Untestable faults identification in GPGPUs for safety-critical applications

Untestable faults identification in GPGPUs for safety-critical applications

Josie E. Rodriguez Condia[1], *Felipe A. Da Silva*[2 3], *S. Hamdioui*[3], *C. Sauer*[2] *and M. Sonza Reorda*[1]

[1]Politecnico di Torino,
Torino, Italy

[2]Cadence Design Systems,
Munich, Germany

[3]Delft University of Technology,
Delft, The Netherlands

Abstract [1] — **Nowadays, General Purpose Graphics Processing Units (GPGPUs) devices are considered as promising solutions for high-performance safety-critical applications, such as those in the automotive field. However, their adoption requires solutions to effectively detect faults arising in the device during the operative life. Hence, effective in-field test solutions are required to guarantee high-reliability levels. In this paper, we leverage the results of Software-Based Self-Test (SBST) based approaches for GPGPUs by deploying new techniques for automating the identification of untestable faults (UF). Our methodology has achieved fault coverage of 82.8% when applied to an open-source implementation of the NVIDIA G80 GPU architecture. The proposed approach combining SBSTs and UFs identification appears as an effective solution for the reliability analysis of GPGPUs.**

Keywords— ***GPGPUs, SBST, Testing, Untestable faults.***

I. Introduction

General Purpose Graphics Processing Units (GPGPUs) are largely used in applications aiming at efficiently processing large amounts of data (such as in scientific computing and in multimedia applications). Nowadays, these devices are also adopted in complex safety-critical applications, such as the automotive ones [1]. GPGPUs are designed targeting performance and power constraints and thus employ aggressive technology scaling solutions. It has been shown that some implementation technologies are more prone to faults during the lifetime of the device [2] causing unaffordable failures in the safety-critical domain. Hence, in-field test is required to early identify these faults. Unfortunately, the architectural complexity of GPGPUs exacerbates the difficulties in the identification of effective test techniques to be used during in-field operations. While several works exist dealing with the effects of transient faults on GPGPUs, in this paper we focus on permanent faults, which also play a major role in determining the reliability figures of GPGPU-based applications, especially due to the advanced semiconductor technologies they typically rely on.

Test solutions for complex embedded systems can be based either on Design for Testability (DfT) approaches, such as Built-In Self-Test (BIST), or on functional solutions. DfT is effective for the end of the manufacturing test. However, it is not always suitable for in-field test, especially due to its intrusiveness and duration. On the other hand, functional test methods based on Software-Based Self-Test (SBST) employ the Instruction Set of the available CPU modules in the device to perform the test. A test program is employed to apply patterns to each target module and propagate the fault effects to visible locations (e.g., memory), thus allowing their detection.

The SBST approach is currently experiencing a growing success, mainly because it offers the possibility to the semiconductor company manufacturing the device (and knowing its internal structure) to develop the test code, grade it in terms of achieved Fault Coverage (FC), and pass it to the system company, which eventually integrates it in the application code. Test code is often activated in small chunks, fitting in the idle times of the application. These codes are organized in a set of procedures, composing *Self-Test Libraries* (STL). STLs are currently offered by several semiconductor and IP companies. Based on this scenario, the availability of effective STL libraries able to achieve a good FC on a given device could represent a significant added value for products used in safety-critical applications.

Concerning GPGPUs, effective techniques for STL development have not yet been devised, mainly due to their architectural complexity and by the lack of suitable HDL models. In this paper, we first summarize our previous work towards the definition of SBST techniques to detect permanent faults in different GPGPU modules. Our work exploited an open-source VHDL GPGPU model (named *FlexGrip*) of the G80 architecture of NVIDIA [3]. In [4] we proposed some first works targeting the warp scheduler in a GPGPU. In [5], we presented an SBST approach to detect permanent faults in the pipeline registers (PRs) of a GPGPU and for the first time, we reported an experimental evaluation of its effectiveness, based on assessing the achieved FC. However, the FC achieved by the SBST techniques may be difficult to precisely assess, since a significant number of untestable faults (UFs) often exist in hardware models, due to the encoding style or to the design flow characteristics.

According to functional safety standards (e.g., ISO 26262 for automotive), the FC must be computed with respect to a fault list from which UFs are removed. Unfortunately, UF identification can hardly be automated when complex devices such as GPGPUs are considered. Contemplating this scenario, the main contributions of this paper are:

[1] This work has been partially supported by the European Commission through the Horizon 2020 RESCUE-ETN project under grant 722325.

- Demonstration of how combined techniques can leverage UF identification.
- Definition of a semi-automated method to evaluate UFs based on the adoption of Formal Methods.

Experimental results show that the proposed method can identify a significant number of UFs, thus allowing to effectively reducing the effort to assess the quality of an SBST test suite and to improve the accuracy of the computed FC.

The paper is organized as follows: Section II introduces the architecture of the GPGPU model we adopted (FlexGrip), the developed test programs and also presents the UFs and identification methods. Section III describes the proposed method for UF identification and FC improvement. Section IV describes the used environment for the identification of UFs. Section V reports some experimental results and Section VI finally draws some conclusions.

II. Background

GPGPUs are special-purpose processors designed to execute simultaneously multiple tasks in groups (32 *threads* form a *warp*) using Streaming Multiprocessors (SMs). Each SM includes multiple execution units (Scalar Processors, or SPs), caches, (local and shared) memories, Register Files (RFs), a warp scheduler and dispatcher controllers. The SM executes the same instruction on different SPs using particular thread operands. Internally, the SM employs multiple pipeline stages to process one instruction and improve performance.

A. Pipeline registers in FlexGrip

The pipeline registers (PRs) are placed between every couple of pipeline stages to store temporary data from the previous stage and supply data to the next one. In FlexGrip, the PRs are distributed between the five stages in the SM, named *Fetch*, *Decode*, *Read*, *Execute* and *Write-back*. PRs are also placed between the Warp scheduler and the *Fetch* and *Write-back* stages.

The PRs store mainly operands for warp instruction execution. Nevertheless, these also include control information related to the warp instruction status. The *Warp-Fetch* (W-F) PR is composed of 140 control bit-fields representing the status of a warp instruction on the SM. These include the Warp program counter (WPC), the initial and active thread mask (AThM), and parameters for shared memory and general-purpose registers size configuration. The *Fetch-Decode* (F-D) PR, with 237 bits, includes the same information of the previous stage, adding the warp instruction operational code. The *Decode-Read* (D-R) PR (391 bits) stores the specific instructions format fields to activate some operational modes or sub-modules in the next stage. The *Read-Execute* (R-E) PR (302 bits) additionally includes Temporary Registers (TRs), which handle a large number of operands (24,697 bits) and predicate conditions for each SP in the execute stage. The *Execute-Writeback* (E-Wr) PR (251 bits) also contains the TRs (24,704 bits). The high number of bits in the R-E and E-Wr registers is caused by the TRs size. These structures temporarily store operands and results of logical, arithmetical and control-flow operations of each thread on an SP in the SM.

The work reported in this paper has been performed on a modified version of the original Flexgrip model described in [3], where we fixed some bugs related to the implementation of the supported instructions, removed some compiler restrictions and added some extensions. Although the FlexGrip model does not completely match the architecture of the most recent GPGPU devices, the reported results are still mostly valid for them as well. Further details about the improvements we introduced in FlexGrip can be found in [6].

B. Preliminary Test program generation

In [5], we proposed a method to write effective SBST programs to test stuck-at faults in the PRs, based on a bottom-up approach and resorting to multiple parallel programs (*kernels*), which focuses on specific PRs fields. Each kernel is written through a high-level CUDA compiler when possible. Some assembly instructions were added when strictly necessary.

PRs are divided into two groups and multiple subsets for the purpose of SBST design. In [5] we described methods to excite and make observable permanent faults affecting fields in the PRs. Those are the Warp instruction status registers (WPC and AThM) and the Kernel parameter fields (GPRS size, shared memory base fields, others). Restrictions are caused by the CUDA-C compiler environment, which employs advanced algorithms for resource and performance optimization. To circumvent these limitations, combinations of assembly and CUDA-C languages and special coding styles have been developed.

C. Untestable faults

An untestable fault (UF) is a fault for which no test exists. This also means that UFs cannot produce any failure in the operating environment. UFs can be classified as *i)* S*tructural (or combinational) UFs* are not testable even if the combinational block where the fault is located is fully controllable and observable. An ATPG tool can identify these faults. *ii)* S*equential UFs* are faults that cannot be tested due to the sequential behavior of the circuit: for example, some internal states required for the test may not be reachable. *iii) On-line functional UFs* [7] are faults that cannot be tested in a functional manner (i.e., without resorting to DfT) in operational conditions, as defined by the hardware configuration. 0

As our experimental results proved [7, 8], UFs represent a significant percentage of the faults. This may be due to different reasons, such as the used encoding style, the constraints adopted when assembling the whole design, etc.

UFs should be removed from the fault list used during fault simulation experiments aimed at assessing the FC achieved by a given Self-Test Library for two reasons: *i)* They are guaranteed not to produce any relevant failure in the operating conditions; thus, the time spent for their fault simulation is wasted. *ii)* They do not impact the reliability: hence, when assessing the reliability parameters (e.g., during an FMECA process) of a system [9] they should not be considered.

In this work, we focus on RT-level descriptions and on the last two categories of UFs.

D. Untestable faults Identification

UF identification is challenging because a fault can be labeled as untestable only if one can prove that it cannot be tested by ANY functional test stimulus. For that reason, fault simulation cannot identify UFs. The formal analysis appears as a good alternative since it is not limited to a specific time or state. Instead, the scope is global, and every evaluation context is considered. Generally speaking, Formal Tools automatically generate properties, not requiring knowledge of formal languages. In addition, they allow integration with Fault Simulators providing fault lists optimization and reducing simulation campaign duration. This work deploys the automated analysis (*Standard Analysis*, or SA) of the Functional Safety Verification (FSV) app from the Cadence® JasperGold (JG) Formal Verification Platform [10].

The SA is applied as a pre-qualification flow for simulation, to reduce the fault list by identifying UFs. The testability of the faults is determined by verifying: *i)* if there is a physical connection between the fault location and the observation points (strobes); *ii)* if the signals that drive the faulty node allows the activation of the fault; *iii)* if the fault could be observable in at least one strobe of the design.

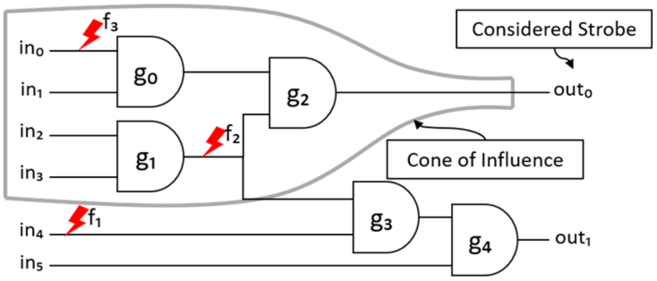

FIG. 1. STANDARD ANALYSIS EXAMPLE (CONE OF INFLUENCE ANALYSIS)

Fig. 1 shows the Structural Examination applied by the SA. This example circuit includes combinational logic (g), inputs (in), outputs (out) and fault targets (f). The following fault behaviors are considered by applying Structural Analysis:

1. The Observation Point (*strobe*) 'out0' only depends on faults in its Cone of influence. Thus, any outside fault 'f1' is considered as UF.

2. Depending on the characteristics of 'g1' drivers, the controllability of 'f2' is defined. If 'g1' always outputs a logic 1, 'f2' would not be controllable for Stuck-at-1 faults. Thus, a Stuck-at-1 fault in 'f2' would be classified as UF.

3. Characteristics of the logic gate 'g2' could propagate a fault 'f3'. If any of the 'g2' (AND gate) inputs is always set to logic 0, the effect of 'f3' would never propagate to 'out0'. Therefore, 'f3'can be classified as UF.

The deployment of formal techniques to reduce the effort of Fault Injection Simulation is explored in different works [11, 12]. An integrated fault analysis flow allows the deployment of the SA before the start of the simulation. The analysis will reduce the number of faults to be simulated by leveraging results for UFs.

III. METHODOLOGY

We aim to combine the efficiency of automatic analysis tools with some assisted checks and structural analysis to identify UFs in complex designs. This approach is integrated into the test program design flow (see Fig 2).

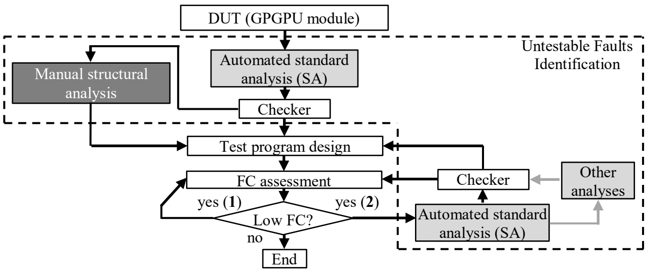
DUT (GPGPU module)
Untestable Faults Identification
Manual structural analysis
Automated standard analysis (SA)
Checker
Test program design
FC assessment
Checker
Other analyses
yes (1)
yes (2)
Low FC?
Automated standard analysis (SA)
no
End

FIG. 2. A GENERAL SCHEME OF THE METHOD TO IDENTIFY UNTESTABLE FAULTS COMBINED WITH THE TEST PROGRAM DESIGN FLOW

Initially, the Device Under Test (DUT) is analyzed in order to identify sub-modules or structures that cannot be tested. This task is performed through a SA analysis. Then, an assisted checker verifies results coherency, considering module operation and tool configuration. An additional manual structural analysis can be performed. Nevertheless, it requires high expertise and deep knowledge of the device operation and architecture. This process can be performed using methods, such as those in [7]. Results from automated and manual methods are combined to reduce the number of faults during test program design.

Test programs are designed using SBST techniques and FC is computed. If the FC is lower than expected, two actions can be taken. The first action (1) consists of test program improvement. The second action (2) is based on complementary UF analysis. Finally, results are used to adjust the test programs or the FC assessment. In the end, two benefits are expected: the UFs identification and the FC improvement.

IV. EXPERIMENTAL RESULTS

2,382 faults were considered in the control-path fields of the PRs during the experiments. The RT-level model of FlexGrip, configured with one SM and 32 SP-cores, was employed and the analysis was limited to the stuck-at faults on the inputs and outputs of the Flip-Flops of each PR. The experiments were performed on a workstation composed of a twelve-core Intel Xeon processor running at 2.5 GHz, and 256 GB of RAM. Fault simulation campaigns required about 6 hours to be completed.

Table 1 shows the features of the 10 SBST kernels we wrote following the proposed techniques. It shows that most of the kernels have a low number of instructions and also a short execution time. We also considered four representative benchmarks for comparison purposes. Their main characteristics are reported at the top of Table 1. Additional details can be found in [5].

Table 1 reports the achieved FC for the applications and the SBST programs. The FC in the applications is obtained as the average result of multiple simulations with various input data sets. Results show a relatively moderate FC (from 32% to 57%) with a high percentage of fault detections as hanging conditions. In contrast, the cumulative FC achieved by all the test kernels is significantly higher (about 66%), and most of the fault effects are visible as a result of data corruption. Initially, UFs were not considered for determining the FC in the previous results. UFs were then identified by a combination of techniques. First, manual UF identification was performed. Then, JG was configured for the SA analysis considering all stuck-at faults inside the SM and their propagation to the strobe outputs. These strobes were defined as the bus connections with the global memory and the output control signals. Moreover, some black boxes replaced the internal memories.

V. EXPERIMENTAL SET UP FOR FC ASSESSMENT

We set up an ad hoc environment to evaluate the stuck-at fault FC. This is based on a fault manager, which translates a fault location into the command sequence for a logic simulator (*ModelSim*). The fault injector tool is composed of a fault controller (FCT), a fault decoder (FDT) and a fault checker and classifier (FCCT).

A fault injection campaign starts by creating the fault list. This list includes all stuck-at faults on each bit of each register. Then, FCT launches a fault-free (*golden*) simulation and the memory results and the kernel time simulation are stored. Afterward, FDT reads one line from the fault list and translates it into the command sequence for Modelsim. This command is executed, and the fault simulation starts. The maximum fault execution time is fixed at twice the golden execution time to consider performance degradation effects by the fault effect.

FCCT compares the memory results and the execution time to classify each fault. Faults are classified in the following categories: *i) Silent Data Corruption fault* (SDC), when the fault generates mismatches in memory, *ii) Hanging (Crash) fault*, if the fault is able to stop or prevent the kernel execution, *iii) Timeout*, if the fault affects the system introducing a delay in the execution and the results are not affected, and *iv) Silent*, when the fault does not affect the system execution and results.

FC is computed as the ratio between the number of faults belonging to the first 3 categories and the total number of faults.

TABLE 1. PERFORMANCE OF THE BENCHMARKS AND THE SBST PROGRAMS

Kernel	**Execution time (Clock C.)**	**Memory size (Bytes)**	**SDC (%)**	**Hanging (%)**	**Timeout (%)**	**FC (%)1**
VectorAdd	28,565	768	18.10	20.82	0.62	**32.37**
MatrixMul	201,365	768	9.74	42.67	0.92	**43.66**
Edge Detection	688,305	2,048	19.89	49.44	1.03	**57.60**
FFT	584,265	512	21.89	42.36	0.67	**53.15**
WS_T_D	16,449	128	4.61	25.23	16.67	38.08
WS_T_V1	2,175	128	4.77	23.33	13.85	34.34
WS_T_V2	1,913	128	4.82	23.64	13.95	34.72
GPR_T_3R	2,273	384	14.51	21.85	0.82	30.35
GPR_T_12R	23,586	8,192	16.77	21.49	0.51	31.74
GPR_T_63R	103,930	400	20.10	22.49	0.56	35.10
B_T	283,714	1,500	9.13	22.51	1.23	26.91
PC_T	31,570	128	21.69	17.59	0.41	38.37
PSR_T	178,750	9,256	19.74	23.54	4.46	39.08
SBST Overall	-	-	38.31	23.44	18.51	**65.70**

Table 2 reports the obtained FC on each PR and some details about the effectiveness of UFs identification techniques. It shows the Testable Fault Coverage (TFC), computed as the ratio between the detected faults and the total number of faults, having removed UFs from the fault list. TFC$_1$ considers UFs identified by the manual method, only, while TFC$_2$ considers the combination of both techniques. Some UFs are simultaneously detected by the two methods.

The manual approach identified 672 UFs in the PRs employing structural information of the model, only. These UFs belong to bit-fields in the WPC register, the initial active thread mask, and some other fields which are present in the design but did not affect the benchmarks or the SBST kernels execution. In contrast, 805 UFs were classified by using the semi-automated method and combining the results. Thus, the FC increased by 2.72% obtaining a TFC$_2$ of 82.98%. It is worth noting that the UFs detected by the automatic methods were not detected during the manual analysis. In the SA UFs, column two values are listed. The first one indicates the total number of UFs. The second one (*in parenthesis*) represents the effective UFs after the assisted checking process. This process removes faults that belong to signals whose value is defined in the configuration phase and remains unchanged during the application execution.

TABLE 2. FC AND UNTESTABLE FAULTS IDENTIFICATION RESULTS

Pipeline Register	**FC (%)**	**Manual UFs**	**TFC1 (%)**	**SA UFs**	**Automated UFs**	**Total UFs**	**TFC2 (%)**
F-D	64.98	72	76.62	84	34	106	83.70
D-R	38.49	72	42.39	290(118)	47	199	45.39
R-E	46.69	256	81.03	162(134)	14	270	84.43
E-Wr	50.0	128	67.11	148(82)	38	166	88.05
Wr-W	65.79	72	90.21	28(0)	0	72	90.21
W-F	68.21	72	91.83	46(18)	0	72	91.83
Overall	**65.70**	**672**	**80.26**	**758(484)**	**133**	**805**	**82.98**

The UFs were classified depending on the location and the intended functionality in the PRs. In the GPGPU, 39.3% of UFs are part of configuration fields, 25.36% belong to execution or data movement operations, 31.27% are fields for thread management and the remaining 4.08% are part of instruction decoding fields. Clearly, the effectiveness of the automated method depends on the target PRs, but in some cases, it significantly improves the results of the manual one. Additionally, it must be highlighted that the effort required by the manual method is significant in terms of skills and time, while the automated methods only require a few hours of computational time.

The combined methods to develop SBST programs and identify UFs seem to be effective for increasing the FC in a target structure. The SBST approach is effective for fault detection on most of the PR fields and the cumulative FC of all kernels reaches a relatively high percentage. The Signature per Thread strategy was crucial in the process of detecting faults in the GPGPU [5]. Similarly, the UFs identification increased the TFC by up to 17.2%. Nevertheless, it is worth noting that the verification of formal properties in a design may be computationally hungry by the excessive number of operations, such as in the propagation analysis. For that reason, it may be more convenient to apply formal methods either to single modules within the whole device or to perform it after some preliminary screening, e.g., on the faults that were not classified after fault injection simulation.

VI. CONCLUSIONS

In this work, we face the issue of effectively in-field testing GPGPU modules with respect to permanent faults. One possible solution lies in adopting functional solutions, such as SBST approaches. In such a case, identifying untestable faults allows first to reduce the fault simulation cost required to compute the Fault Coverage, and secondly to correctly compute the same figure, pruning the fault list from untestable faults. For this purpose, we experimentally evaluated a solution based on the adoption of a commercially available tool originally intended for design validation, showing that it is able to both reduce the required effort and improve the obtained results.

The proposed method for untestable faults identification was able to identify a significant number of them with a reduced effort and could more precisely assess the Fault Coverage achieved by the Self-Test Library we developed for the GPGPU module. Work is currently being done to improve the Self-Test Library (in terms of achieved Fault Coverage and targeted modules) and to adopt further functionalities of the Cadence suite, thus increasing the effectiveness of the approach.

REFERENCES

[1] W. Shi, M. B. Alawieh, X. Li, and H. Yu, "Algorithm and hardware implementation for visual perception system in autonomous vehicle: A survey," *Integration,* vol. 59, pp. 148-156, 2017/09/01/ 2017.

[2] S. Hamdioui, D. Gizopoulos, G. Guido, M. Nicolaidis, A. Grasset, and P. Bonnot, "Reliability challenges of real-time systems in forthcoming technology nodes," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2013, pp. 129-134.

[3] K. Andryc, M. Merchant, and R. Tessier, "FlexGrip: A soft GPGPU for FPGAs," in *2013 International Conference on Field-Programmable Technology (FPT)*, 2013, pp. 230-237.

[4] B. Du, J. E. R. Condia, M. Sonza Reorda, and L. Sterpone, "About the functional test of the GPGPU scheduler," presented at the On-Line Testing Symposium (IOLTS) 2018 IEEE 24th International, 2018.

[5] J. E. R. Condia and M. Sonza Reorda, "Testing permanent faults in pipeline registers of GPGPUs: A multi-kernel approach," presented at the 25th IEEE International Symposium on On-Line Testing and Robust System Design (IOLTS), 2019, to appear.

[6] B. Du, J. E. R. Condia, and M. Sonza Reorda, "An extended model to support detailed GPGPU reliability analysis," presented at the 14th IEEE International Conference on Design & Technology of Integrated Systems in Nanoscale Era (DTIS), 2019.

[7] R. Cantoro, A. Firrincieli, D. Piumatti, M. Restifo, E. Sanchez, and M. Sonza Reorda, "About on-line functionally untestable fault identification in microprocessor cores for safety-critical applications," in *2018 IEEE 19th Latin-American Test Symposium (LATS)*, 2018, pp. 1-6.

[8] C. Gursoy and e. al., "New categories of Safe Faults in a processor-based Embedded Systems," presented at the 22th International Symposium on Design and Diagnostics of Electronic Circuits&Systems (DDECS), Cluj-Napoca, Romania, 2019, to appear.

[9] H.-L. Ross, *Functional Safety for Road Vehicles: New Challenges and Solutions for E-mobility and Automated Driving*: Springer Publishing Company, Incorporated, 2016.

[10] I. Cadence Design Systems, "JasperGold Functional Safety Verification App User Guide," Product version 2018.03 ed. ed, 2018.

[11] K. Devarajegowda and J. Vliegen, "Deploying formal and simulation in mutual-exclusive manner using jaspergolds proofcore technology," presented at the Cadence User Conference CDNLive EMEA, 2017.

[12] S. Marchese and J. Grosse, "Formal fault propagation analysis that scales to modern automotive SoCs," presented at the 2017 Design and Verification Conference and Exhibition DVCON Europe, 2017.