POLITECNICO DI TORINO
Repository ISTITUZIONALE

Evaluating the Impact of Transition Delay Faults in GPUs

(Article begins on next page)

03 May 2024

Evaluating the Impact of Transition Delay Faults in GPUs

Josie E. Rodriguez Condia*, Matteo Sonza Reorda*
*Politecnico di Torino - Department of Control and Computer Engineering (DAUIN)
{josie.rodriguez, matteo.sonzareorda}@polito.it

Abstract—This work proposes a method to evaluate the effects of transition delay faults (TDFs) in GPUs. The method takes advantage of low-level (i.e., RT- and gate-level) descriptions of a GPU to evaluate the effects of transition delay faults in GPUs, thus paving the way to model them as errors at the instruction level, which can contribute to the resilience evaluations of large and complex applications. For this purpose, the paper describes a setup that efficiently simulates transition delay faults. The results allow us to compare their effects with stuck-at-faults (SAFs) and perform an error classification correlating these faults as instruction-level errors. We resort to an open-source model of a GPU (*FlexGripPlus*) and a set of workloads for the evaluation. The experimental results show that, according to the application code style, TDFs can compromise the operation of an application from 1.3 to 11.63 times less than SAFs. Moreover, for all the analyzed applications, a considerable percentage of sites of the Integer (5.4% to 51.7%), Floating-point (0.9% to 2.4%), and Special Function unit (17.0% to 35.6%) can become critical if affected by a SAF or TDF. Finally, a correlation between the fault's impact from both fault models and the instructions executed by the applications reveals that SAFs in the functional units are more prone (from 45.6% to 60.4%) to propagate errors at the software level for all units than TDFs (from 17.9% to 58.8%).

Index Terms—Graphics Processing Units (GPUs), Functional units, Instruction-level fault impact

I. INTRODUCTION

Nowadays, Graphics Processing Units (GPUs) are boosting the execution of data-intensive and complex algorithms in the safety-critical domain (e.g., autonomous systems, including self-driving cars) with extensive requirements in terms of reliability and dependability [1]–[3].

Both requirements (*reliability* and *dependability*) impose challenges in the safety-critical domain since several studies [4], [5] have demonstrated that modern GPUs, built with the latest transistor technology nodes, are more prone to faults and their effects may affect more severely the applications in comparison to devices from other technology generations. Thus, analyzing fault impacts is crucial, especially to identify the most critical faults. These analyzes can also contribute to the vulnerability analysis of internal modules, during early design stages, so supporting the development of design improvements and effective countermeasures. Finally, they allow devising suitable error/fault models working at higher abstraction levels, which may better support the reliability analysis of complex systems integrating GPUs and the evaluation of heavy workloads (e.g., Neural Networks), trading off accuracy

with the required computational effort. However, until recently, the characterization and fault's impact evaluation in GPUs have been mainly focused on phenomena led by external effects affecting the device (i.e., transient faults induced by radiation) and have barely extended to effects arising during the operational phase of a device (i.e., faults caused by aging or wear-out), such as stuck-at faults (SAFs) and transition delay faults (TDFs).

Commonly, the fault's impact evaluation and analysis for processors and accelerators, such as GPUs, is performed using three main strategies: *i)* real experiments, *ii)* software-based evaluation, and *iii)* architectural-based characterization. The first strategy (such as the exposure to external sources produced by radiation beams) is excellent in characterizing fault effects on specific workloads. In fact, beam experiments effectively evaluate the effects of transient faults in a device. Unfortunately, the effects of permanent fault models (such as SAF and TDFs) are more complex to evaluate using this strategy since they physically and permanently corrupt a device. Alternatives, such as hardware fault emulation [6] of a target device, can be employed to evaluate SAFs effects but can hardly evaluate the effects of TDFs due to the lack of timing accuracy in the emulation of a GPU. The second approach (software-based evaluation) is based on the mutation of instructions executed by a real device to represent fault effects, which allows modeling a larger set of faults but introduces some limitations in the accuracy of the results. Finally, architectural-based characterizations use representative circuit/system models (functional, structural, and low-level micro-architectural at RT and gate level) in their analyses. These architectural evaluations are closer to the real hardware operation and can provide a representative characterization of the impact of faults. However, long simulation times are required due to the complexity of the architectural model and the target abstraction level. Other characterization methods include multi-level strategies and application-based characterizations. In the first case, the combination of several abstraction levels (i.e., micro-architectural and real experiments) boosts the reliability evaluation on complex applications [7], [8], [3]. In the second case, the fault's impact evaluation is based on the application, and equivalent fault effects are injected with limited accuracy regarding the relation between hardware faults and their propagation as error effects [9].

Just recently, some authors [10] started to characterize the

effects of permanent faults on units inside a GPU, such as the schedulers and some functional units. Unfortunately, to the best of our knowledge, analyzing the impact of transition delay faults in GPU structures has never been performed so far. Thus, it was not possible until now to evaluate the effects of transition delay faults considering the structural features of the units in a GPU.

In this work, we propose a method to characterize and evaluate the impact of transition delay faults in functional units of GPUs. Moreover, we compare and evaluate the criticality effect of SAFs and TDFs in the functional units of a GPU for several workloads. Moreover, the proposed method provides a preliminary step towards high-level error models (e.g., as instructions errors) for the analysis of complex applications in GPUs. The proposed approach is based on combining the complete workload's execution in micro-architectural models and a multi-thread fault propagation evaluation of the targeted units. For the evaluation and validation of the method, we resort to an open-source GPU model (*FlexGripPlus*) [11], implementing a micro-architecture of NVIDIA, targeting three main functional units: the Integer (INT), Floating-point (FPU), and Special Function Unit (SFU).

Results concerning the impact of TDFs, on the execution of applications on a GPU are presented for the first time. The fault's impact comparison between TDFs and SAFs shows that impacts of SAFs are higher (from 45.6% to 60.4%) than those of TDFs (from 17.9% to 58.8%). Moreover, the results show that considerable percentages of sites of the INT (up to 51.7%), FPU (up to 2.4%), and SFU units (up to 35.6%) are prone to produce failures when affected by any type of permanent fault (SAFs or TDFs). Finally, we correlate both fault types (SAFs and TDFs) to instruction-level software errors. This analysis identifies the instruction types prone to excite and cause errors for a given workload. The reported results show that both fault types affect arithmetic, logic, and conversion-type instructions in different proportions.

The proposed method is intended to support the analysis in the functional safety domain and identify vulnerable and critical structures to SAFs and TDFs, which can guide designers to provide improvements or introduce adequate countermeasures, such as hardening specific points within the GPU. Moreover, the proposed method can support the development of accurate high-level error models, which can be simulated in a much faster way than fine-grain micro-architectural fault effects.

This work includes the following contributions:

- The analysis of the low-level micro-architectural effects of SAFs and TDFs in the INT, FPU, and SFU of GPUs for several workloads;
- The identification of structural sites in the functional units of a GPU, which are most prone to produce failures due to SAFs and TDFs;
- The identification of faults corresponding to software errors and the correlation of fault effects from SAFs and TDFs to instruction-type errors.

The paper is organized as follows. Section II provides a background about the organization of a GPU. Section III
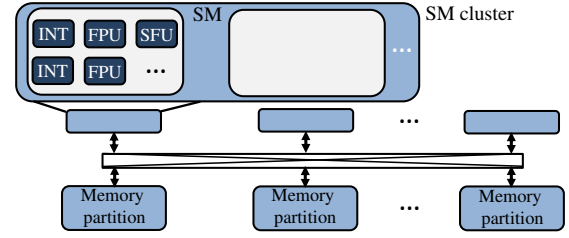

Fig. 1. A general scheme of the internal organization of a GPU.

describes the proposed method to evaluate the effects of faults in the GPU functional units. Section IV describes the experimental setup, and Section V provides the results of the fault effect evaluation on a set of typical workloads. Section V reports the analysis performed on the results. Finally, Section VI draws some conclusions and lists some future works.

II. ORGANIZATION OF A GPU

GPUs are special-purpose accelerators specially designed to provide high throughput on an application by exploiting multi-core parallelism. Thus, the organization of general-purpose GPUs comprise arrays of configurable parallel processors (*'SIMD engines'*, *'Streaming multiprocessors'* or SMs), see Figure 1. Internally, Each SM also exploits parallelism by implementing an architecture based on the Single-Instruction Multiple-Data (SIMD) paradigm or variations, such as the Single-Instruction Multiple-Thread (SIMT), and including several scalar functional units (e.g., *'Streaming Processors'* or SPs) and special-purpose accelerators (*'Special Function Units'*, or SFUs, and Tensor Processing Units, or TPUs). Modern GPU architectures are organized as hierarchical sets of SMs (i.e., *'SM clusters'*), which include two to four SMs, to provide significant control and enforce hardware parallelism.

In detail, each SM is organized in multiple pipeline stages to allow the fetching, decoding, and processing of instructions. Initially, one thread-group *(warp)* instruction is submitted for parallel execution on the available functional units. One or more instructions can be executed in parallel in the same SM by dividing the functional units according to scheduling policies in the controllers.

The functional units in the SMs (or CUDA cores) are highly regular and are the main operative workhorse modules in a GPU. Moreover, all thread operations performed with the functional units directly address any level of the memory hierarchy in the GPU since each operative thread in the SMs can address a complete memory hierarchy. This hierarchy includes a register file and shared, local, constant, and global memories. In case of hardware faults inside the functional units, their effects mainly correspond to errors in one of the memory locations (the register file and the main memory) after executing an instruction.

III. PROPOSED METHOD

This section describes the proposed method to evaluate the impact of SAFs and TDFs in functional units of GPUs.
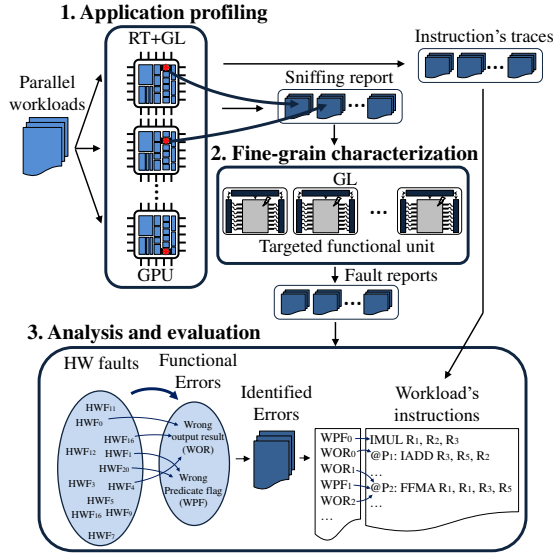
Fig. 2. A general scheme of the proposed method to evaluate the impact effect of TDFs and SAFs in functional units of GPUs.

The proposed method is based on three main steps (see Figure 2): *i)* Application profiling, *ii)* Fine-grain fault injection and propagation, and *iii)* Analysis and evaluation.

In the first step *(Application profiling)*, each application is profiled at the micro-architectural level (RT). The targeted functional unit for the evaluation is the only one represented at the gate level to increase the precision and accuracy in the execution of the application. This step produces a trace profile, storing the functional information of a targeted unit in the GPU. Then, this trace profile serves as input for the second step *(fine-grain fault injection and propagation)*. In this second step, a local and fine-grain fault simulation campaign evaluates the effect of faults and the propagation effects for the targeted unit. We exploit the fact that all functional units store the outputs in any of the resources of the memory hierarchy (i.e., *register file* or *global memory*) of the GPU. Thus, the analysis of the primary outputs of the functional units is equivalent to the analysis of the main observable points in the complete GPU (memories). In this step, the primary outputs are collected and stored. Finally, in the third step *(Analysis and evaluation)*, the collected outputs and the original trace profile are analyzed to evaluate the fault impacts and identify the error effects at the instruction levels.

The following subsections describe each step of the proposed method for fault effect evaluation in detail.

A. Application profiling

This step consists of the trace, profile, and storage of all changes on the primary inputs and outputs of a detailed description of the targeted unit (gate-level). Thus, it is possible to observe a fault-free operation of an application executed in a low-level description of the GPU (RT- and gate-level).

In detail, a dynamic profiling mechanism is employed to collect information from the primary input and outputs of a targeted unit in the GPU. In fact, the mechanism is an additional structure included in the original design, which exploits the multi-threading execution to speed up the simulation and profile several cores in parallel using the same running application. Moreover, this dynamic mechanism can be used to perform the profiling on individual cores. The generated traces (*Sniffing reports*) serve as a fault-free operation of the target units and supports the fine-grain fault injection campaigns and the analysis steps. The second set of reports (*Instruction's traces*) stores the traces of the executed instructions per application, which are employed to identify the instructions exciting the faults in the units and support the evaluation and analysis of impact fault effects.

B. Fine-grain fault injection and propagation

In this step, a focused fault injection campaign is performed on the gate-level version of a target functional unit only. Since we are interested in the fault's impact effects on the functional units, we only consider the effects of those faults propagated across the unit and capable of affecting any output during the execution of the application's instructions, which are equivalent to the propagation of errors into one of the memory resources in the GPU. For this purpose, a fault is detected when at least one mismatch is produced in the unit's outputs as an effect of an internal fault.

The fault-free profile feeds the functional units during the fault injection campaigns, so injecting the equivalent data from the application on each specific unit. In practice, all functional units are profiled and fault simulated to evaluate the fault impact for all threads in a parallel program. As in the first step, the performance's speed up is obtained when exploiting the multi-threading execution of fault injection campaigns. As a result of the fault injection campaigns, a set of output fault reports (*Fault reports*) are combined with the *instruction traces* to analyze and evaluate the impact of the fault effects.

The proposed strategy collects the information of each executed instruction. In case of a mismatch with the fault-free simulation, the report stores the fault effects from the unit's primary outputs and the detection time, which are later used in the correlation of the fault impacts of the units and the effect on the application's instructions. This procedure also identifies performance issues caused by TDF effects.

C. Analysis and evaluation

This step is internally divided in two sub-steps: *1)* the preliminary analysis, and *2)* the correlation analysis. In the first case, the preliminary analysis evaluates the impact of faults inside the unit and the propagation effects on the application. Moreover, this analysis identifies the vulnerable and critical sites inside each functional unit for SAFs, TDFs, or both. Finally, the analysis provides the binary-level fault effects on the operations' outputs. The second analysis (*correlation analysis*) identifies and maps the fault effects caused by TDFs and SAFs, as software errors impacting the applications. In this case, the analysis determines those faults causing errors in the instructions of an application. An additional analysis focuses on the software errors to classify the instruction types exciting

the faults (in the functional units) which impact the execution of the instructions inside the applications. The correlation analysis combines the collected reports (*sniffing* and *fault*) and relates the detected time (of a fault) and the execution of a given instruction, so identifying the source instruction activating and propagating the effect of a fault. This analysis is intended to support the development of accurate high-level error models by identifying error effects at the instruction levels for TDFs and SAFs.

In detail, the fault reports are analyzed, and the fault effects are classified following three main categories according to the effects on the outputs of a target unit: *i)* Output Value Corruption (OVC), *ii)* Hang, and *iii)* masked. Faults are classified as OVCs when the effect of a fault corrupts at least one output. Moreover, a fault is identified as Hang when it corrupts, stops, and collapses the operation of the targeted functional unit, so it cannot provide a stable output. Finally, the fault is classified as masked when the fault is not propagated or does not cause any effect on the functionality and the output value. At the instruction level, we classify the fault impacts as instruction errors using two categories: *i)* impact on results (as *Wrong output result error* or **WOR**), and *ii)* impact on predicate flags (as *Wrong Predicate Flag error* or **WPF**). A WPF is a fault corrupting the output predicate flags of the instruction, so incorrectly enabling or disabling predicate conditions, which serve as input conditions for other instructions in the parallel program. Both categories are selected considering the possible effects of a fault on the individual thread execution of an instruction and the feasible representation of an error by modifying the instruction. An WOR error represents a fault inside a unit impacting an instruction by affecting the output value with a wrong result.

It is worth noting that the proposed approach uses several fault simulators to handle the complexity of a large design, such as GPUs. An individual fault simulator can hardly be used as the main solution since most of them are optimized to analyze SAFs and TDFs in scan-based designs. Moreover, the computational power and latency increase according to the design's complexity and size when dealing with designs without those structures. Thus, the proposed method handles and reduces the analysis time for such large designs.

IV. EXPERIMENTAL SET-UP

A custom framework was developed based on a general controller that manages the operation of the three steps in the proposed method. The first step *(application profiling)* uses a logic simulator *(ModelSim by Siemens EDA)* handling a mixed description (at the RT and gate level) of the GPU. Moreover, the focused fault injection campaigns are performed by adapting the logic simulator *(Z01X by Synopsis)* with the targeted functional unit at the gate level. This logic simulator can be configured to inject either (*Slow-to-Rise* and *Slow-to-Fall*) TDFs or SAFs by a main controller of the framework. For the experiments, one fault is injected per fault simulation. Finally, custom tools analyze the reports and evaluate the effect of the fault as instruction error effects. Moreover, the complete

TABLE I
MAIN FEATURES OF THE SELECTED APPLICATIONS

	Duration (cc)	Size (ins.)	Data memory footprint (bytes)	Functional units		
				INT	FPU	SFU
Sobel	115,666	98	1,024	√	√	√
Euler	8,178	74	8,192	√	√	√
nn	44,909	25	8,192	√	√	√
Vector_add	33,583	12	16,384	√	√	
Reduction	9,134	69	8,612	√	√	
Transpose	19,896	53	12,288	√		
FFT	41,096	175	768	√		

framework supports multi-threading operation to speed up the simulation and analysis procedures, so several functional units (e.g., FPU_0, FPU_1, and FPU_2 inside an SM) are analyzed simultaneously during the execution of an application.

Since the functional units directly employ the memory hierarchy of the GPU (i.e., the register file), the propagation of any fault to the primary outputs also implies the propagation and visibility on any of the destination registers per thread. Thus, there is no missing observability on faults during the focused fault injection campaigns to characterize the impact of faults inside a unit.

V. EXPERIMENTAL RESULTS

The fault's impact evaluation targets three main functional units inside the SMs of a GPU: INT, FPU, and SFU. The FlexGripPlus model was configured with one SM, 32 INT cores, 32 FPUs cores, and 4 SFUs to force the execution of all tasks from a parallel program on the same functional units and analyze the fault impact considering all possible operands from an application. In fact, all functional units (INT, FPU, and SFUs) in an SM core are profiled, evaluated, and analyzed. The gate-level descriptions of the functional units for the fault injection experiments were obtained after synthesis using the 15nm open-cell technology library [12].

In the evaluation experiments, one fault (SAF or TDF) is injected into a target site before the execution of each fault simulation. A TDF is labeled as detected when there is at least one mismatch in the outputs or delay effects of at least one clock cycle. Seven typical workloads (*Reduction*, *Sobel*, *Euler*, Nearest Neighbor or *nn*, *Vector_Add*, *Transpose*, and *FFT*) are selected for the evaluation of the fault impact on the functional units. These applications excite different functional units and are selected among the CUDA SDK samples and Rodinia test suites [13]. Table I reports the main features of the selected applications, including the number of instructions, execution time, memory footprint, and functional units used.

The analyses consider each parallel application's workload per core (all executed operations and input operands per thread). More in detail, the performed experiments report information about $6.85x10^5$, $1.51x10^6$, and $9.91x10^6$ faults injected in the SFU, INT, and FPU units, respectively, for both fault models (SAFs and TDFs). The simulation experiments required from about 0.25 hours to 2.5 hours, per application, for a total of around 10 hours.
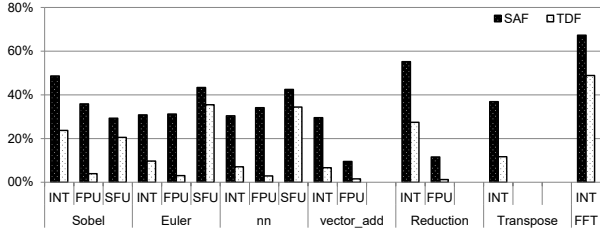
Fig. 3. Average percentage of SAFs or TDFs faults in the functional units producing a failure on the outputs for several workloads.
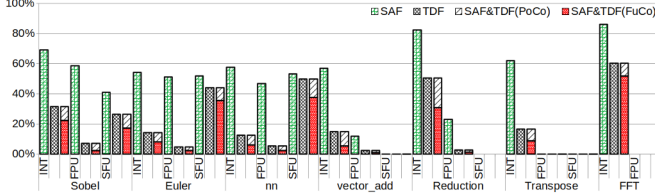

Fig. 4. Percentage of hardware sites in the functional units propagating SAF, TDF, or both effects.

A. Architectural analysis

Figure 3 depicts the average percentage of faults (SAFs and TDFs) affecting the outputs for each application (OVC and hang effects), which are obtained after analyzing the faults propagated on all functional units.

In general, SAF effects are more critical than TDFs for all programs and corrupt more frequently the operation of the functional units for the evaluated workloads from 1.23 (e.g., SFU for *euler*) to 11 times (e.g., FPU for *nn*).

In detail, an overview of the instruction's workloads shows that INT cores are used to calculate the addressing of memory resources and the management of parallel parameters (i.e., thread ID), which contributes to explaining the moderate criticality of SAFs (from 29% to 67%). Interestingly, TDFs inside the INT cores produce lower effects (from 6.6% to 48.9%) for all analyzed workloads.

The results show that SAFs affect the FPUs in a moderate percentage (from 9.53% to 35.9% of propagated effects), but TDFs in the FPUs barely propagate effects (from 1.3% to 3.9%). The limited number of 'FPU-type' instructions in the programs and the structure of the FPUs seem to be the causing factor for the observed impacts of SAFs and TDFs. Regarding the SFU unit, the experimental results show that a moderate-low percentage of SAFs (6.2% to 43.3%) and TDFs (1.9% to 35.5%) corrupt the operations on the evaluated workloads. In detail, the impact effects of SAFs and TDFs are proportional to the number of instructions per application using the SFU unit. In fact, *euler* and *nn* employ a high number of 'SFU-type' instructions (i.e., *RCP*, *EXP*, or *SIN*), which are more prone to fault effects than other applications (i.e., *sobel*) with a few number of SFU-type instructions.

A micro-architecture analysis of fault effects reveals that SAFs or TDFs impact the three functional units (INT, FPU, and SFU) similarly. In fact, a fault can corrupt from 1 to 35 primary outputs in any functional unit. In detail, the analysis shows that most propagated fault effects mainly corrupted one

(33.3% to 84.9% of SAFs, and from 26.5% to 87.6% of TDFs), or two (9.8% to 44.2% of SAFs, and from 7.7% to 35.7% of TDFs) primary outputs for all analyzed workloads.

A second analysis identifies the most vulnerable sites concerning SAFs and TDFs (see Figure 4). The first and second bars represent the percentage of sites that, when affected by a SAF or TDF, respectively, produce a failure. The third bar discriminates the percentage of 'Partially Corruptible' (PoCo) and 'Fully Corruptible' (FuCo) sites impacted by both fault types. A PoCo site is affected by at least one and up to 2 or 3 faults from both fault types. In contrast, FuCo sites propagate all (*four*) permanent faults (*2 SAFs and 2 TDFs*). Interestingly, the same percentage of sites affected by TDFs is also vulnerable to SAFs. Moreover, the percentage of FCs varies from around 32.1% to 85.8% of all identified sites for each functional unit. Thus, the unit's fault vulnerability depends on the application and its coding style, which affects the percentage of critical sites prone to SAFs, TDFs, or both.

B. Correlation between hardware faults and software errors

Based on the gathered results, we performed an analysis to correlate the propagated faults and the possible software errors for each application. In detail, we analyze the potential effect of all propagated faults (from each functional unit) as instruction-level software errors. The instruction-level errors are classified following the **WOR** and **WPF** categories, introduced in Section III, which can easily be used for instruction-level fault modeling in functional units.

Table II reports, for each functional unit, the percentage of SAFs and TDFs mapped as instruction-level errors. The identified errors only consider propagated faults causing either WORs (around 99% of identified errors) or WPFs (about 1% of identified errors). The missing percentage of faults corresponds to those producing other error effects, which can hardly be represented as effects during the instruction's execution (i.e., hanging). The results show that SAFs can be more easily associated with instruction-level errors than TDFs. In fact, a considerable percentage of faults are directly associated with instruction errors, from about 55.6% to 60.4%, 45.6% to 48.9%, and 52.9% to 62.6%, for the INT, FPU, and SFUs, respectively. In contrast, the association of TDFs as errors varies according to the functional unit. In detail, a considerable percentage of faults (from 42.2% to 58.8%) are associated with errors for the INT unit. Similarly, a moderate percentage of TDFs (from 39.3% to 47.8%) in the SFUs maps into instruction-level errors. However, just a few percentages

TABLE III
CORRELATION BETWEEN INSTRUCTION-LEVEL SOFTWARE ERROR, PRODUCED BY SAFs AND TDFs, AND THE INSTRUCTION TYPES PER APPLICATION.

Fault type Unit Ins-types	SAFs									TDFs								
	INT				FPU			SFU		INT				FPU			SFU	
	arith. (%)	Logic (%)	Conv. (%)	Config (%)	arith. (%)	Conv. (%)	Config (%)	arith. (%)	Config (%)	arith. (%)	Logic (%)	Conv. (%)	Others (%)	arith. (%)	Conv. (%)	Others (%)	arith. (%)	Others (%)
Sobel	67.4	13.2	16.7	2.7	79.9	8.2	11.8	98.5	1.5	65.8	12.3	14.4	7.5	86.0	7.7	0.0	99.5	0.5
Euler	71.4	16.9	7.3	4.4	96.8	2.7	0.6	99.1	0.9	60.2	24.5	0.0	15.3	95.7	0.0	0.0	99.3	0.7
nn	63.2	14.7	15.1	7.1	99.5	0.0	0.5	98.1	1.9	67.7	26.8	3.2	2.2	86.7	13.0	0.0	99.5	0.5
vector_add	77.2	9.1	8.9	4.7	98.1	0.0	1.9	-	-	76.1	13.2	8.5	2.2	100.0	0.0	0.0	-	-
Reduction	68.6	14.4	14.6	2.4	89.2	8.9	1.9	-	-	63.9	11.5	15.5	9.0	57.1	13.7	14.6	-	-
Transpose	61.8	14.8	19.8	3.6	-	-	-	-	-	14.3	3.0	69.3	13.4	-	-	-	-	-
FFT	72.3	17.1	8.8	1.8	-	-	-	-	-	82.5	11.5	5.5	0.4	-	-	-	-	-

of TDFs (from 17.9% to 25.9%) are propagated into errors. The structure of the units and their critical paths (in particular in the FPU) seem to be the main factor for the reduced percentage of TDFs associated with software errors.

Finally, we associated the identified software errors with the dynamic instructions in the applications to determine the main instruction types exciting faults and propagating the errors. Table III reports (for each application) the normalized percentage of software errors (*WORs* + *WPFs*) classified among the instruction-types using any functional unit. In detail, the instruction types for the three functional units are: *logic*, *arithmetic*, and *conversion*. Additionally, for the SAFs evaluation, the (*config*) category represents errors activated by any instruction. Similarly, the (*others*) category, for the TDFs evaluation, describes errors affecting the performance by adding one or more clock cycles (without impacting the produced data). For SAFs, the arithmetic type (i.e., ADD and MAD) is dominant and causes the most errors in all units. In fact, this type is highly used in all applications to configure threads and process operands, thus explaining the results. Moreover, a moderate percentage of errors (from 2.6% to 19.8%) are caused by logic or conversion types. Interestingly, a small percentage of errors (from 0.5% to 4.7%) are classified by any instruction type. In principle, the errors produced by TDFs follow the same trend, and arithmetic-type operations are highly prone to errors in most programs. However, the *Transpose* program shows a high percentage of errors (69.3%) caused by conversion-type instructions. These results imply that the program's coding styles play a major role in the fault impact. Interestingly, a detailed overview of the results shows that a small percentage of the errors produced by TDFs do not corrupt the result but affect the performance (*others* classification from 0.5% to 15.3%) only. As conclusion, the correlation of software errors and instructions types allows the development of accurate fault models mimicking the effects of SAFs and TDFs at higher abstraction levels.

VI. CONCLUSIONS AND FUTURE WORK

This work first proposes a method to characterize the effect of transition delay and stuck-at faults on three GPU's functional units (INTs, FPUs, and SFUs). The results show that, in general, transition delay faults are less likely to produce failures than stuck-at faults. Both transition delay and stuck-at faults may impact a high percentage of structures inside the considered units. Moreover, all sites prone to propagate errors from transition delay faults are also vulnerable to stuck-at faults. Furthermore, the results show that the workload's

coding style plays a significant role in the possible mapping of faults to software errors. These results are a preliminary step in developing accurate high-level error models for faults affecting functional units and support a significant speed-up in the evaluation of safety for GPU applications.

The proposed method can support preliminary steps of the failure mode and effect analysis in the functional-safety domain since the method evaluates and combines the effect of faults with the possible error impacts on running applications. Moreover, the method effectively identifies critical structures affected by stuck-at and transition-delay faults.

In future works, we plan to extend the analyses and identify accurate fault-error correlations for individual instructions toward developing instruction-level error models.

REFERENCES

[1] W. Shi *et al.*, "Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey," *Integration*, vol. 59, pp. 148 – 156, 2017.

[2] F. F. d. Santos *et al.*, "Demystifying gpu reliability: Comparing and combining beam experiments, fault simulation, and profiling," in *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 289–298.

[3] ——, "Analyzing and increasing the reliability of convolutional neural networks on gpus," *IEEE Transactions on Reliability*, vol. 68, no. 2, pp. 663–677, 2019.

[4] S. Hamdioui *et al.*, "Reliability challenges of real-time systems in forthcoming technology nodes," in *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, 2013, pp. 129–134.

[5] C. Lunardi *et al.*, "On the efficacy of ecc and the benefits of finfet transistor layout for gpu reliability," *IEEE Transactions on Nuclear Science*, vol. 65, no. 8, pp. 1843–1850, 2018.

[6] C. Lopez-Ongil *et al.*, "Autonomous fault emulation: A new fpga-based acceleration system for hardness evaluation," *IEEE Transactions on Nuclear Science*, vol. 54, no. 1, pp. 252–261, 2007.

[7] F. F. d. Santos *et al.*, "Demystifying gpu reliability: Comparing and combining beam experiments, fault simulation, and profiling," in *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2021, pp. 289–298.

[8] ——, "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)*, 2021, pp. 292–304.

[9] J. Wei *et al.*, "Analyzing the impact of soft errors in vgg networks implemented on gpus," *Microelectronics Reliability*, vol. 110, p. 113648, 2020.

[10] J. E. R. Condia *et al.*, "An effective method to identify microarchitectural vulnerabilities in gpus," *IEEE Transactions on Device and Materials Reliability*, vol. 22, no. 2, pp. 129–141, 2022.

[11] ——, "Flexgripplus: An improved gpgpu model to support reliability analysis," *Microelectronics Reliability*, vol. 109, p. 113660, 2020.

[12] M. Martins *et al.*, "Open cell library in 15nm freepdk technology," in *Proceedings of the 2015 Symposium on International Symposium on Physical Design*, 2015, p. 171–178.

[13] S. Che *et al.*, "Rodinia: A benchmark suite for heterogeneous computing," in *IEEE international symposium on workload characterization (IISWC)*, 2009, pp. 44–54.