A dynamic hardware redundancy mechanism for the in-field fault detection in cores of GPGPUs

Josie E. Rodriguez Condia[†], Pierpaolo Narducci*, M. Sonza Reorda[‡], L. Sterpone[§]
Politecnico di Torino, Torino, Italy
*pierpaolo.narducci@studenti.polito.it, {†josie.rodriguez, ‡matteo.sonzareorda,§luca.sterpone}@polito.it

Abstract—**In the past, in most General-Purpose Graphic Processing Units (GPGPUs) application fields (e.g., multimedia and gaming), the reliability features were not so relevant. Nowadays, GPGPUs are used in new domains, such as the automotive one, where reliability plays a significant role. In this work, we describe a dynamic duplication with a comparison (DDWC) mechanism intended to harden the Scalar Processor (SP) units located in the Streaming multiprocessors (SM) of a GPGPU. The proposed mechanism targets the permanent faults that may arise inside the SPs. One additional SP unit is included in the system to compute redundantly the same operations of a selected SP. Results are compared, and possible failures detected. A custom reconfiguration instruction allows the dynamic selection of the target SP to be monitored. Experimental results show that the proposed mechanism introduces a limited area overhead while it provides a significant increase in the in-field fault detection capabilities of the GPGPU. Its flexibility allows selecting the best trade-off between fault detection latency and performance overhead.**

Keywords—**Duplication with Comparison (DWC), Fault detection, General Purpose Graphics Processing Units (GPGPUs), Graphics Processors**

I. INTRODUCTION

Graphic Processing Units (GPUs) were used in the past, mainly in commodity applications involving high-data intensive operations, such as in the gaming and multimedia sectors. Currently, GPUs have successfully extended their capabilities and are employed in more complex and demanding applications, evolving into General Purpose Graphic Processing Units (GPGPUs). These applications include High-Performance Computation (HPC) and some in the safety-critical domain [1]. Modern GPGPUs are designed employing the latest technology scaling approaches to achieve power and performance requirements. However, some studies proved that these technologies could be quite prone to faults during the operative life, thus showing reduced long-term reliability [2].

Designers face reliability challenges in GPGPUs by adding special structures, such as *Error Correcting-Codes* (ECCs), to reduce the sensitivity to faults of some structures, such as memories, register files, and communication interfaces. Nevertheless, other internal modules, such as the execution cores and the task controllers, cannot be easily protected with an acceptable cost in terms of design and manufacturing. Thus, these units represent a challenge when dealing with their protection [3]. Moreover, traditional solutions to extend the reliability of control and execution units (EUs), such as dual-core lock-step and design diversity [4], increase the hardware overhead and the complexity of the device exponentially.

Traditional structures to increase the in-field reliability in processor-based systems, such as *Duplication with Comparison* (DWC), *Double,* and *Triple Modular Redundancy* (DMR, TMR), are mainly neglected in GPGPU products due to economic and technical reasons. Nevertheless, these solutions may be used in applications where safety is a major constraint. In these fields, the additional cost in terms of design and production can be reasonably accepted due to the reliability benefits. To the best of our knowledge, there are no hardware solutions for in-field operation used in real GPGPU cores based on the previous techniques.

Multiple solutions using DWC as an error detection strategy have been proposed in the literature for GPGPU devices and are based on two main approaches: software and hardware. In both cases, the main target is to provide the detection or correction of faults. However, hardware approaches may be complex to develop and implement, but these are more efficient than software approaches.

On the one hand, software DWC mechanisms exploit time redundancy by repeatedly executing instructions [5-7], functions [8], or application tasks [8-10]. At the end, results are compared to detect faults. These mechanisms introduce zero hardware overhead but cause significant performance degradation, additional switching activity, and also a moderate overhead in the memory and similar resources. Similarly, in [11], the authors proposed an automatic multithreading environment to modify the program of a GPGPU and duplicate the operations. This solution may use spatial and time redundancy, but performance degradation directly depends on the behavior of the application.

On the other hand, hardware solutions exploit spatial redundancy and consist of adding special structures in the design to increase the reliability by allocating operations into redundant and independent modules. Results are then compared to detect possible faults. The main advantage of these mechanisms is a reduced performance degradation. However, the hardware overhead directly depends on the redundant modules to be added and modified. In [12], a mechanism duplicates the fragment cores in a Shader module of a GPU. Special structures are added in the input and output modules to process the data operands and the interconnections. Nevertheless, the mechanism was only evaluated resorting to a structural simulator; hence, the hardware overhead and the resulting performance degradation were not quantitatively assessed. In [13], the warped-Dual Modular Redundancy was proposed, which uses the free execution cores inside the *Streaming Multiprocessor* (SM) to provide a redundant mechanism and detect errors. The inactive threads are configured using a *Register Forwarding Unit* (RFU), which can be configured either as EUs or as comparators. The RFU uses the original structures to monitor and verify the cores using timing redundancy. Although this method was evaluated in a simulator, results showed that the associated hardware overhead might be significant, considering the RFU existing in each core. In [14], additional execution units are included in the design of a GPGPU to replace faulty units at the end of the manufacturing phase. In [15], the authors propose and addition of heterogeneous cores to detect faults in a multi-core processor when executing out-of-order operations. Finally, another approach includes parallel execution cores aiming to repair faulty units during the in-field operation of a GPGPU [16]. The repairing method may use SBST, DfT, or a combination of both as a fault detection mechanism to identify a defective module.

In this work, we propose a dynamic redundancy mechanism targeting the detection of permanent stuck-at faults in the EUs or SPs of a GPGPU. The proposed solution is called *Dynamic Duplication with Comparison* (DDWC), and it is based on the adaptation of the classical DWC mechanism to protect structures in a GPGPU. This solution is intended to introduce small or null performance overhead and minimal modifications to the existing structures.

More in detail, the DDWC mechanism is based on the addition of one spare EU module that performs the same operations as one of the original ones and also checks the coherence of the results. A custom instruction (DDWC_i) is introduced, which identifies the EU modules to be monitored. This solution combines the flexibility provided by the software to identify the target module, combined with the low-performance degradation and spatial redundancy obtained by the hardware solutions, thus providing an effective fault detection mechanism. This technique can be employed for the in-field test by slightly modifying the application code by just adding the new DDWC_i instruction at specified locations, so apart from the DDWC_i instructions, the DDWC mechanism is entirely transparent to the programmer. Moreover, the designer can choose a suitable trade-off between fault detection latency and performance overhead by selecting the frequency of execution of the DDWC_i instruction.

The DDWC solution was implemented and evaluated, resorting to an enhanced version of the FlexGrip GPGPU model, which owns the same micro-architecture of some NVIDIA devices. Results proved that the DDWC strategy increases the fault detection capabilities of the system with a limited cost in terms of hardware and performance overhead. Moreover, since the selection of the SP core to be monitored is made in software by an ad hoc instruction, we can select the best frequency for its activation, trading-off fault detection latency, and performance degradation.

The paper is organized as follows. Firstly, section II introduces the basic overview of the enhanced open-source GPGPU model. Section III describes the proposed DDWC fault detection mechanism. Then, section IV outlines its implementation in the FlexGrip model. Finally, section V reports the gathered experimental results and estimates the detectability enhancement provided by the strategy. Section VI draws some conclusions and highlights future works.

II. THE FLEXGRIP GPGPU MODEL

FlexGrip is an open-source soft-GPGPU model described in VHDL and aimed initially to be synthesized in Xilinx FGPA platforms [17]. The University of Massachusetts developed this GPGPU model, and it implements the G80 GPGPU micro-architecture by NVIDIA. Moreover, it is compatible with the

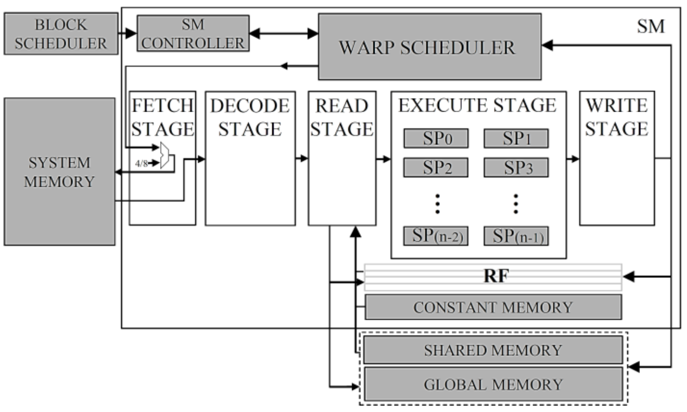

Fig 1. The general architecture of the SM in FlexGrip (adapted from [19])

standard programming environment for GPGPUs (CUDA in the SM_1.0 compatibility). This special-purpose processor requires the interaction with a Host processor to start the execution of an application program. This Host controls and monitors the GPGPU operation.

We improved the original version of the FlexGrip model to remove or correct initial limitations related to compiler restrictions, structural bugs, and instruction format support. The enhanced GPGPU model now fully supports 28 instructions in 64 formats. For this work, we employ this improved version of the model. Readers may refer to [18] for a full list of supported formats and instructions in FlexGrip and additional details of the performed modifications.

The architecture of FlexGrip supports the SIMT (*Single-Instruction Multiple-Thread)* paradigm. It exploits a custom SM core composed of five stages pipeline (*Fetch*, *Decode*, *Read*, *Execution/Control-flow,* and *Write-back*), as shown in Fig 1. The EUs or *Scalar Processors* (SPs) are only present in the *Execution/Control-flow* pipeline stage. The SM executes the same instruction (warp instruction) for a group of 32 threads. A warp can be defined as a group of 32 consecutive threads.

Inside the SM, one warp scheduler controller manages the execution of instructions and controls the operation of threads. In the SIMT taxonomy, each warp instruction is fetched, decoded, and assigned for execution to an independent SP in the SM. The *Read* and *Write-back* stages load and store data operands from/to the *Register File* (RFs, or the shared, global, and constant memories. It is worth noting that FlexGrip can only process operations of integer type. The original design of the G80 architecture includes 8 SP cores in the SM. FlexGrip can configure the parallel EUs among 8, 16, and 32 SPs.

The SPs are regular structures within the SM of a GPGPU and are composed of multiple sub-modules (Fig. 2). Each SP core executes all the signed and unsigned logic and arithmetic operations required by a thread task. A thread task can be defined as the sequence of operations to be executed by one of the threads in a parallel program.

The input operands of an SP core are organized in data channels (SPCs). The SPCs are composed of 32 bit-size input data operands (SRC1, SRC2, and SRC3) and the predicate flags (4 bit-size). Each SP core has an independent and statically assigned SPC by a warp scheduler in the SM.

In the *Read* stage, the input operands are loaded from memory and sent to each SPC. External control signals redundantly configure each SP core according to the instruction to be executed. The output data channel (SPDC) of each SP core is composed of the result data (*DST*) and the output predicate registers. These values are statically connected to the next pipeline stage.

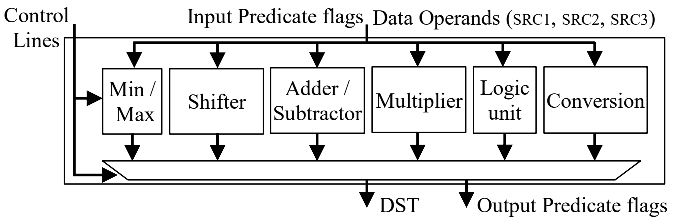

Fig 2. A general scheme of the architecture of an SP core in FlexGrip

III. PROPOSED FAULT DETECTION TECHNIQUE

The proposed in-field fault detection strategy (DDWC) is based on a dynamic mechanism able to detect permanent faults in modules of processor-based systems or accelerators, such as a GPGPU. DDWC is an adaptation of the classical DWC structure and employs the concept of sphere of redundancy (Fig 3), which is based on replicating one or more modules to increase the fault detection or to mitigate the fault effect in a system.

The DDWC structure is mainly composed of an input selector module, a redundant module, which is a copy of the target module, an output selector module, and a comparator. Finally, one controller manages the input and output selectors to provide the data and feed the redundant and comparator modules with a selected option in the controller. The controller is dynamically programmed through a custom instruction that activates and selects the input data channel of one of the modules in the sphere of redundancy. The input and output selector modules are composed of crossbar or meta-crossbar structures. Some additional decoders and registers could be included in the DDWC structure.

It should be noted that the sphere of redundancy can be applied at multiple levels of granularity in a system. However, the DDWC structure could be unfeasible to be applied to big modules, which might represent a considerable hardware overhead for a system, such as pipeline blocks in a processor or a GPGPU. Moreover, the non-regularity of some structure could limit the adaptation of the DDWC to keep low performance and hardware costs. It is worth noting that this work focuses on the fault detection mechanism, so error handling is out of the scope of this paper, and additional mechanisms might be required to perform the error handling tasks.

A. Basic operation

Once the DDWC module is active, the controller selects one input data channel, and all input operations are redundantly executed by both: the target module and the redundant. This comparison and fault detection structure is well-known and contributes to reducing the performance latency for fault detection. Then, the results of both modules (*the target module and the redundant one*) are compared. The fault detection is performed based on a direct comparison of results and output flags. A fault is detected if there is at least one mismatch in the results. The comparator generates an output error signal triggering the logic state that indicates the possible fault detection.

B. Proposed Architecture for the SPs in a GPGPU

The DDWC structure is intended to detect permanent faults during the in-field operation of the GPGPU.

Initially, we explore multiple modules in the GPGPU to implement the DDWC strategy. Firstly, the entire *Execute* pipeline was considered as a target module in the sphere of redundancy. However, as commented before, the non-regularity of its internal structures limited the adaptation of the fault detection technique, so causing unacceptable hardware overhead in a GPGPU. Then, we considered the possibility of focusing on the internal structures in the SPs in the SM. In this case, each sub-module should be duplicated, reaching a similar overhead. On the other hand, by exploiting the regularity of the SPs, and using them as the main target of the sphere of redundancy, it is possible to duplicate these structures temporarily. A combination between the regularity of the SPs and the reconfiguration mechanism contributes to reducing the total overhead costs in the system. In the end, we propose the addition of only one SP, and a newly introduced custom instruction allows selecting a target SP with the comparison structure. Fig 4 summarizes a general scheme of the mechanism using SP cores as the main modules in the sphere of redundancy.

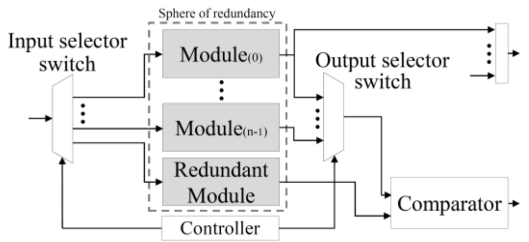

Fig 3. A general scheme of the concept of sphere of redundancy applied to the DDWC strategy

The input switch collects all the SPCs and activates one of the data paths coming from the previous pipeline stage and feeds the redundant SP core. In contrast, the output switch collects all SPDCs for the active SPs, selects one among them, and feeds the comparator module. A switch controller is included to manage the channel selection in both switches (input and output). The signals decoded by the DDWC_i instruction are employed to reconfigure both switching structures.

A redundant SP is placed in parallel to the existing SP modules, and the inputs are directly connected to the input switching selector. Similarly, the outputs of the redundant SP are connected to one of the inputs of the comparator module.

The comparator module is a bit-wise comparator of the output results (SPDCs) coming from a target SP and the redundant SP. An output strobe flag is included as observability mechanism to inform the Host, or an exception handler in the GPGPU, about a possible fault found in one of the SPs. A mismatch in the results indicates a fault in the SP, hence triggering the output flag.

Some additional combinational modules, such as decoding hardware and interconnections, are included in the DDWC mechanism. Similarly, a register retains the configuration of the DDWC structure after using the DDWC_i instruction. It is worth noting that the DDWC mechanism starts in a disable mode after a reset or switch-off event and should be activated with the DDWC_i instruction. The explicit selection of the target SP core using the DDWC_i instruction may affect the optimal fault detection performance of all elements in the sphere of redundancy. Moreover, input operations and the frequency of the custom instruction can also limit the number of patterns applied to each core to identify errors. Thus, the DDWC mechanism requires a balance between the selection of the target core and the insertion of the DDWC_i instruction in the application code to obtain the best fault detection performance.

It is worth noting that other modules in the GPGPU, such as memories and control units, were not targeted and are out of the scope of the proposed DDWC solution. The use of the DDWC mechanism in these modules may introduce hardware overheads greater than 100%, considering the duplication of modules and the required additional structures, such as registers, comparators, and interconnections.

IV. Implementation details

The enhanced version of FlexGrip was used to implement and evaluate the fault detection capabilities of the proposed DDWC strategy experimentally. Three pipelines modules (*Decode*, *Read,* and *Execution/Control-flow*) were modified to include the DDWC structures.

In the *Decode* stage, the DDWC_i instruction was implemented by carefully adding some combinational logic. The instruction set architecture (ISA) in FlexGrip (SASS) was analyzed, and the available operation code was selected for its description. The format of the DDWC_i instruction includes 5 bits for SPC channel and SP selection, 2 bits to enable or disable the DDWC module, and 6 bits stating instruction type.

In the *Read* stage, a bypass structure was included. This bypass is composed of registers and keeps the pipeline coherence during the execution of the instruction by managing the operands and configuration DDWC signals to be used in the *Execution/Control-flow* stage.

The *Execution/Control-flow* stage was modified, adding a copy of an SP core (SPx), the input and output multiplexers (implementing the input and output selector switches), and a logic comparator (Fig 4).

The two multiplexers are placed in the inputs and outputs of the SP core (SPx) to select the data channel from each thread (SPC (in) and SPDC).

The comparator is built with XOR gates. The output multiplexer selects and feeds one of the inputs of the comparator with results in one SPDC coming from SP cores. The other input is connected to the outputs of the redundant core (SPx). An output flag in the comparator is propagated to the next stage and then used to indicate the Host that an error was detected. This flag is intended to activate an interrupt in the host and indicate the presence of a fault in the SP core.

An additional controller, one decoder, and some registers were also included to configure both multiplexers. As previously mentioned, this configuration is based on the decoding signals from the DDWC_i instruction.

The data-path interconnections for the input (SPC) and output (SPDC) data channels were duplicated to feed the multiplexer units. On the other hand, the control-path fields in SPC were not considered as these are shared among the SPs and SPx. The controller also propagates the configuration code to the Host. This code can be used to locate a faulty SP core in the SM by the Host.

Under the inactive mode of the DDWC structure, both multiplexers are unconnected without feeding the SPx core and the comparator to avoid unnecessary switching activity in both modules, thus reducing the dynamic power during inactivity periods. It is worth noting that the selection of a sphere of redundancy targeting the SPs only for implementing the DDWC, contributed to avoiding any modification in the original memory hierarchy and the warp scheduling of the GPGPU.

In principle, the code of any application requires a minor modification to use the DDWC mechanism for in-field detection. It is suggested to include the DDWC_i instruction at the beginning of the application code to reduce performance degradation. Similarly, the DDWC_i instruction can be placed in strategic locations in the code (i.e., before any logic-

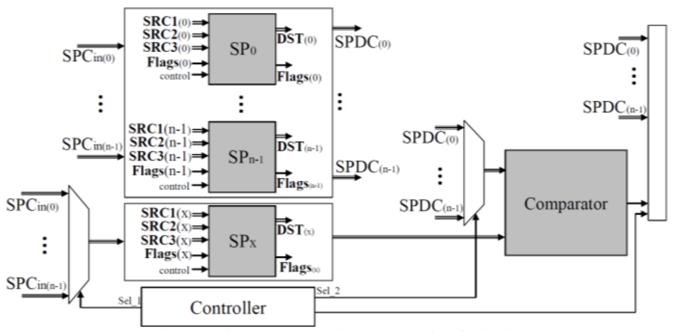

Fig 4. A general scheme of the proposed structure for fault detection

arithmetic instruction) or in a periodical manner (after a given number of instructions). However, in both cases, the performance degradation for the application must be considered. In any case, during the in-field execution, the output flag signal in the comparator is directly assigned as a source of fault identification.

It is worth noting that the DDWC structure is intended to be used during the in-field operation of the GPGPU. Once the DDWC structure is active, the redundant core should swap among the SPs of the SM. For this purpose, the program code is modified by adding some DDWC_i instructions. Depending on the application and on the selected frequency for the swap, the locations where the DDWC_i instructions are inserted can be suitably selected. A higher rate reduces the fault detection latency but increases the performance overhead stemming from the DDWC_i instruction addition. The method is flexible since it allows adopting the solution which best fits the target system specifications.

V. Experimental Results

In the experiments, all possible SP core configurations (8, 16, and 32 SP cores) of the FlexGrip model were considered. This model flexibility allows us to analyze the benefits of the proposed structures under multiple SP core configurations.

The applications executed by the GPGPU with 8 and 16 SP core configurations employ the DDWC mechanism four times and twice per SP core, respectively. The explanation for this behavior is directly related to the thread management in the SM under these SP configurations. The warp controller, of an SM, configured with 8 or 16 SPs, distributes the thread-tasks into the available SPs. Thus, for an 8 SPs configuration, four thread-tasks share the same SP. Similarly, for a 16 SPs configuration, the controller assigns two threads to each SP. Thus, for every configuration of the DDWC structure, the comparison, and possible fault detection are performed twice or four times with instructions belonging to the same warp.

The experimental performance results were found resorting to a gate-level version of the modified FlexGrip model. The Synopsis toolchain (*Design Vision*) was used to estimate the hardware overhead and performance degradation. The NAND-Open-cell library [19] was used for synthesis purposes.

A. Hardware overhead

Both GPGPU models (including the DDWC structure and the original one) were compared for all configurations of SP cores. Results in terms of size of cells are reported for the affected modules and the whole design in Table 1. Results do not include the cost of cells in memories and RFs in FlexGrip.

Results show that the hardware overhead is relatively low. In the *Decode* stage module, this overhead represents only 3% and seems to be insignificant ($\approx 0.1\%$) in the *Read* stage module. In contrast, in the *Execution/Control-flow* pipeline module, the overhead is inversely proportional to the number of

SPs in the SM. Hence, a lower number of SPs introduce a higher overhead cost. However, this overhead seems to be moderate ($\approx$3.5-10%). Analyzing the overhead cost in the whole design, this is lower than 3% for all SPs configurations. These results support the initial intention of limiting the impact in the hardware overhead by the DDWC structures.

The above results allow us to claim that the proposed DDWC mechanism represents a practical solution as a fault detection strategy without including a critical impact in the area of the GPGPU.

TABLE 1. HARDWARE AND PERFORMANCE OVERHEAD OF THE DDWC MECHANISM FOR MULTIPLE CONFIGURATIONS IN THE GPGPU MODEL

Modules	SP cores	Number of Cells		Area overhead (%)	Time delay in the critical path (pS)		Performance degradation (%)
		FlexGrip	FlexGrip + DDWC		FlexGrip	FlexGrip + DDWC	
Decode	8/16/32	1,229	1,266	3.04	1.72	1.77	1.16
Read	8/16/32	142,397	142,545	0.10	3.65	3.65	0
Execute	8	60,309	65,959	9.37	6.51	7	7.52
	16	113,293	118,739	4.81	6.69	7.68	14.79
	32	219,261	226,822	3.45	7.52	8.54	13.56
All	8	229,515	235,964	2.81	11.88	12.42	4.54
	16	280,132	286,360	2.22	12.06	13.1	8.62
	32	386,100	394,516	2.18	12.89	13.96	8.3

B. *Performance overhead*

The performance overhead is evaluated at the module and design levels. Initially, the critical path delay was determined for each modified module and the whole design. It is worth noting that in both cases, the synthesis and analysis were performed without adding constraints or optimizations in the synthesis tool. A clock period of 10 ns was selected for the synthesis. The results are presented in Table 1.

By looking at the results, it can be noted that the additional structures in the *Decode* module added a small percentage of performance degradation (1.16%). In the *Read* module, a bypass register was added. Nevertheless, this does not introduce any overhead. This behavior can be explained by observing that the bypass register was concatenated with existing structures in the module, thus guaranteeing the same functionality of the original structures. In contrast, the performance degradation in the *Execution/Control-flow* module seems to be directly affected by the number of SP cores. For a low number of SP cores, the critical delay path is increased by 7.52%. The configuration of 16 SP cores seems to present the maximum percentage of performance overhead with 14.79%. In contrast, the overhead drops for the 32 SP cores. Although the total delay overhead is higher for 32 SPs, the delay overhead is lower. This behavior can be explained considering that added SPs are placed parallel in the design, thus adding a low timing overhead. However, the most representative timing effects are due to the added modules in the path. These modules are the input and output switches and the comparator.

C. *Fault detection capabilities*

A set of benchmarks with different workloads was employed to validate and evaluate the detection capabilities of the DDWC strategy experimentally. The fault injection campaigns were performed in the FlexGrip model with and without the DDWC mechanism and considering single stuck-at faults at the RT level. On each fault campaign, the GPGPU model was configured with 8 SPs and 32 threads per block. Each benchmark was adapted to include the DDWC_i instruction activating the DDWC structure. Gathered results were combined to mimic the long-term operation of the GPGPU with the DDWC mechanism rotating among the SPs.

The fault injection environment is based on the ModelSim framework, and the injection methodology we used is the same introduced in [5, 20]. Further details regarding the descriptions and configurations of the used benchmarks can be found in [18]. The output flag from the comparator was included as an observability mechanism to detect faults in the simulation environment.

Results from the experiments showed a 35% average increment in fault detection capabilities. The final Fault Coverage that can be achieved on each SP resorting to the proposed method strongly depends on the application. Moreover, it should be considered that the DDWC strategy requires the explicit selection of a target SP core to perform fault detection. Thus, a balance between the frequency of the SP switching and the application features is required to obtain optimal in-field fault detection.

D. *Estimation of the fault detection time*

The proposed DDWC strategy increases the fault detection capabilities concerning faults in the target structures. However, as the DDWC structure is intended to select the SP to be monitored during in-field execution dynamically, the overall fault detection capabilities of the DDWC strategy depends on several parameters, such as the switching and detection time, and the detection capabilities of the test patterns. A test pattern is one or a sequence of values applied to the inputs of the target SP to excite a fault and propagate the error to the outputs.

Considering a fault-free DDWC structure and an SM composed of *n* SPs, the SM in principle cannot detect permanent faults in the SPs. Hence, fault detection capabilities are zero. Using the DDWC mechanism, the fault detection capabilities in the system increase. This increase can be estimated by resorting to the relation between the fault observability and the required time to detect a fault. The fault observability (Ob) [21] in one SP can be defined as:

$$Ob_{SP(p)} = \frac{N_p}{N_p + N_{np}} \tag{1}$$

Where N_p and N_{np} are the numbers of input patterns that propagate and do not propagate a fault effect to the output, respectively, and $P=N_p+N_{np}$ is the number of patterns, assuming that the average observability remains constant across the time. In the DDWC mechanism, the patterns are mainly generated by the execution of instructions and data operands.

The time for detecting a fault (*tt*) is composed of a set of time intervals needed to perform the fault detection in an SP. If a fault arises in the system at time $t=0$, the fault is excited by a pattern after a time t_1. Then, it is propagated to the output after a time t_2, and finally, it is detected after a time t_3. Thus, it is possible to estimate the time for fault detection (ETFD) in a continuous fault detection structure case as:

$$ETFD_{DWC(t,p)} = \frac{(N_p + N_{np}) \cdot tt}{N_p}; tt = t_1 + t_2 + t_3 \tag{2}$$

In the expression (2), short times are desirable to perform fault detection. However, the configuration of the DDWC strategy directly depends on the time interval employed to switch among SP cores t_4 and the time required to execute the configuration instruction t_5. In this way, it is possible to express the time for fault detection (ETFD) in the SPs using the DDWC mechanisms as:

$$ETFD_{GPGPU(tt,p)} = \frac{(N_p + N_{np}) \cdot (tt + \sum_{i=1}^{n}(t_4 + t_5))}{N_p} \tag{3}$$

Where n is the number of SPs in the SM, and time t_5 is proportional to the number of instructions between two sequential configuration instructions.

Figure 5 represents the worst-case scenario for fault detection in an SP, assuming that the switching among the available SPs is performed every 100 instructions in the program code. The expression is calculated using a clock period of 10 ns, an average time for instruction execution of 20 clock cycles, and times for pattern injection, fault propagation and fault detection at a fixed rate of 3 clock cycles. It is assumed that the fault can be propagated to the output by one of the test patterns.

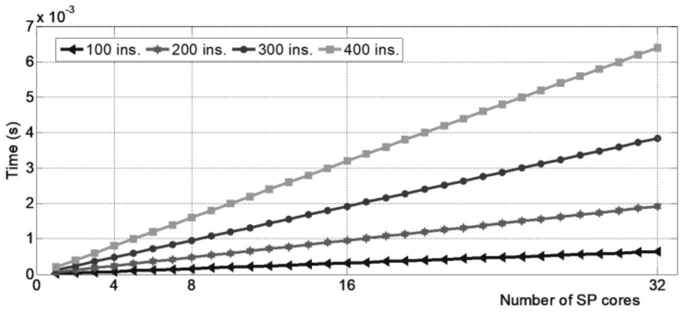

Fig 5. The maximum value of ETFD for multiple frequencies of the DDWC_i instruction (100, 200, 300, and 400 instructions) in the application code and under multiple SPs configurations in the SM

The expression in (3) can be used to find an optimal trade-off between switching frequency for the DDWC mechanism among the SPs and the performance degradation in the application by the insertion of DDWC_i instructions.

E. Comparison with other fault detection strategies

A comparison of the proposed mechanism with classical fault detection methods, such as lock-step and Build-In Self-Test (BIST), shows that the lock-step structure requires the duplication of each structure in the design. Thus, the hardware overhead overcomes 100%, considering the additional structures. On the other hand, a BIST mechanism can be an effective solution for the end-of-production test. Nevertheless, the same approach could be challenging to use to the SPs during the operational phase of a GPGPU. These difficulties are based on the required execution time to perform fault detection.

Moreover, the required structures for implementing such a solution may be equal to or greater than the DDWC strategy in terms of hardware overhead. Other fault detection solutions based on Software-Based Self-Test (SBST) mechanisms add zero cost to the system. However, the time required performing the test pattern injection, and fault detection are higher in comparison with the time in the DDWC mechanism. Finally, the DDWC could be combined with an SBST technique to increase the fault observability.

The previous analysis allows us to claim that the DDWC mechanism is more convenient for in-field fault detection than lock-step and BIST by the lower cost in terms of hardware. Moreover, the proposed mechanism may coexist with other structures to detect, mitigate, or repair faults in the SP cores.

VI. Conclusions

A dynamic fault detection strategy (DDWC) was described and evaluated, targeting the in-field detection of permanent faults in the execution units (SPs) in SMs of a GPGPU. DDWC is based on the duplication with comparison strategy and exploits the structural regularity of the SP cores. The SP core to be monitored can be dynamically selected, resorting to one ad hoc instruction. Thanks to its flexibility, low hardware overhead, and moderate performance degradation, this strategy could be effectively employed to increase the reliability of GPGPUs when they are adopted in safety-critical applications.

As future works, we plan to develop and evaluate dynamic fault detection mechanisms for other critical modules in the GPGPU, such as the scheduling controllers.

Acknowledgments

The European Commission has partially supported this work through the Horizon 2020 RESCUE-ETN project under grant 722325.

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.

[2] 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.

[3] S. Alcaide, L. Kosmidis, C. Hernandez, and J. Abella, "High-Integrity GPU Designs for Critical Real-Time Automotive Systems," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2019, pp. 824-829.

[4] NVIDIA. (2016). *NVIDIA Announces World's First Functionally SafeAI Self-Driving Platform*. Available: https://nvidianews.nvidia.com/news/nvidia-announces-worlds-first-functionally-safe-ai-self-driving-platform.

[5] M. Gonçalves, M. Saquetti, F. Kastensmidt, and J. R. Azambuja, "A low-level software-based fault tolerance approach to detect SEUs in GPUs' register files," *Microelectronics Reliability,* vol. 76-77, pp. 665-669, 2017.

[6] M. Gonçalves, M. Saquetti, and J. R. Azambuja, "Evaluating the reliability of a GPU pipeline to SEU and the impacts of software-based and hardware-based fault tolerance techniques," *Microelectronics Reliability,* vol. 88, pp. 931-935, 2018.

[7] M. Goncalves, F. Fernandes, I. Lamb, P. Rech, and J. R. Azambuja, "Selective Fault Tolerance for Register Files of Graphics Processing Units," *IEEE Transactions on Nuclear Science,* 2019.

[8] D. A. G. Oliveira, *et al.*, "Modern GPUs Radiation Sensitivity Evaluation and Mitigation Through Duplication With Comparison," *IEEE Transactions on Nuclear Science,* vol. 61, pp. 3115-3122, 2014.

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

[10] A. Mahmoud, S. K. S. Hari, M. B. Sullivan, T. Tsai, and S. W. Keckler, "Optimizing software-directed instruction replication for GPU error detection," in *International Conference for High Performance Computing, Networking, Storage, and Analysis*, Dallas, Texas, 2018.

[11] J. Wadden, A. Lyashevsky, S. Gurumurthi, V. Sridharan, and K. Skadron, "Real-world design and evaluation of compiler-managed GPU redundant multithreading," *ACM SIGARCH Computer Architecture News,* vol. 42, pp. 73-84, 2014.

[12] J. W. Sheaffer, D. P. Luebke, and K. Skadron, "A hardware redundancy and recovery mechanism for reliable scientific computation on graphics processors," in *Graphics Hardware*, 2007, pp. 55-64.

[13] H. Jeon and M. Annavaram, "Warped-DMR: Light-weight Error Detection for GPGPU," in *2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, 2012, pp. 37-47.

[14] J. R. Nickolls, "EFECT TOLERANT REDUNDANCY," NVIDIA Corporation, 2009.

[15] S. Ainsworth and T. M. Jones, "Parallel Error Detection Using Heterogeneous Cores," in *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 338-349.

[16] J. E. R. Condia, P. Narducci, M. Sonza Reorda, and L. Sterpone, "A dynamic reconfiguration mechanism to increase the reliability of GPGPUs," in *VTS 2020: VLSI Test Symposium*, San Diego, USA, 2020.

[17] 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.

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

[19] J. Knudsen, "Nangate45nm Open Cell Library," *CDNLive, EMEA,* 2008.

[20] J. E. R. Condia and M. Sonza Reorda, "Testing permanent faults in pipeline registers of GPGPUs: A multi-kernel approach," in *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2019, pp. 97-102.

[21] S. Hurst, "VLSI testing and testability considerations: an overview," *Microelectronics Journal,* vol. 19, pp. 57-69, 1988.