Chapter 4

Design techniques to improve the resilience of computing systems: software layer

Alberto Bosio[1], Stefano Di Carlo[2], Giorgio Di Natale[3], Matteo Sonza Reorda[2], and Josie E. Rodriguez Condia[2]

Hardware techniques to improve the robustness of a computing system can be very expensive, difficult to implement and validate. Moreover, they require long evaluation processes that could lead to the redesign of the hardware itself when reliability requirements are not satisfied. This chapter will cover the software techniques that allow improving the tolerance of the system to hardware faults by acting at software level only. We will cover the recently proposed approaches to detect and correct transient and permanent faults.

4.1 Introduction

This chapter presents the reliability issues and solutions targeting the software layer of a computing system. The software layer plays an important role from the system reliability point of view. Indeed, software can either mask or amplify errors thus improving or reducing the overall computing system reliability. This is the main idea behind the concept of Software-Implemented Fault Tolerance (SWIFT) Techniques: how to write the software in order to maximize the error-masking effect.

Before moving to the details of software level fault-tolerant techniques, let us first introduce some basic concepts. Figure 4.1 depicts a simple view of a computing system divided into hardware and software layers. From the figure, it is possible to identify the "propagation" of the hardware faults (i.e., Physical faults) through the hardware layers composing the computing system. Some of these faults are masked by hardware layers, while some others reach the software layer. It is interesting to point out that at software level two more source of faults can be identified: the presence of **bugs** and the **misuse** of the software external User Interface (UI). These sources are completely independent of the hardware level.

[1]Lyon Institute of Nanotechnology, École Centrale de Lyon, Lyon, France
[2]Department of Control and Computer Engineering, Politecnico di Torino, Torino, Italy
[3]TIMA Laboratory, CNRS, Grenoble, France

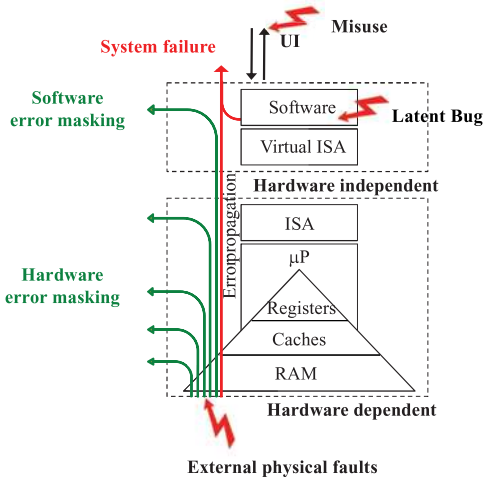

Figure 4.1 System layers and fault propagation

The chapter is structured as follows: Section 4.2 presents the taxonomy of the faults affecting the software layer. Section 4.3 reviews the existing Software-Implemented Hardware Fault Tolerance (SIHFT) solutions. Section 4.4 describes the techniques for Software-Based Self-Test (SBST), while Section 4.5.1 focuses the analysis of SBST solutions for GPUs.

4.2 Fault taxonomy

As already pointed out in the introduction, faults affecting software have different sources that can be classified using the following definitions:

- **Design faults**: these faults are introduced during the software implementation. Usually this kind of faults is referred to as *bugs*.
- **Physical faults**: these faults are originated at hardware level and reach the software level through propagation.
- **Interaction faults**: these faults are due by interaction between the software level and the external environment.

Independently from the earlier sources, faults can be further classified by the following characteristics:

- **Intent**: the fault can be intentionally or not introduced into the software. In the first case, the intent is to obtain a malfunction of the software, and in the literature the term *malicious* faults is usually adopted. In the second case, the term *non-malicious* fault is adopted [1].
- **Nature**: the fault is defined as *Permanent* if it is always present. The fault is defined as *Transient* if it appears at a certain time and then disappear.

- **Effect**: the fault effect can impact a single or multiple locations. The location can be either a variable or an instruction.
- **Impact**: the fault can have different impacts at application-level:
  - *Hang*: the application does not terminate within a reasonable time interval. The time interval depends on the application itself.
  - *Silent Data Corruption (SDC)*: the output of application has been corrupted.
  - *Data Undetected Error (DUE)*: an unexpected exception, assertion, or segmentation fault, deadlock or interrupt occurred.
  - *Masked*: no mismatch at the application output.

Table 4.1 presents the fault taxonomy through a fault source/characteristic matrix. Each row corresponds to one fault characteristic, while each column corresponds to fault source. As it can be seen, independently from the fault source, all the fault characteristics have to be considered. For example, a design fault can be malicious if the software code has been intentionally modified to introduce the fault itself. The latter can be a Trojan or a virus [2,3]. In the same way, an interaction fault can be malicious too, and in this case we have to consider the case of intentionally misuse that typically occurs during an *attack* [4]. Malicious Physical faults have been intentionally introduced into the hardware level of the system (e.g., Trojan) [5].

The next subsection will present how faults are modeled at software level.

4.2.1 Software faults

Table 4.2 reports a detailed list of Software fault models induced by hardware faults. They can be grouped into three main categories:

- **Data fault models**: they enable to model faults corrupting data processed by a software application. They include (i) Wrong Data in an Operand, (ii) Not-accessible Operand and (iii) Operand Forced Switch;
- **Code fault models**: they enable one to model faults that corrupt the set of instructions composing a program. They include (i) Instruction Replacement, (ii) Faulty Instruction and (iii) Control Flow Error.
- **System fault models**: they enable one to model both timing faults and communication/synchronization faults during the software execution. They include (i) External Peripheral Communication Error, Signaling Error, Execution timing Error and Synchronization Error.

Table 4.1 Software fault taxonomy/characteristic matrix

	Design fault	**Interaction fault**	**Physical fault**
Intent		Malicious, non-malicious	
Nature	Permanent	Permanent, Transient	Permanent
Effect		Single, multiple	
Impact		Hang, SDC, DUE, Masked	

Figure 4.2 shows an example of the earlier fault modeling. It represents the multiplication instruction as specified in the ARM© Instruction Set Architecture (ISA) [6]. We will consider the case of three different faults (F1, F2 and F3) affecting different locations in different time. F1 affects the portion of the instruction responsible of the encoding of the destination register (Rd). Due to F1, it is possible that the Rd changes so that the result will be stored in a different register w.r.t. the fault-free one. This case is modeled by the Data fault model and more specifically by the Source Operand Forced Switch model. F2 affects the instruction opcode. This case may lead to a different opcode and thus the microprocessor decodes the faulty instruction as a different one w.r.t. the fault-free one. This case is modeled by the Code fault models

Table 4.2 Software fault models

Software fault model	**Description**
Wrong Data in a Operand	An operand of the ISA instruction changes its value
Not-accessible Operand	An operand of the ISA instruction cannot change its value
Source Operand Forced Switch	An operand is used in place of another
Instruction Replacement	An instruction is used in place of another
Faulty Instruction	The instruction is executed incorrectly
Control Flow Error	The control flow is not respected (control-flow faults)
External Peripheral Communication Error	An input value (from a peripheral) is corrupted or not arriving
Signaling Error	An internal signaling (exception, interrupt, etc.) is wrongly raised or suppressed
Execution timing Error	An error in the timing management (e.g., PLL) interferes with the correct execution timing
Synchronization Error	An error in the scheduling processes causes an incoherent synchronization of processes/tasks

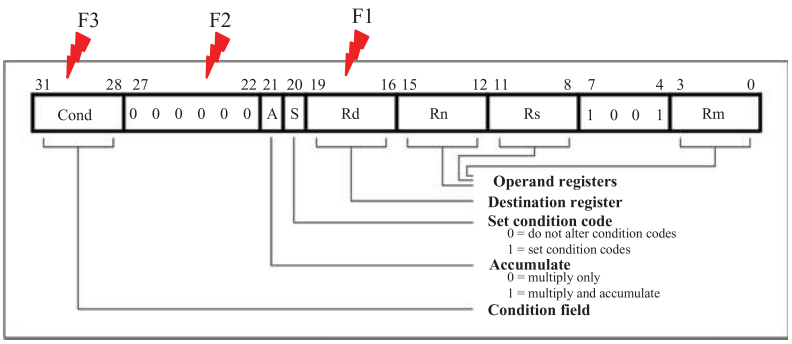

Figure 4.2 Fault modeling example

and more specifically by the Instruction Replacement. Finally, F3 affects the condition flags (cond) of the instruction. This case may lead to have an erroneous flag thus impacting the control flow of the program. This case is modeled by the Code fault models and more specifically by the Control Flow Error.

4.3 Software-Implemented Hardware Fault Tolerance

The concept of Commercial Off-The Shelf (COTS) hardware and software components has been introduced in safety-critical applications. These components guarantee high performance at the price of a low dependability. Since COTS hardware cannot be modified to introduce fault-tolerant mechanisms, the only possibility is to protect systems acting at the software layer. More in particular, the low-cost solution is to take advantage of SIHFT techniques that allow, by only using software, to detect and correct errors affecting the hardware.

SIHFT techniques are, in general, based on the addition, to the original target application, of software routines able to check the validity and correctness of the executed code and the managed data [7]. This section presents recent SIHFT techniques to guarantee the correct behavior of the system, even in the presence of hardware faults. Most of the existing solutions are inspired by equivalent solutions implemented in hardware but then adapted in software so that their cost is reduced. These techniques can be classified into two main categories:

1. techniques that modify the software in order to reduce the probability of fault occurrences;
2. techniques that allow detecting/tolerating the presence of an error: mainly based on redundancy, control flow integrity, checkpoints/rollbacks and the so-called Algorithmic-Based Fault Tolerance (ABFT).

The following subsections will detail each of the previous categories.

4.3.1 Modify the software in order to reduce the probability of fault occurrences

These techniques mainly aim at modifying the code source in order to use in a more smart way the hardware resources. The ultimate goal is to reduce the probability that a fault-affecting hardware resources will propagate to the software.

Let us resort to an example depicted in the assembly code of Listing 4.1. The reader can notice that register $r0$ is written at line 2 and then read at line 5. This means that the *lifetime* of such a register corresponds to three cycles.* The point here is that higher the lifetime higher the exposure time and thus higher the probability to observe a single event upset. By simply change the code source, it is possible to minimize the lifetime and thus reduce the probability of observing a fault at hardware level. The

*For the sake of simplicity, we consider that each instruction needs one clock cycle to be executed.

code shown in Listing 4.2 provides a lower fault probability because the lifetime of $r0$ has been reduced to 1 w.r.t. to the first code.

In [8–10] works, the main idea is to perform instruction rescheduling (after the performance-optimized scheduling) to reduce the vulnerable periods of registers. The main drawback of such approaches is that register file covers only a small portion of the processor layout. As a result, these techniques provide limited reliability improvement (from 2% to 9%) w.r.t. to a normal code.

4.3.2 Detecting/tolerating the presence of an error

N-Version programing is probably the most applied software level fault diversity technique [11]. The idea behind N-version programing is the development of N implementations of the same software application (with $N > 2$) by an independent development team. These versions are all functionally equivalent, i.e., they implement the same functionalities, but given the different instruction flow, they expose different failure characteristics that increase the likelihood that not all versions fail at the same time in a specific fault scenario. N-version programing can be coupled with redundancy techniques such as Dual Modular Redundancy and Triple Modular Redundancy.

When recovery from failures is a key point, software-based checkpoint recovery techniques are an interesting solution [12]. The overall idea when implementing checkpointing is to modify the software by inserting checkpoint instructions inside the code. A good practice to decide where checkpoint instructions must be inserted is to identify instructions with high error probability and to place checkpoints just before these instructions. Inserting a checkpoint means inserting calls to proper routines able to save the state of the program in a reliable storage area. In the case of failure, the program execution can be restarted from a safe state by restoring the latest

```
1  mov r0, @a
2  inc r0   ; r0 write
3  mov r1,@b
4  add r2, r1
5  mov @a, r0 ; r0 read
```

Listing 4.1 Asm example

```
1  mov r0, @a
2  inc r0   ; r0 write
3  mov @a, r0 ; r0 read
4  mov r1,@b
5  add r2, r1
```

Listing 4.2 Asm example: reduced lifetime

saved checkpoint. The literature is the reach of software-based checkpointing techniques. Compilers can be modified in order to assist the insertion of checkpoints as proposed in [13] that propose the use of an adaptive scheme to minimize the storage overhead required to save the checkpoints. Software libraries such as Libckpt [14] and libFT [15] have been released to the developers to facilitate the task of dumping the state of a program when performing checkpointing. However, full checkpointing automation is still not supported. At the Operating System (OS) level, [16] proposes a loadable kernel module for providing application-aware reliability and dynamically configuring reliability mechanisms. The module is implemented in Linux and supports the detection of application/OS failures and transparent application checkpointing.

Besides diversity and checkpointing, several software and compiler level techniques propose protection schemes based on data redundancy and control flow checking.

Data error detection using Duplicated Instructions [17] and SWIFT [18] and REliable Code COmpliler [10] represent the most famous software redundancy techniques based on duplicated instructions followed by checkpoint instructions able to compare the result of the two executions usually placed before store and/or conditional branches. These techniques generate a significant performance and memory overhead due to redundant instruction execution and shadow memory locations to store redundant data, respectively. Performance overhead can be also aggravated by the increased cache usage to hold redundant data for computation of original and duplicated instructions, generating additional memory traffic.

Control flow checking techniques instead aim at verifying that the control flow of the application is properly respected during the execution. The program is usually split into elementary blocks of instructions with a single entry and a single exit, usually referred to as basic blocks. A reference signature representing the correct execution flow in the blocks is calculated off-line and stored. At run-time the same signature is calculated again and compared with the golden one. Software-based control flow checking techniques insert appropriate instructions to compute the execution signature at run-time.

Different techniques such as Block Signature Self Checking [19], Control Checking with Assertions [20], Control-Flow checking via regular expressions [21] and Control Flow Checking by Software Signatures [22].

Similar to data redundancy techniques, also control-flow checking techniques may introduce significant overhead in the software execution associated with the tasks of computing and checking the software signatures.

A completely different approach is to implement fault tolerance techniques at the algorithm level by exploiting the characteristic of specific computations implementing the so-called ABFT [23]. Figure 4.3 shows a simple example of ABFT application. The algorithm is the matrix multiplication. Here it is possible to exploit the property of the algorithm in order to add an extra row and column of the two matrices. These extra elements contain a kind of code (in the example is the sum of the elements). After the multiplication, the extra row and column will satisfy the same property. It is thus possible to identify which element of the matrix has been affected by a fault.

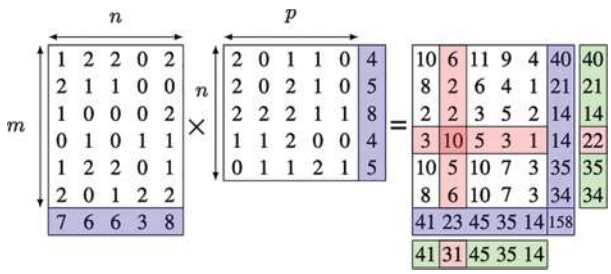

Figure 4.3 ABFT example

4.4 Software-Based Self-Test

In the last decades, electronic systems have been increasingly used in safety-critical applications, where the effects of possible faults affecting the hardware may have severe consequences. Several solutions were introduced to early detect the presence of permanent faults, or to mitigate their effects. The latter are based on (hardware, information, time) redundancy, the former involves in-field test, which allows detecting possible permanent faults arising during the operational phase (e.g., due to aging) before they cause serious consequences. In this way, the resulting failure probability can be decreased. In-field test of an embedded system can be performed in different ways depending on the considered scenario and the required reliability targets. In some cases, in-field test is performed at the system power-on, before the real application is started. In other cases, it is performed periodically, often exploiting the idle times of the application. Alternatively, the in-field test is performed concurrently to the application, e.g., by monitoring the produced results. In any case, in-field test must take the form of self-test, since no support from the outside can be provided, and must minimize the intrusiveness with respect to the resources used by the application. It is worth emphasizing that the activation frequency of in-field test and the fault coverage it must achieve are higher when semiconductor technologies with lower reliability are used. Since the latest technologies are known to be less reliable, their adoption in safety-critical systems makes the constraints on in-field test even harder. Different alternatives exist to implement effective in-field test solutions in a device used for a safety-critical application. If the device is specifically developed for that application, and hardware overhead constraints allow for that, solutions based on Design for Testability (DfT), e.g., Logic BIST, can be successfully exploited. This approach has several advantages, including a good support from commercial EDA tools, the ability to reach a high fault coverage (at least for static faults), and the possibility to reuse at least some of the hardware infrastructures already used for end-of-manufacturing test. On the other side, its main drawback lies in the fact that the required DfT hardware must be introduced early in the design flow, and the mechanism for its access from the outside must be agreed between the semiconductor company producing the device and the system company managing the in-field test. Moreover, when each activation

of the in-field tests must fit into relatively short-time slots, this solution may not be suitable. As an alternative, in the last years several semiconductor and IP companies, including Infineon [24], STMicroelectronics [25], Renesas [26], Cypress [27], Microchip [28], ARM [29], started adopting a solution based on the so-called Self-Test Libraries (STLs). The idea is to use the CPU existing in most of the considered devices and to develop a set of procedures, whose execution can be easily triggered by the application software or by the OS (if any). When executed, these procedures perform a suitable sequence of operations, able to trigger possible permanent faults in the CPU or in other modules and to produce results that can reveal the existence of the faults. This approach is known in the literature as SBST [30]. Since STLs are developed by the semiconductor or IP companies, which know the structure of the hardware, the fault coverage (e.g., in terms of stuck-at faults) that STLs can achieve can be computed via fault simulation. This approach provides a nice compromise between the requirements of semiconductor and IP companies, which want to preserve the property of their hardware but must provide a flexible and effective solution for its test, and those of the system companies, which must test in-the-field the different devices composing their systems to achieve a given reliability or safety. As a further advantage, this approach performs a test of the whole device while it is operating in the same conditions of the application and can thus detect defects (e.g., delay or interconnection defects) that can hardly be caught by the DfT-based solutions. Finally, it is worth mentioning the fact that being based on the execution of a piece of code, a test based on SBST can be easily changed during the product life, e.g., to target new defects. On the other side, the major limitation of the solution based on STLs lies in the cost for their development, since this activity must be done manually without very limited support by EDA tools.

4.4.1 Basics on SBST

The idea of using a piece of code to test a CPU was first proposed several decades ago [31] to face the scenario in which the CPU was a simple processor, and its ISA and basic architecture were known only. More recently, the same idea was exploited to support end-of-production test of high-end processors, with the main goal of avoiding the usage of expensive high-frequency testers [30]. A similar approach found applications in industry to support silicon debug and speed binning [32]. A comprehensive overview about the usage of SBST for end-of-manufacturing CPU testing can be found in [33]. Similar solutions were also explored for testing communication peripheral components [34], system peripherals [35], and on-chip memories [36], including caches [37]. The growing interest toward in-field test pushed researchers to analyze how SBST could be effectively used in that domain. In principle, SBST has several nice properties, as mentioned earlier. However, some key points must be faced, which are not relevant when SBST is used for end-of-production test, such as (i) how to trigger the execution of each test procedure, (ii) how to retrieve the results, (iii) how to limit the invasiveness of each test procedure while still maintaining the final fault coverage, (iv) how to limit the duration of each test procedure to the maximum allowed time, (v) how to write the test code such that it complies with the

coding stiles and rules that are valid for the application software. Some first analysis of the issues connected to the usage of SBST for on-line test, together with some first solutions are reported in [38]. In [39], some examples of solutions adopted on real test cases from industry are reported, while algorithms for automatically compacting existing test programs to reduce their size [40] or duration [41] have been recently developed. Finally, the work in [42] shows that formal techniques can be successfully used to automate the generation of test programs to be used for in-field test of pipelined processors. Current challenges in the area of SBST include the techniques for developing and optimizing STLs for multicore systems, and the solutions for addressing special categories of faults, such as the performance faults, i.e., those faults that only impact on the performance of a system, while still producing correct result values [43]. Extension of SBST techniques to special types of computing elements, such as VLIW processors [44] or GPGPUs [45] is also a hot topic at the moment.

4.5 SBST for GPGPUs

This section first summarizes the state of the art in terms of SBST solutions for permanent faults in GPGPUs. Then it shows that the potential effects of permanent faults in critical units of GPGPUs may become relevant. Finally, some SBST techniques are introduced to detect those faults.

4.5.1 Introduction

GPGPUs are an effective solution to speed up massively parallel computation and are mainly employed as accelerators in highly data-intensive applications such as video, image and multi-signals processing, due to their powerful parallel architecture. Nowadays, these devices are promising solutions for new low-energy, real-time and high-performance applications with safety-critical requirements, such as autonomous automotive drivers and autonomous industrial machines [46]. As commented below, in order to match the requirements for these applications, these devices are designed using aggressive technology scaling techniques, thus increasing the fault-rate across the operational lifetime, mainly because these devices are prone to internal and external sensitive effects, such as aging and radiation [47–49]. Moreover, traditional end-of-manufacturing test solutions cannot guarantee the correct operation and unexpected misbehaviors could arise in the application. When considering system integration companies developing GPGPU-based solutions in safety-critical domains, a critical issue is that these companies often do not have detailed knowledge of the implementation of the adopted GPGPU devices. In this context, functional test techniques represent a viable solution to guarantee the correct in-field operation. This issue becomes critical when a product for safety-critical environments should follow industrial standards, such as the ISO 26262 for the automotive applications. The adoption of SBST techniques for GPGPU devices is feasible in such a scenario, although the cost of developing effective test solutions for such complex devices,

including large numbers of parallel execution units, may be challenging. By principle, SBST techniques introduce zero hardware overhead. However, restrictions of execution cycles, resource overhead,and power consumption should be considered. Moreover, GPGPUs are mainly special-purpose processors and potentially all previous SBST solutions for single-core processor devices can be adapted to these parallel devices. Some SBST solutions for GPGPUs have been proposed in the past. Some of them focus on data-path modules, including the register files and the execution units [45] using adaptations of well-known SBST programs for single and multicore processors. Other works [50] employed internal thread identifiers to schedule tasks in the GPGPU, avoiding corrupted units and mitigating errors in the application. In [51], the authors proposed new mitigations strategies to face permanent faults in the processing core units, or Streaming Multiprocessors (SMs) (in Nvidia's terminology) employing a reverse engineering approach for the block scheduler policies and distributing the application blocks across the fault-free units. Finally, the work in [52] analyzed the fault sensibility and its relation with the employed sub-modules and program description.

4.5.2 Effects of permanent faults in GPGPU devices

One important cause of permanent faults in GPGPU devices lies in the aging effects damaging the hardware integrity. In most multimedia applications, a fault located in the data-path can generate errors in the output. Nevertheless, some of these could be tolerated due to the graphical nature of the application (they only produce slight degradation of the image quality, which is often even difficult to detect). On the other hand, a fault located in a control-unit can generate severe consequences for the running application. When hitting a sensitive location, a fault could generate execution hanging or thread execution missing. In order to present an example showing the effects of permanent faults affecting the control logic of a GPGPU device, we considered an image preprocessing application (edge-detection) and performed some experiments resorting to the GPGPU-SIM simulator [53]. In this case study, a permanent fault is injected in a memory cell of the scheduler. The fault prevents the execution of a particular thread in the program kernel. During the execution of the GPGPU program, the affected thread can partially damage the neighbor thread results introducing errors. Results are graphically visible in the produced image (Figure 4.4), as the reader can see, the effects are far from being negligible. The previous application shows the impact of one permanent fault in a sensitive location. One or a few faults in a more complex and critical application could produce critical misbehaviors compromising the entire execution. SBST solutions can be applied to detect permanent faults in special-purpose units of a GPGPU. The next subsection introduces some strategies applied to control units in GPGPU devices.

4.5.3 SBST techniques for testing the GPGPU scheduler

Developing effective SBST procedures for control-path modules in processor-based systems is not a trivial task. This is true also for GPGPU-based systems. The warp

Figure 4.4 Effects of one permanent fault in the control unit of a GPGPU. Original fault-free gray-scale image (left), fault-free edge-detection output image (top-right) and faulty edge-detection output image (bottom-right).

scheduler is one of the control-path modules, and it is a critical unit for the GPGPU operation. This unit manages the parallel execution of multiple threads inside the SM, and the detection of permanent faults in this unit is crucial to avoid the application collapsing. The basic functions of this unit are (i) warp submission, (ii) warp execution checking (this process is done after finishing each instruction by the warp and updating the related information) and (iii) warp termination. A fault in this unit is able to generate critical issues, such as execution hanging, performance degradation and SDC effects. This module includes some sub-modules, such as warp generators, dispatchers and checkers. Additionally, some special-purpose memories are included. One of these memories is the status warp pool memory. This memory stores the status information of each warp dispatched to the SM in an entry line. Each entry line is composed of a Thread-Mask (ThMk) field, indicating the number of active threads

per warp, the warp program-counter (WPc) field and some other fields. In [54], some approaches to detect faults affecting the warp scheduler based on the SBST solution have been proposed. The authors used the available instructions to design SBST programs targeting permanent faults in the warp pool memory of a GPGPU. These techniques are mainly based on combinations of multiple instructions and clever algorithmic mechanisms to generate the input sequences to the targeted unit in order to make the faulty effects visible. In this work, architectural information about the targeted GPGPU was available, and it was possible to use it to develop a suitable test for each field of each entry line inside the memory. The proposed algorithms are based on a sequence of subroutines to generate stimuli able to write to and read from a specific field inside the warp entry line. The targeted fields in the entry line were the ThMk and WPc. These fields depend on control-flow instructions. ThMk can be written by adding multiple combinations of conditional control-flow instructions in the program kernel. The WPc field can be modified through unconditional control-flow instructions. A major difference with respect to other strategies is the observability mechanism, based on signatures. The method that presents better results implements the test by means of a subroutine, which computes one signature per thread to check its correct execution and hence detect possible faults affecting the ThMk and WPcm fields. The subroutine execution generates thread divergence. Moreover, this changes the program counter location. Then, on each path (taken and not-taken) each thread modifies its signature, allowing the fault detection. At the end, the signature is stored in global memory and checked by the host. This strategy takes the advantage of supported instructions and includes zero hardware overhead in the system. A moderate memory overhead is required and the total number of required memory locations is equal to the number of threads per block to be executed by the SM. A comparison between a reference application, a typical embarrassing parallel application (denoted as Basic), and the proposed approach is shown in Figure 4.5. Results show that the proposed approach increases the percentage

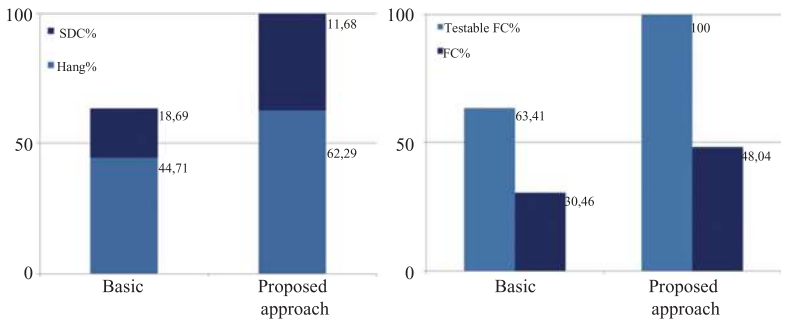

Figure 4.5 *Comparison of fault coverage between a typical parallel application and the proposed method to detect permanent faults in the scheduler warp pool memory of a GPGPU (detailed testable FC (left), testable FC and FC (right))*

of permanent faults detection. The method is able to reach the 100% of detectable fault coverage. The detectable fault coverage corresponds to the total number of faults that can be detected using SBST strategies. On the other hand, the total fault coverage is lower. This difference occurs due to the presence of faults that cannot be tested, e.g., because they relate to unused memory bits.

References

[1] Avizienis A, Laprie JC, Randell B, *et al.* Basic Concepts and Taxonomy of Dependable and Secure Computing. IEEE Transactions on Dependable and Secure Computing. 2004;1(1):11–33.

[2] Avizienis A and He Y. Microprocessor entomology: A taxonomy of design faults in COTS microprocessors. In: Dependable Computing for Critical Applications 7; 1999. p. 3–23.

[3] Xiao G, Zheng Z, Yin B, *et al.* Experience report: Fault triggers in Linux operating system: From evolution perspective. In: 2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE); 2017. p. 101–111.

[4] Singh YN and Singh SK. A Taxonomy of Biometric System Vulnerabilities and Defences. International Journal of Biometrics. 2013;5(2):137–159. Available from: http://dx.doi.org/10.1504/IJBM.2013.052964.

[5] Xiao K, Forte D, Jin Y, *et al.* Hardware Trojans: Lessons Learned After One Decade of Research. ACM Transactions on Design Automation of Electronic Systems. 2016;22(1):6:1–6:23. Available from: http://doi.acm.org/10.1145/2906147.

AQ4 [6] ARM ISA. Accessed: 2019-06-27. https://www.arm.com.

[7] Benso A, Di Carlo S, Di Natale G, *et al.* Data criticality estimation in software applications. In: International Test Conference, 2003. Proceedings. ITC 2003. vol. 1; 2003. p. 802–810.

[8] Rehman S, Shafique M, and Henkel J. In: Introduction. Cham: Springer International Publishing; 2016. p. 1–21. Available from: https://doi.org/10.1007/978-3-319-25772-3_1.

[9] Xu J, Tan Q, and Shen R. The instruction scheduling for soft errors based on data flow analysis. In: 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing; 2009. p. 372–378.

[10] Benso A, Chiusano S, Prinetto P, *et al.* A C/C++ source-to-source compiler for dependable applications. In: Proceeding International Conference on Dependable Systems and Networks. DSN 2000; 2000. p. 71–78.

[11] Avizienis A. The N-Version Approach to Fault-Tolerant Software. IEEE Transactions on Software Engineering. 1985;(12):1491–1501.

[12] Koo R and Toueg S. Checkpointing and Rollback-Recovery for Distributed Systems. IEEE Transactions on Software Engineering. 1987;(1):23–31.

[13] Li CC and Fuchs WK. Catch-compiler-assisted techniques for checkpointing. In: Digest of Papers. Fault-Tolerant Computing: 20th International Symposium. IEEE; 1990. p. 74–81.

[14] Plank JS, Beck M, Kingsley G, *et al.* Libckpt: Transparent Checkpointing Under Unix. Computer Science Department; 1994.
[15] Huang Y and Kintala C. Software implemented fault tolerance: Technologies and experience. In: FTCS. vol. 23. IEEE Computer Society Press; 1993. p. 2–9.
[16] Wang L, Kalbarczyk Z, Gu W, *et al.* An OS-level framework for providing application-aware reliability. In: 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06). IEEE; 2006. p. 55–62.
[17] Oh N, Shirvani PP, and McCluskey EJ. Error Detection by Duplicated Instructions in Super-Scalar Processors. IEEE Transactions on Reliability. 2002;51(1):63–75.
[18] Reis GA, Chang J, Vachharajani N, *et al.* Software-Controlled Fault Tolerance. ACM Transactions on Architecture and Code Optimization. 2005;2(4):366–396. Available from: http://doi.acm.org/10.1145/1113841.1113843.
[19] Miremadi G, Harlsson J, Gunneflo U, *et al.* Two software techniques for on-line error detection. In: Digest of Papers. FTCS-22: The Twenty-Second International Symposium on Fault-Tolerant Computing. IEEE; 1992. p. 328–335.
[20] Alkhalifa Z, Nair VS, Krishnamurthy N, *et al.* Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection. IEEE Transactions on Parallel and Distributed Systems. 1999;10(6):627–641.
[21] Benso A, Di Carlo S, Di Natale G, *et al.* Control-flow checking via regular expressions. In: Proceedings 10th Asian Test Symposium. IEEE; 2001. p. 299–303.
[22] Oh N, Shirvani PP, and McCluskey EJ. Control-Flow Checking by Software Signatures. IEEE Transactions on Reliability. 2002;51(1):111–122.
[23] Huang K-H and Abraham JA. Algorithm-Based Fault Tolerance for Matrix Operations. IEEE Transactions on Computers. 1984;C-33(6):518–528.
[24] Infineon. 2018. https://www.hitex.com/software-components/selftest-libraries-safety-libs/pro-sil-safetcore-safetlib/.
[25] STMicroelectronics. AN3307 – Application note. In Guidelines for Obtaining IEC 60335 Class B Certification for any STM32 Application; 2016.
[26] Renesas. 2018. https://www.renesas.com/en-eu/products/synergy/software/add-ons.html#read.
[27] Cypress. AN204377 FM3 and FM4 Family, IEC61508 SIL2 Self-Test Library; 2017.
[28] Microchip. DS52076A 16-bit CPU Self-Test Library User's Guide; 2012.
[29] ARM. 2018. https://developer.arm.com/technologies/functional-safety.
[30] Chen L and Dey S. Software-Based Self-Testing Methodology for Processor Cores. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 2001;20(3):369–380.
[31] Thatte SM and Abraham JA. Test Generation for Microprocessors. IEEE Transactions on Computers. 1980;C-29(6):429–441.
[32] Parvathala P, Maneparambil K, and Lindsay W. FRITS – A microprocessor functional BIST method. In: Proceedings. International Test Conference; 2002. p. 590–598.

[33] Psarakis M, Gizopoulos D, Sanchez E, *et al.* Microprocessor Software-Based Self-Testing. IEEE Design Test of Computers. 2010;27(3):4–19.

[34] Apostolakis A, Gizopoulos D, Psarakis M, *et al.* Test Program Generation for Communication Peripherals in Processor-Based SoC Devices. IEEE Design Test of Computers. 2009;26(2):52–63.

[35] Grosso M, Perez WJH, Ravotto D, *et al.* A software-based self-test methodology for system peripherals. In: 2010 15th IEEE European Test Symposium; 2010. p. 195–200.

[36] van de Goor A, Gaydadjiev G, and Hamdioui S. Memory testing with a RISC microcontroller. In: 2010 Design, Automation Test in Europe Conference Exhibition (DATE 2010); 2010. p. 214–219.

[37] Di Carlo S, Prinetto P, and Savino A. Software-Based Self-Test of Set-Associative Cache Memories. IEEE Transactions on Computers. 2011;60(7):1030–1044.

[38] Paschalis A and Gizopoulos D. Effective Software-Based Self-Test Strategies for On-Line Periodic Testing of Embedded Processors. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. 2005;24(1): 88–99.

[39] Bernardi P, Cantoro R, De Luca S, *et al.* Development Flow for On-Line Core Self-Test of Automotive Microcontrollers. IEEE Transactions on Computers. 2016;65(3):744–754.

[40] Gaudesi M, Reorda MS, and Pomeranz I. On test program compaction. In: 2015 20th IEEE European Test Symposium (ETS); 2015. p. 1–6.

[41] Gaudesi M, Pomeranz I, Reorda MS, *et al.* New Techniques to Reduce the Execution Time of Functional Test Programs. IEEE Transactions on Computers. 2017;66(7):1268–1273.

[42] Riefert A, Cantoro R, Sauer M, *et al.* A Flexible Framework for the Automatic Generation of SBST Programs. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 2016;24(10):3055–3066.

[43] Sabena D, Reorda MS, and Sterpone L. On the Automatic Generation of Optimized Software-Based Self-Test Programs for VLIW Processors. IEEE Transactions on Very Large Scale Integration (VLSI) Systems. 2014;22(4):813–823.

[44] Hatzimihail M, Psarakis M, Gizopoulos D, *et al.* A methodology for detecting performance faults in microprocessors via performance monitoring hardware. In: 2007 IEEE International Test Conference; 2007. p. 1–10.

[45] Di Carlo S, Gambardella G, Indaco M, *et al.* A software-based self test of CUDA Fermi GPUs. In: 2013 18th IEEE European Test Symposium (ETS); 2013. p. 1–6.

[46] Shi W, Alawieh MB, Li X, *et al.* Algorithm and Hardware Implementation for Visual Perception System in Autonomous Vehicle: A Survey. Integration. 2017;59:148–156. Available from: http://www.sciencedirect.com/science/article/pii/S0167926017303218.

[47] Hamdioui S, Gizopoulos D, Guido G, *et al.* Reliability challenges of real-time systems in forthcoming technology nodes. In: 2013 Design, Automation Test in Europe Conference Exhibition (DATE); 2013. p. 129–134.

[48] Agbo I, Taouil M, Hamdioui S, *et al.* Read path degradation analysis in SRAM. In: 2016 21th IEEE European Test Symposium (ETS); 2016. p. 1–2.

[49] Baumann RC. Radiation-Induced Soft Errors in Advanced Semiconductor Technologies. IEEE Transactions on Device and Materials Reliability. 2005;5(3):305–316.

[50] Defour D and Petit E. A Software Scheduling Solution to Avoid Corrupted Units on GPUs. Journal of Parallel and Distributed Computing. 2016;90–91:1–8. Available from: http://www.sciencedirect.com/science/article/pii/S0743731516000022.

[51] Di Carlo S, Gambardella G, Martella I, *et al.* An improved fault mitigation strategy for CUDA Fermi GPUs. In: Dependable GPU Computing workshop, Dresden; 2014.

[52] Farazmand N, Ubal R, and Kaeli D. Statistical fault injection-based AVF analysis of a GPU architecture. In: IEEE Workshop on Silicon Errors in Logic; 2012.

[53] Bakhoda A, Yuan GL, Fung W, *et al.* Analyzing CUDA workloads using a detailed GPU simulator; 2009. p. 163–174.

[54] Du B, Condia JER, Reorda MS, *et al.* About the functional test of the GPGPU scheduler. In: 2018 IEEE 24th International Symposium on On-Line Testing And Robust System Design (IOLTS); 2018. p. 85–90.