocked Storage Cell (Dice). It is worth noting that the evaluation of the ΔTMR mechanism as selective hardening mechanism only considered the capabilities of protection against SEUs in the FFs.

Although the selective hardening strategies can be employed and applied at several levels of granularity, we work at the register level, so if a FF in a register (or sequential structure) is sensitive to faults, the complete register is targeted for the hardening. Finally, we evaluate and compare the hardware overhead costs and the effect in terms of fault tolerance in the GPU modules and the complete GPU core.

First, we trace back the sources of the faults, in the GPU core, that produced the observed errors. In case of the PRs, we found that most critical locations are part of the control-path registers (see Control (DUE) and Control (SDC) in Figure 3),

TABLE II
Modules and identified critical sub-modules. (*) The Warp scheduler size includes the internal memory cells

Module	Size of the module (FFs)	Submodule	Size of the Submodule (FFs)
PRs	3,007	Data path	1,536
		Predicate Flags	256
		current mask	192
		Memory addresses	384
		Address pointers	224
		Other	167
INT core	595	Input/Output registers	131
		Internal structures	395
FP32 core	4,451	Input/Output registers	0
		Internal structures	2,598
SFU core	3,133	Input/Output registers	32
		Internal structures	1,952
SFU controller	288	Internal structures	288
WS controller	4,502(*)	Internal memory controllers	262
		Warp status registers	108
		Memory source registers	15
		Memory cells	4,160(*)

which are employed to store and manage the active threads during the execution of a parallel program. The affected registers are devoted to store the predicates (*predicate flags*), active threads (*current mask*), memory parameters (*addresses* and *pointers*), among others. When corrupted, these locations affect several threads.

The analysis of the execution units revealed that, unfortunately, it is not possible to identify locations for the INT and FP32 modules that are more critical than others. Thus, the identification of submodules in these units is not as effective as with the PRs. In the execution units there are a few locations that cause the corruption of the output (such as the output registers of the module). However, there are several internal structures that once affected by faults propagate the fault effect.

For the WS, the results illustrate that different substructures in the WS were affected by faults. These submodules include state machine registers, registers handling the access to the internal memories, and registers addressing memory resources or storing information of the operatives threads. Interestingly, in most cases, once an element of the submodule is corrupted, several threads are affected. Moreover, almost all memory cells inside the internal memories play a crucial role in the operation of the WS, so these submodules are also listed as critical structures.

Table II reports only the identified critical locations in the modules and main candidates for the selective hardening. Other locations in the GPU modules, even when corrupted, do not affect computation, due to fault masking or missing activation patterns, so these were discarded as potential reliability targets in the present work. It is worth noting that deeper analyzes, using more kernels would be required to guarantee the injection of most activation patterns, so allowing the complete identification of insensitive submodules.

The implementation of the selective hardening targets all the identified submodules (see Table II) and their FFs. In detail, complete submodules (i.e., registers, counters, state machines, etc) are protected even when only internal parts were classified as sensitive to faults. It is worth noting that for the purpose of this work, those internal cells in the memories of the WS are not targeted for selective hardening, since efficient hardening

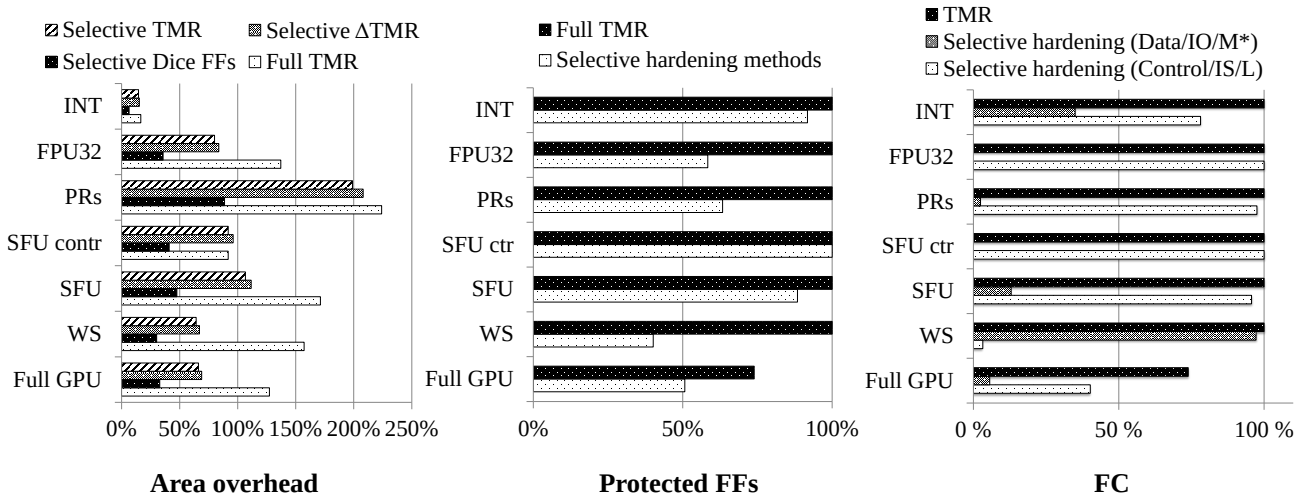

Fig. 4. Area overhead (*left*), percentage of protected FFs (*center*), and Fault Coverage (FC) (*Right*) of the selective hardening strategies and traditional TMR in the evaluated modules. The Functional units are divided as IS/IO and the PRs are divided as control path/data path. (*) Memory cells in the WS unit.

solutions for memory structures can be directly applied (i.e., SEC-DED ECC). Instead, we focus on the protection of the logic structures in the WS.

The hardware overhead cost (of selective hardening in the modules and the full GPU) is determined by performing hardware synthesis of the individual module and their hardened versions using the three strategies. Furthermore, we also synthesize the modules with a full TMR for comparison purposes. In the experiments, the synthesis framework employs the 15nm Open-Cell NAND-gate library [50] without any type of optimization.

Figure 4 shows the performance results of the three strategies of selective hardening and the traditional hardening in the GPU. The Figure shows the relative hardware overhead (Area) for the hardened modules. Moreover, we also show the percentage of protected FFs in all hardening approaches (selective TMR, ΔTMR, Dice FFs, and the full TMR) and the Fault Coverage (FC). It is worth noting that all other GPU modules are considered unprotected modules in the estimation of the cost and benefit of the strategies.

If we consider the full GPU, triplicating all the critical modules would increase the area of 127.5% (memories and other modules are not hardened), while each selective hardening approach would increase the area in 66.3%, 69.1% and 33.0% for the selective TMR, ΔTMR, and Dice FFs approaches, respectively. More in detail, the selective hardened strategies cover up to 50.7% of FFs in the GPU core. However, it must be noted that memory cells in the WS are not included as locations for hardening. In any case, for the evaluated approaches, the area overhead is almost 50% less of the overhead imposed by the traditional TMR applied into all modules.

In general, each selectively hardened module has an area overhead cost in the range of 6.5% to 208%. In case of the INT units, the overhead is minimal (between 6.5% and 15.3%). Interestingly, this module required the hardening of more than 85% of the FFs. Nevertheless, most of the area of INT is dedicated to combinational logic, so explaining the limited area cost when hardening the FFs in this unit. For other modules (i.e., FP32 and SFU), the selective hardening increase the area in the range of 35% to 111% and require to harden just between the 60% and 85% of the available flip-flops on each module, respectively.

On the other hand, the SFU controller requires a complete hardening with considerable overhead in area (about 90 to 96%) for the selective and full TMR strategies. The small size of the module and the high fault sensitivity of this module justify this choice. However, the Dice FF strategy can reduce in up to 50% the area overhead. In case of the PRs, the hardware cost follows a trend similar for the full TMR and the selective hardened version (near 200% overhead). A large part of the hardware cost is devoted to harden the control-path. Finally, the logic part of the WS module (controllers) increase the area overhead in 30% and 50% for the selective hardening strategies using the Dice FFs and both TMRs, respectively. as previously described, the internal memories in the WS are not included in the estimation of the area overhead.

As observed for all critical modules, the area overhead for the analyzed hardening strategies show consistent trends for all modules. While the Dice FFs strategy provides the lowest cost per module (as depicted in black for every module in the Figure), the two TMR hardening versions (selective TMR and ΔTMR) increase the area overhead in more than 50% with respect to the hardened Dice FFs versions. Curiously, both hardening strategies reduce the area of the functional units and the WS controller in up to 50% with respect to the complete TMR strategy. In contrast, the overhead in the PRs and the SFU controller units keep a similar cost (differences lower than 20%) when comparing the selective and full TMR hardened versions. In case of the PRs, a selective Dice FFs hardening strategy is more efficient in area than the other evaluated strategies.

The Fault Coverage results, see FC in Figure 4 (on the right), give an indication of the benefits of the selective hardening. The reported FC is the average coverage obtained with the applications and micro-benchmarks. The results are divided in

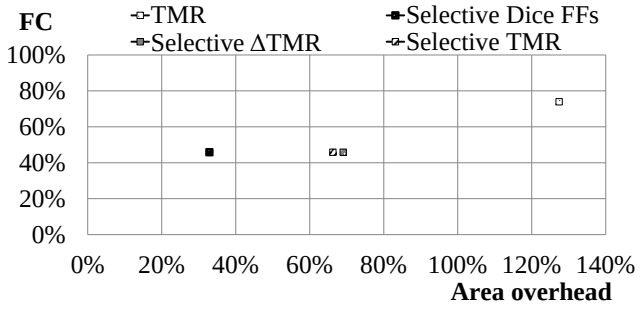

Fig. 5. A general overview between the total area overhead and fault coverage for the three hardening mechanisms evaluated in all modules (*FPU*, *SFU*, *INT*, *WS* and *PRs*) of the full GPU.

submodules for the evaluated functional units (IO/IS) and the PRs (control path/data path).

From results, the FC in the PRs increases when selectively protecting the control path (about 85% to 99%), see Figure 4(Right). In contrast, a low percentage (<15%) is observed when protecting the data path, only. In the execution units, the IS of each module plays an important role in the application of the selective hardening. According to results from Figure 4, the selective hardening of the internal structures can increase the reliability of an application in a range from 50% to 100%. In contrast, the protection of the input and output registers (IO) has a limited benefit: FC between 0%, 13%, and 60% in the FP32, SFU and INT modules, respectively. The WS is an special hardening case. According to results, the internal memories are critical and are the mainly sources of errors in the module. nevertheless, the logic in this controller is also the source of a considerable percentage of faults (about 8%). In fact, for several applications, these locations is responsible of most hanging effects in the applications. Thus, the hardening of those locations is crucial for the correct operation of the GPU. The cells in the memory are not targeted in this work, since ECC solutions can be applied to such structures.

As observed for the full GPU, the hardening of IS and control in the modules contribute to the FC in about 38%. Finally, combining the previous results in Figure 5, an area optimized hardened GPU reduces of up to 46% the number of faults of the entire design with less than 35% of area overhead for the selective Dice FFs strategy. It must be noted that the selective TMR and ΔTMR can also provide other benefits (i.e., minimal degradation in performance and SETs mitigation), but with additional (almost double area) costs in the hardened design.

VI. Analyzing critical sites to permanent fault effects

In this Section, we extended the reliability analysis in the FFs of the 5 targeted GPU modules (PRs, FP32, INT, SFU and WS) to observe the effect of permanent faults and their propagation across the system. This evaluation provides the identification of the critical sites that could be more susceptible to rising of permanent faults (i.e., caused by aging). These faults can arise during the in-field operation of a device and can collapse the complete execution of an application. Thus, this evaluation is intended to serve as a preliminary analysis and support the development of special countermeasures (e.g., hardening at layout levels) or design improvements in the units in order to protect, or remove, such set of critical FFs. At the end, a correlation between the identified critical sites to permanent and transient faults is determined with the purpose of supporting the development of complementary hardening mechanisms and protect critical structures against both fault types.

In the experiments, 50 fault-injection campaigns injected Stuck-at faults (0-1) for each internal site on each targeted module (see Table I). In total, the custom framework injected 31,952 permanent faults in all selected GPU modules. In detail, the framework performs one fault simulation by injecting one fault in a site of the GPU model and then executes one program (micro-benchmarks or codes). When a program hangs or finishes with mismatches in the output results, the framework classifies the fault as critical.

As inputs for the micro-benchmarks and applications, we employed the same set of patterns employed in the transient experiments to excite the internal location of each module.

Figure 6 reports the fault rate for permanent faults of the programs on each evaluated module. This fault rate is determined as the ratio between the number of detected errors in the application and the number of injected faults in a module. Thus, results show the impact of permanent faults in the modules for the analyzed applications.

At first glance, we realized that permanent faults affect a higher percentage of internal sites (FFs) than transient faults. In particular, the effect of permanent faults on the functional units produced a considerable percentage of SDCs (ranging from 20% to 70%, 5% to 55%, and 40% to 55% for INT, FP32, and SFU, respectively) caused by corrupted sites in both submodules and input and output registers of the units. As expected, most errors in the micro-benchmarks only caused SDCs since these programs only use the functional units for data-based operation. In contrast, a considerable percentage of DUEs (from 10 to 40%) are caused for those applications using the INT cores to resolve memory addresses.

In the PRs, most errors in the applications are located between the control and data-path structures. Interestingly, the results for micro-benchmarks show that permanent faults corrupting data-path registers are the primary source of errors. On the other hand, the control-path registers are the main sources of errors for more complex codes (Edge, FFT, and Vadd). Possible explanations for the previous behavior are based on the different parallel configurations of the programs (number of threads) and the coding styles (i.e., use of control-flow instructions).

The observed results about permanent fault effects support the idea that controllers inside a GPU are highly critical in the execution of an application. In fact, the high percentage of fault effects (from 30% to 86%) demonstrate that these units are particularly vulnerable to faults, especially the internal memories. A detailed analysis of the results in the WS reveals that the internal memory is the source of most errors (70-92% of effects per module). Nevertheless, the logic structures in the controller also produce up to 13% of the observed

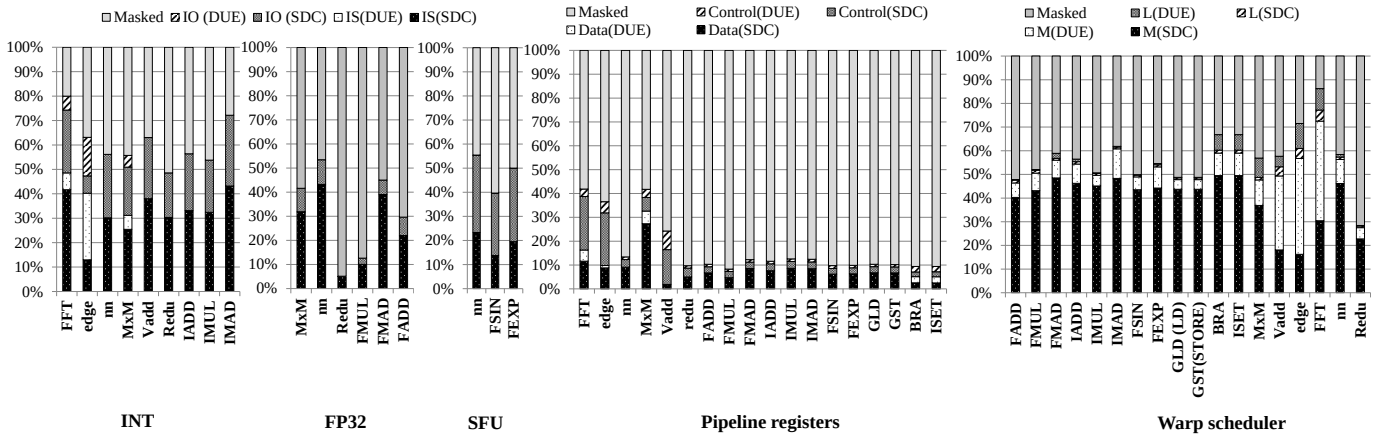

Fig. 6. Fault rate of permanent faults on the functional units (FP32, INT, SFU), PRs and WS for the different applications and micro-benchmarks. A fault corrupting the outputs is also classified as SDC, faults causing hanging or crashing effects are classified as DUEs.

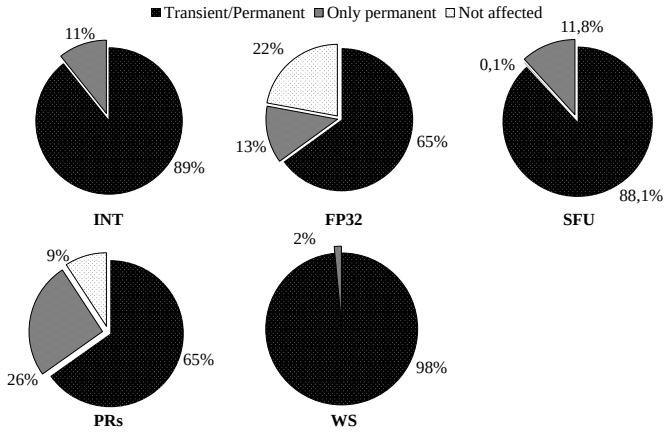

Fig. 7. Cumulative distribution of sensitive submodules (FFs) in the GPU units to transient and permanent faults for the evaluated applications.

effects. They are mostly produced in applications and micro-benchmark using control-flow instructions (i.e., BRA) while remaining minimal (<3%) for other programs. Interestingly, the same submodules (e.g., registers to address the internal memories, handling the parallel parameters or part of the state machines) are affected by faults for all applications.

Results in figures 3 and 6 show the fault impact in each module of the GPU for both fault types (transient and permanent). A general comparison between the results of the experiments shows that permanent faults are prone to corrupt the operation of applications up to 3 times more than transient faults. This behavior is because both fault types (transient and permanent) may corrupt an instruction and propagate the fault effect across the program. In particular, transient faults can hit an idle module (not activated by any instruction), then neglecting their effects. Moreover, instructions and programs corrupted by transient faults can also produce masking effects, so reducing or removing fault effects before an error reaches observation points. In contrast, a permanent fault (once generated) inside a module remains active during the program's execution, so each instruction that interacts with the module can also activate the fault and propagate the fault effect with lower chances of masking.

The profile for each module shows different fault impacts since that some GPU modules are more critical than others. In the case of the functional units (FP32, INT, and SFU), we observed that both fault types produce SDCs in the programs. Interestingly, a large percentage of sites in the input and output registers were more prone to be affected by permanent faults than transient ones. In contrast, the PRs show a moderate sensitivity to fault effects for both fault types. In fact, permanent faults affecting the PRs are mainly critical in data-path sites. On the other hand, transient faults are highly prone to corrupting more control-path registers and causing more errors to the outputs. Finally, for the WS, the memory cells are highly prone to corrupt the operation of the programs for both faults types (permanent and transient). In fact, in both cases, up to 95% of identified sites corrupting instructions belong to memory cells. Sites affected by faults in the controller's logic produce a minor percentage of fault effects (<15%), causing more DUEs than SDCs and hanging the operation of the system. In detail, registers addressing the memories, the status, and the state machines registers are the most sensitive for both cases.

In order to determine the effect and distribution of the critical faults in the different modules of the GPU, we evaluate and compare the effect of transient and permanent faults inside the different modules. For this purpose, we analyze the fault reports of all programs and determine the sensitive FFs in the GPU units that, once affected by faults, cause errors in the applications. Figure 7 shows the cumulative distribution of sensitive submodules to transient, permanent, or both fault types. It is worth noting that we consider as sensitive registers or FFs, all those sites with at least one element able to propagate either a transient or a permanent fault and cause errors in a program.

Results show that a high percentage of critical sites (from 78% to 99%) can corrupt applications once affected by faults. Moreover, all identified vulnerable FFs to transient faults (more than 65% of sites) are also prone to produce errors

when affected by a permanent fault. In detail, INT, FP32 SFU, and PRs modules contain a considerable percentage of critical sites (11%, 13%, 11.8%, and 26%, respectively) sensitive only to permanent faults. In contrast, the WS module shows that most critical sites (98%) are prone to corrupt instructions when affected by either transient or permanent faults. It is worth noting that the cumulative distribution for the WS also included the memory cells, which are prone to cause errors when affected by both fault types.

VII. Conclusions

In this work, we proposed a methodology to evaluate the reliability of different modules in a GPU. First, we analyzed the microarchitectural impact of transient faults on a selected set of modules in a GPU, and then we performed and evaluated three selective hardening on those identified critical sites in the modules. The evaluation is based first on selecting a custom set of micro-benchmarks and some applications exploiting the functional operation of the modules in the GPU to reduce the required computational effort with respect to traditional exhaustive techniques for microarchitectural hardware units.

The experimental results show that the considered selective hardening techniques are particularly effective when selecting specific sub-structures in some units in a GPU, such as the control-path registers in the pipeline registers (85% to 99%). Results also showed that some hardening solutions reduce by up to 65% and 58% the area overhead in the individual modules and the entire GPU core, respectively, with respect to a complete triple modular redundancy strategy.

A complementary evaluation of the impact of permanent faults showed that a high percentage (from 65% to 98%) of those identified critical sites to transient fault effects are also prone to errors by permanent faults. These results also indicate that designers might consider more sophisticated countermeasures, hardening mechanisms, or employ design improvements to mitigate also permanent fault effects on vulnerable sites in the modules of a GPU and extend its reliability.

As future activities, we plan to explore compacted hardening mechanisms facing multiple fault models affecting hardware modules in GPUs.

References

[1] P. Rech, L. Carro, N. Wang, T. Tsai, S. K. S. Hari, and S. W. Keckler, "Measuring the Radiation Reliability of SRAM Structures in GPUS Designed for HPC," in *IEEE 10th Workshop on Silicon Errors in Logic - System Effects (SELSE)*, 2014.

[2] F. Gruner, S. Keil, and J. S. Montrym, "Error detection and correction for external dram," Nov. 8 2016. US Patent 9,490,847.

[3] K. Lee, M. B. Sullivan, S. K. S. Hari, T. Tsai, S. W. Keckler, and M. Erez, "On the trend of resilience for gpu-dense systems," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks – Supplemental Volume (DSN-S)*, pp. 29–34, 2019.

[4] J. R. Nickolls, "Defect tolerant redundancy," Apr. 12 2005. US Patent 6,879,207.

[5] J. Y. Chen, "Gpu technology trends and future requirements," in *2009 IEEE International Electron Devices Meeting (IEDM)*, pp. 1–6, 2009.

[6] S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Making convolutions resilient via algorithm-based error detection techniques," 2020.

[7] NVIDIA, "NVIDIA Announces World's First Functionally Safe AI Self-Driving Platform." "https://nvidianews.nvidia.com/news/nvidia-announces-worlds-first-functionally-safe-ai-self-driving-platform", 2018.

[8] S. Tselonis and D. Gizopoulos, "Gufi: A framework for gpus reliability assessment," in *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 90–100, 2016.

[9] T. Tsai, S. K. S. Hari, M. Sullivan, O. Villa, and S. W. Keckler, "Nvbitfi: Dynamic fault injection for gpus," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 284–291, 2021.

[10] S. K. S. Hari, T. Tsai, M. Stephenson, S. W. Keckler, and J. Emer, "SASSIFI: An architecture-level fault injection tool for GPU application resilience evaluation," in *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 249–258, 2017.

[11] P. Rech, L. L. Pilla, P. O. A. Navaux, and L. Carro, "Impact of gpus parallelism management on safety-critical and hpc applications reliability," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 455–466, 2014.

[12] F. F. d. Santos, P. F. Pimenta, C. Lunardi, L. Draghetti, L. Carro, D. Kaeli, and P. Rech, "Analyzing and increasing the reliability of convolutional neural networks on gpus," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, 2019.

[13] J. E. R. Condia, F. F. dos Santos, M. Sonza Reorda, and P. Rech, "Combining architectural simulation and software fault injection for a fast and accurate cnns reliability evaluation on gpus," in *2021 IEEE VLSI Test Symposium (VTS)*, pp. 1–6, 2021.

[14] J. E. R. Condia, P. Rech, F. F. dos Santos, L. Carrot, and M. S. Reorda, "Protecting gpu's microarchitectural vulnerabilities via effective selective hardening," in *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pp. 1–7, 2021.

[15] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas, and Y. Makris, "Instruction-level impact analysis of low-level faults in a modern microprocessor controller," *IEEE Transactions on Computers*, vol. 60, no. 9, pp. 1260–1273, 2011.

[16] N. Karimi, M. Maniatakos, A. Jas, and Y. Makris, "On the correlation between controller faults and instruction-level errors in modern microprocessors," in *2008 IEEE International Test Conference*, pp. 1–10, 2008.

[17] V. Petrovic and M. Krstic, "Design flow for radhard tmr flip-flops," in *2015 IEEE 18th International Symposium on Design and Diagnostics of Electronic Circuits Systems*, pp. 203–208, 2015.

[18] L. Yang, B. Nie, A. Jog, and E. Smirni, "Practical resilience analysis of gpgpu applications in the presence of single- and multi-bit faults," *IEEE Transactions on Computers*, vol. 70, no. 1, pp. 30–44, 2021.

[19] F. G. Previlon, C. Kalra, D. R. Kaeli, and P. Rech, "A comprehensive evaluation of the effects of input data on the resilience of gpu applications," in *2019 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pp. 1–6, 2019.

[20] M. M. Goncalves, J. R. Azambuja, J. E. R. Condia, M. S. Reorda, and L. Sterpone, "Evaluating software-based hardening techniques for general-purpose registers on a gpgpu," in *2020 IEEE Latin-American Test Symposium (LATS)*, pp. 1–6, 2020.

[21] J. E. R. Condia, M. M. Goncalves, J. R. Azambuja, M. Sonza Reorda, and L. Sterpone, "Analyzing the sensitivity of gpu pipeline registers to single events upsets," in *2020 IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, pp. 380–385, 2020.

[22] 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. 76-77, pp. 665–669, 2017.

[23] F. F. d. Santos, S. K. S. Hari, P. M. Basso, L. Carro, and P. Rech, "Demystifying gpu reliability: Comparing and combining beam experiments, fault simulation, and profiling," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 289–298, 2021.

[24] F. Fernandes dos Santos, L. Draghetti, L. Weigel, L. Carro, P. Navaux, and P. Rech, "Evaluation and mitigation of soft-errors in neural network-based object detection in three gpu architectures," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, pp. 169–176, 2017.

[25] B. Fang, K. Pattabiraman, M. Ripeanu, and S. Gurumurthi, "A systematic methodology for evaluating the error resilience of gpgpu applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 12, pp. 3397–3411, 2016.

[26] F. F. d. Santos, J. E. R. Condia, L. Carro, M. S. Reorda, and P. Rech, "Revealing gpus vulnerabilities by combining register-transfer and software-

level fault injection," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pp. 292–304, 2021.

[27] I. Polian and J. P. Hayes, "Selective hardening: Toward cost-effective error tolerance," *IEEE Design Test of Computers*, vol. 28, no. 3, pp. 54–63, 2011.

[28] M. Gonçalves, J. R. Condia, M. S. Reorda, L. Sterpone, and J. Azambuja, "Improving gpu register file reliability with a comprehensive isa extension," *Microelectronics Reliability*, vol. 114, p. 113768, 2020. 31st European Symposium on Reliability of Electron Devices, Failure Physics and Analysis, ESREF 2020.

[29] J. E. Rodriguez Condia, P. Narducci, M. Sonza Reorda, and L. Sterpone, "A dynamic reconfiguration mechanism to increase the reliability of gpgpus," in *2020 IEEE 38th VLSI Test Symposium (VTS)*, pp. 1–6, 2020.

[30] T. Koal and H. T. Vierhaus, "Logic self repair based on regular building blocks," in *New Trends in Audio and Video / Signal Processing Algorithms, Architectures, Arrangements, and Applications SPA 2008*, pp. 109–114, 2008.

[31] J. E. R. Condia, P. Narducci, M. Sonza Reorda, and L. Sterpone, "Dyre: a dynamic reconfigurable solution to increase gpgpu's reliability," *The Journal of Supercomputing*, p. 1–18, 2021.

[32] W.-J. Lee, S.-H. Lee, J.-H. Nah, J.-W. Kim, Y. Shin, J. Lee, and S.-Y. Jung, "Sgrt: A scalable mobile gpu architecture based on ray tracing," in *ACM SIGGRAPH 2012 Talks*, SIGGRAPH '12, (New York, NY, USA), Association for Computing Machinery, 2012.

[33] K. Kwon, S. Son, J. Park, J. Park, S. Woo, S. Jung, and S. Ryu, "Mobile gpu shader processor based on non-blocking coarse grained reconfigurable arrays architecture," in *2013 International Conference on Field-Programmable Technology (FPT)*, pp. 198–205, 2013.

[34] J. Zhao, G. Sun, G. H. Loh, and Y. Xie, "Energy-efficient gpu design with reconfigurable in-package graphics memory," in *Proceedings of the 2012 ACM/IEEE international symposium on Low power electronics and design*, pp. 403–408, 2012.

[35] F. F. dos Santos, M. Brandalero, M. B. Sullivan, P. M. Basso, M. Hübner, L. Carro, and P. Rech, "Reduced precision dwc: An efficient hardening strategy for mixed-precision architectures," *IEEE Transactions on Computers*, vol. 71, no. 3, pp. 573–586, 2022.

[36] D. A. G. Gonçalves 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*, vol. 65, no. 3, pp. 791–804, 2016.

[37] J. Chen, S. Li, and Z. Chen, "Gpu-abft: Optimizing algorithm-based fault tolerance for heterogeneous systems with gpus," in *2016 IEEE International Conference on Networking, Architecture and Storage (NAS)*, pp. 1–2, 2016.

[38] D. Sabena, M. Sonza Reorda, L. Sterpone, P. Rech, and L. Carro, "On the evaluation of soft-errors detection techniques for gpgpus," in *2013 8th IEEE Design and Test Symposium*, pp. 1–6, 2013.

[39] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, "Real-world design and evaluation of compiler-managed gpu redundant multithreading," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pp. 73–84, 2014.

[40] J. E. R. Condia *et al.*, "Flexgripplus: An improved gpgpu model to support reliability analysis," *Microelectronics Reliability*, vol. 109, 2020.

[41] R. Baumann, "Soft errors in advanced computer systems," *IEEE Design Test of Computers*, vol. 22, pp. 258–266, May 2005.

[42] O. Subasi *et al.*, "Characterizing the impact of soft errors affecting floating-point alus using rtl-ievel fault injection," in *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, Association for Computing Machinery, 2018.

[43] 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," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 29, IEEE Computer Society, 2003.

[44] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "Nvidia tesla: A unified graphics and computing architecture," *IEEE Micro*, vol. 28, pp. 39–55, March 2008.

[45] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *2009 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 44–54, 2009.

[46] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[47] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv*, 2018.

[48] B. Du, J. E. R. Condia, M. Sonza Reorda, and L. Sterpone, "On the evaluation of seu effects in gpgpus," in *2019 IEEE Latin American Test Symposium (LATS)*, pp. 1–6, 2019.

[49] 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)*, pp. 1–6, 2019.

[50] M. Martins *et al.*, "Open cell library in 15nm freepdk technology," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design ISPD'15*, p. 171–178, Association for Computing Machinery, 2015.

Josie E. Rodriguez Condia (Member, IEEE) received the M.Sc. degree in electronics engineering from Universidad Pedagógica y Tecnológica de Colombia (UPTC), Colombia in 2017, and the Ph.D. degree in Computer Engineering from Politecnico di Torino, Turin, Italy in 2021. His research interests include functional test, parallel architectures, Graphics Processing Units, and embedded system design.

Paolo Rech (Senior member, IEEE) received the master's and PhD degrees from Padova University, Padova, Italy, in 2006, and 2009, respectively. He is an associate professor at the University of Trento, Italy, and at the Federal University of Rio Grande do Sul, Brazil. In 2021 he was a Marie Curie fellow with Politecnico di Torino, Italy. His main research interests include the reliability of radiation-induced effects in large-scale HPC, autonomous vehicles and space exploration.

Fernando Fernandes dos santos (Member, IEEE) received the B.Sc. degree in computer science from State University of Western Parana (UNIOESTE), Cascavel, Brazil in 2014, and the M.Sc. degree and Ph.D. degree in computer science from Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, in 2016 and 2021, respectively. His main research interests include fault tolerance in embedded and safety-critical applications.

Luigi Carro (Member, IEEE) received the B.Sc. degree in electrical engineering, and the M.Sc. and Ph.D. degrees from Universidade Federal do Rio Grande do Sul (UFRGS), Brazil, in 1985, 1989, and 1996, respectively. He is presently a full professor at the Informatics Institute of UFRGS, in charge of Computer Architecture and Organization.

Matteo Sonza Reorda (Fellow member, IEEE) received the M.Sc. degree in electronics in 1986 and the Ph.D. degree in computer engineering in 1990, respectively, both from Politecnico di Torino, Italy. Currently he is a full professor in the Department of Control and Computer Engineering of the same Institution. His research interests include test of SoCs and fault tolerant electronic system design.