Functional and Timing Implications of Transient Faults in Critical Systems

Angeliki Kritikakou*, Panagiota Nikolaou ¶‖, Ivan Rodriguez-Ferrandez†‡, Joseph Paturel*,
Leonidas Kosmidis†‡, Maria K. Michael¶‖, Olivier Sentieys*, David Steenari§

*Univ Rennes 1, INRIA, Irisa, France
¶Department of Electrical and Computer Engineering, University of Cyprus, Nicosia, Cyprus
‖KIOS Research and Innovation Center of Excellence, University of Cyprus, Nicosia, Cyprus
†Universitat Politècnica de Catalunya (UPC), Barcelona, Spain
‡Barcelona Supercomputing Center (BSC), Barcelona, Spain
§European Space Agency (ESA), Noordwijk, The Netherlands

***Abstract*—Embedded systems in critical domains, such as automotive, aviation, space domains, are often required to guarantee both functional and temporal correctness. Considering transient faults, fault analysis and mitigation approaches are implemented at various levels of the system design, in order to maintain the functional correctness. However, transient faults and their mitigation methods have a timing impact, which can affect the temporal correctness of the system. In this work, we expose the functional and the timing implications of transient faults for critical systems. More precisely, we initially highlight the timing effect of transient faults occurring in the combinational and sequential logic of a processor. Furthermore, we propose a full stack vulnerability analysis that drives the design of selective hardware-based mitigation for real-time applications. Last, we study the timing impact of software-based reliability mitigation methods applied in a COTS GPU, using a fault tolerant middleware.**

***Index Terms*—Transient faults, Real-time systems, Critical Systems, Vulnerability analysis, Fault tolerance**

I. Introduction

Embedded systems in critical domains, such as automotive, aviation, space domains, are often required to guarantee both functional and temporal correctness. Functional correctness is provided by a meticulous design and verification process, which guarantees the high assurance of these systems. Temporal correctness is based on estimations of the Worst-Case-Execution-Time (WCET) and timing guarantees, which are provided when the application's worst-case response time is less than the deadlines and/or the total execution does not exceed a given latency requirement [46].

However, electronic systems are susceptible to faults due to their nature. The system is threatened by several phenomena, such as manufacturing process variation, aging and soft errors, that can lead to several permanent or temporary faults, occurring during execution [44]. Especially due to the reduced transistor sizes and lower supply voltages of modern technologies [13], [21], systems are becoming more and more sensitive to environmental sources [38], such as ionization, radiation, and high-energy electromagnetic interference, leading to temporary reliability violations, called transient faults, which can affect the system execution. Transient faults due to their challenging nature are considered as one of the most important reliability threats [31], [38]. Transient faults can affect the system behavior in several ways. The energy, transferred by a particle, can corrupt the state of sequential logic, e.g., by flipping the bit contained in a memory cell or a flip-flop. The result is a single-bit error, known as Single-Event-Upset (SEU). On top, combinational logic can be also affected, as the transferred energy by the particle can create a current or voltage transient, known as Single-Event-Transient (SET). The SET is propagated in the forward cone of the impacted combinational cell and it can be eventually latched by sequential logic [43], modifying one or several bits, potentially leading to multiple-bit errors, known as Multi-Bit-Upset (MBU).

In critical systems, applications with approximate processing characteristics, such as computer vision, image, video, speech and other types of signal processing, are often used. These applications typically produce outputs, computed based on dynamic input and feedback, and approximate computational kernels. The unpredictable input data streams can affect, in many cases, the quality of the application's output. Transient faults can exacerbate this problem, despite the fact that some of the faults (but not all) can be tolerated, due to the approximate nature of these applications. However, some faults can still cause catastrophic events if they affect specific parts of the approximate circuit, by causing Silent Data Corruption (SDC) errors or application crashes.

Overall, the exploration of how transient faults, along with corresponding mitigation methods, impact the application output, quality and time, is inevitable for critical systems.

Contributions: In this work, we expose the functional and timing implications of transient faults and mitigation methods applied at different system levels for critical systems. In particular, after examining the related work in Section II, in Section III we expose the timing effect of transient faults, occurring in the combinational and sequential logic of a processor. Section IV presents an emulation-based vulnerability analysis that drives selective hardware-based reliability mitigation for real-time approximate applications. Last, in

Section V we study the timing impact of software-based reliability mitigation methods applied in a COTS GPU, using a fault-tolerant middleware.

II. Related work

Vulnerability analysis and fault mitigation approaches can be applied at different system levels in order to maintain the functional correctness of the software. However, transient faults and mitigation methods have a timing impact, which can affect the temporal correctness of the system.

Regarding vulnerability estimation approaches, existing methods currently focus only on estimating the functional correctness, i.e., functional interruptions and erroneous values of the system under study. To achieve that, fault injection is applied through simulation at software or hardware level, and emulation, speeding-up the fault injection time. Software fault injection is hardware agnostic and can flip bits only in the application data structures [25], [30], [32], [51]. To improve accuracy, vulnerability analysis approaches have to consider the hardware details. They typically inject single-bit faults at the circuit sequential logic [5], [36]. Few hardware approaches considering also the circuit combinational logic, through single-bit and multiple-bit faults [9], [34]. However, not only the functional behaviour, but also the timing behaviour must be taken into account during vulnerability analysis for critical systems. Few recent studies, in the domain of iterative methods, explore such impact through software fault injection, focusing on average performance [25], [30], [32]. To provide accurate timing vulnerability analysis, hardware fault injection in sequential and combinational logic, considering single-bit and multiple-bit faults, is required.

Note that, fault injection is a time consuming procedure, especially in the case where the analysis spans different system layers. Therefore, emulation-based fault injection techniques are used to speed up fault injection campaigns and to provide acceptable observability and controllability. Several works use emulation-based techniques to analyze the effects of faults [14], [26], [37], [49]. Although these works manage to provide valuable information for the whole system, they lack the capability of providing analysis at finer granularity. The works in [7], [8], [11] provide both coarse and fine-grained reliability analysis using a fault injection emulation-based framework and show that the severity of SEUs can change at different abstraction layers. However, they do not incorporate this analysis to design fault-tolerant solutions, able to decrease the area overheads.

Several approaches exist for fault detection and mitigation at various system layers. Typical software approaches for real-time and autonomous systems are based on task replication [4], [6], [22] and check-pointing/re-execution [2], [19]. Their timing impact is taken into account in order to provide timing guarantees through schedulability analysis [10], [33], [47]. Other approaches targeting the automotive domain use diverse redundancy in the form of dual-lockstep execution potentially combined with check-pointing [18] or exploiting the intrinsic redundancy available in hardware platforms [3], [45]. Other real-time solutions based on hardware redundancy focus on faults in memories, to maintain the initial timing characteristics of hardware, despite the presence of faults, e.g., by using cache redundant entries [1]. Applying such techniques, especially at the system level, can lead to prohibitive overheads in terms of area, complexity, power, etc. However, some components can be inherently tolerant to errors, and thus, may not require protection [42]. Therefore, selective hardware redundancy-based protection can significantly reduce area and power overheads with minimal impact on system's reliability [15], [16], [35], [42]. To avoid costly and unnecessary use of redundancy mechanisms, the reliability of individual components, along with the quality of the application's output, must be accurately assessed first. Consequently, selective hardening can be applied to only critical components leading to cost-effective mitigation solutions.

Last, but not least, modern processors and systems-on-chip (SoCs) have special features to report the presence of SEUs and/or their location in the hardware, as well as whether the error was corrected or not. For example, x86 based processors use the *machine check architecture (MCA)* in order to provide this information [52]. This mechanism is implemented as a set of a specific registers which encode this information, and various software utilities which can be used to read this information. ARM-based architecture have recently obtained similar functionality, which is called *Reliability, Availability, and Serviceability (RAS)* [40]. This feature is also implemented using a combination of architecture specific registers as well as vendor specific registers. Similarly, peripheral devices incorporate error reporting functionalities for hardware failures. Storage devices such as hard drives and solid state drives (SSDs) use the *Self-Monitoring, Analysis and Reporting Technology (SMART)* feature in order to report recovered errors in these devices and health indications which can signify device failures in future. The information provided from such error reporting mechanism can be integrated in a fault tolerant middleware in order to take proper mitigation actions. Furthermore, the timing impact of such actions should be analysed for critical systems.

III. Exposing the timing impact of transient faults

Due to technology size reduction, faults occurring in both combinational logic and smaller sequential logic inside the processor cannot be considered negligible anymore [28], [43]. Such faults can significantly affect the execution time of a task. In critical systems, such an increase can impact the WCET estimation, compared to the fault-free one. In this section, we expose the following key aspect: the presence of transient faults in processors does not impact only the functional correctness of an application, but it also has a significant impact on the application's execution time. To show that, we perform a vulnerability analysis for an open-source RISC-V processor, considering both the functional correctness and the timing correctness of applications, when executed on a processor, under the presence of single and multiple transient faults, occurring in both the sequential logic and the combinational logic of the processor.

A. Functional and timing vulnerability analysis methodology

Our aim is to analyse the impact of hardware faults on the functional and timing correctness of applications. To perform a realistic analysis, we consider transient faults that can lead to single-bit and multiple-bit errors. By flipping the bits stored in the sequential logic of the processor (e.g., pipeline registers and the register file), we model SEU. By inserting pulses to its combinational logic (e.g., Fetch, Decode, ALU, multiple-cycle operators, multiplexers, Forwarding Unit, etc.), we model SET, which can be propagated and latched as MBUs. Our vulnerability analysis is based on fault injection, applied at two layers.

The gate-level analysis characterizes the SEU occurrence and the SET propagation, under latching window masking of sequential logic and logical masking of combinational logic, based on the processor clock, the size, delay and type of combinational and sequential cells, taking into account the processor technology. Fault injection is performed per each pipeline stage using a single-cycle simulation. The gate-level netlist is modified by inserting an injection block at the output of each cell of the netlist, which flips the output of the cell for a given time period. The inputs of the pipeline stage are instructions randomly generated from the instruction set of the processor, with random operands. For each such input, a fault-free cycle is executed to obtain the fault-free output. The selection of the cell to inject the fault is driven by the area of the cells. If the selected cell is sequential, the fault is injected directly to the pipeline register, and thus, a single-bit error occurs. If the selected cell is combinational, an SET is inserted to the netlist. The time offset for SET injection is randomly chosen within a clock cycle. Its duration is chosen from the results obtained by physical simulation tools based on the processor target technology [20]. Then, the SET is injected and the output is latched by the register. If the injected fault survived both logical and window latching masking, it led to a single-bit or multiple-bit error. The number and the position of faulty register bits are logged.

The microarchitectural-level analysis characterizes the impact of single-bit and multiple-bit errors, obtained from the gate-level analysis, on the functional and timing correctness of the application, under microarchitecture and application structure masking. A cycle-accurate bit-accurate simulator of the processor is used for fault injection. The simulator injects faults to the processor registers, while the application runs. The cycle to inject the faults is chosen randomly between the first cycle and the total number of cycles needed for the fault-free execution. The location, where the faults are injected, is driven by the size of the combinational and sequential logic of the processor. The larger the area, the higher its probability to be selected. The type of the fault to be injected in the register (i.e., single-bit or multiple-bit and which register bits are affected) is provided by the gate-level results. The more times a specific error has appeared during gate-level analysis, the higher is its probability to be injected, during the microarchitecture-level analysis. After the fault injection and upon application termination, the results are compared to the set of golden references, in order the impact of faults on the application to be categorised as:

- *Execution Cycles Mismatch (ECM):* The execution cycles are different than the fault-free execution cycles.
- *Hang (H):* The application has entered an infinite loop.
- *Crash (C):* The execution of the application has terminated unexpectedly.
- *Application Output Mismatch (AOM):* The application output is different than that of the golden reference.
- *Internal State Mismatch (ISM):* The system state (memory and registers) are different than those of the golden reference.
- *Functionally Masked (FM):* The application has finished execution, with no AOM and no ISM.

B. Experimental results

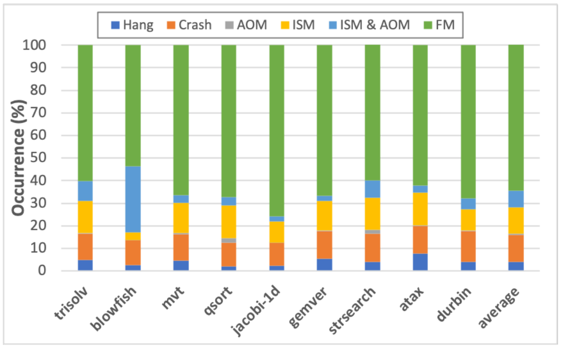
Our case study is an open-source 32-bit 5-stage pipeline RISC-V processor [41]. The processor is synthesized using Synopsys Design Compiler with a target frequency of 500 MHz. The target technology library is 28nm FDSOI from ST-Microelectronics using a supply voltage of 1.0V. The design kit cells are analyzed using MUSCA [20], considering neutron injections with an LET equal to 58MeV/cm. The peak of the SET distribution is used as SET duration in our experiments, i.e., 400 ps. Nine benchmarks from MiBench and PolyBench are analyzed, with a statistical fault injection (99.8% confidence interval, 5% error margin) [50].

Table I provides the minimum and maximum number of concurrent erroneous bits and error size with the highest occurrence per pipeline stage. These results confirm the importance to consider multiple-bit upset (MBU) when analyzing the vulnerability of a processor to transient faults.

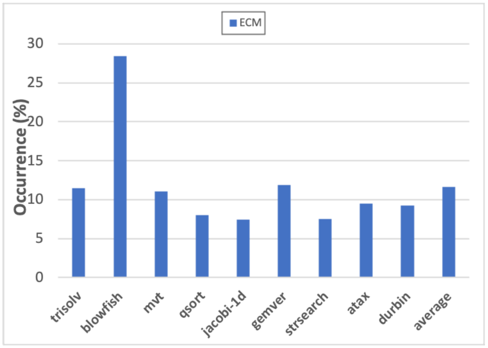
TABLE I: Min, max and highest occurrence gate-level error size.

Pipeline stage	**Fetch**	**Decode**	**Execute**	**Memory**
Min	1	1	1	1
Max	63	79	81	58
Highest occurrence	18	44	47	5

Figure 1 shows the results regarding functional and timing vulnerability. The threshold for considering an application as not responsive (Hang) is set to eight times the number of execution cycles without faults. Regarding the functional correctness (Figure 1a), on average 4,15% of the fault injections has led to application hangs, 11,76% to application crashes, 0,64% to wrong application output, 11,76% to wrong internal state, 7,27% to both wrong application output and wrong internal state, and 64,87% were masked. Regarding timing correctness (Figure 1b), all applications experienced mismatches in their number of execution cycles. The least affected application is `jacobi-1d`, where 7.45% of the total benchmark executions, under the presence of faults, lead to a



(a) Functional correctness



(b) Timing correctness

Fig. 1: Vulnerability metrics

different number of execution cycles compared to the fault-free execution. The most affected application is `blowfish` (28.43%). On average, 11.63% of the executions under faults lead to ECM.

In order to provide further insight on the timing impact, we perform a detailed analysis of the executions, where ECM occurred. More precisely, we consider the following cases:

- *case i*: Execution cycle mismatch and application output mismatch (only AOM).
- *case ii*: Execution cycle mismatch and internal state mismach (only ISM).
- *case iii*: Execution cycle mismatch, application output mismatch and internal state mismatch (AOM and ISM).
- *case iv*: Execution cycle mismatch only (neither AOM, nor ISM).

Table II depicts the number of occurrence for the above cases, per benchmark. Overall, the most observed cases are *case iii* and *case iv*, with 6.55% and 3.46%, respectively.

Especially *case iv* is of utmost interest: when the application is executed under faults, the fault impact cannot be observed neither in the application output, nor in the internal state of the processor. However, the number of execution cycles was different than the fault-free execution. This timing impact cannot be detected by typical fault tolerant approaches that replicate tasks and compare their outputs. Furthermore, such comparison may have to be delayed if one replica takes more cycles due to a faults, compared to the fault-free execution. Figure 2 depicts in details the difference in execution cycles,

TABLE II: Percentage of appearance over all experiments

Benchmark	Case i	Case ii	Case iii	Case iv	Total
trisolv	0.39%	1.86%	8.25%	0.96%	11.46%
blowfish	0.04%	0.44%	27.75%	0.21%	28.44%
mvt	0.39%	0.61%	3.17&	6.91%	11.08%
qsort	1.37%	1.15%	3.63%	1.89%	8.04%
jacobi-1d	0.06%	0.81%	2.37%	4.21%	7.45%
gemver	0.17%	1.91%	2.32%	7.50%	11.90%
strsearch	1.63%	0.77%	3.87%	1.27%	7.54%
atax	0.25%	1.94%	2.83%	4.49%	9.51%
durbin	0.38%	0.41&	4.81&	3.69&	9.29%
Average	0.52%	1.1%	6.55%	3.46%	11.64%

compared with the fault-free execution, for all the benchmarks. The maximum difference is observed for benchmark `atax`, where the execution time took 659.64% more than the fault free execution.

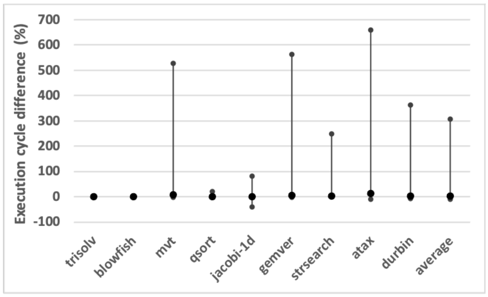


Fig. 2: Timing behavior under transient faults for *case iv*.

Last, but not least, Table III provides the minimum, average and maximum execution cycle difference for *case i* up to *case iv*, considering all applications. The maximum observed average difference is 12.398%, when the fault is manifested only as AOM, 273.95%, when the fault is manifested as ISM, 542.246% when the fault is manifested as AOM and ISM, and 332.742%, when the fault is masked, and it cannot be observed by the application output or the processor internal state, when the application ends.

TABLE III: Average execution time difference

	Min	Avg	Max
Case i	-3.422%	-0.224%	12.398%
Case ii	-13.749%	5.862%	273.975%
Case iii	-84.252%	3.062%	542.246%
Case iv	-9.027%	3.536%	332.742%

IV. Real-time vulnerability analysis driven selective hardening for approximate applications

In this section, we explore how vulnerability analysis can lead to selective protection for real-time applications. We provide a fine-grain, emulation-based, in-field and full stack SEU vulnerability analysis, considering hardware and software levels. Our framework is used to study the impact of SEUs and to propose optimized selective protection. We use an FPGA-emulated hardware model, which enables fault injections at different layers and the operation of the

TABLE IV: Components and total number of Bits per Component in the DMC part of the system

Component	#Bits
Scan Line Buffers (*scnbf*)	96
Old Pixel Cost (*pco*)	2240
New Pixel Cost (*pno*)	2240
Serial-in Parallel Out (*dtpr*)	3072
Multiplier (*m4*)	8
Multiplier (*m6*)	8
Box Filter (*bflr*)	7680
WTA (*wta*)	2000
Address Generator (*ag*)	74
Address Column Sum (*cs*)	11
Sync 1 (*s1*)	22
Sync 2 (*s2*)	21
Total	17472

application in real-time. Using this analysis, we determine the most critical hardware components, which are driving the design of selective protection with reduced area overheads. Moreover, we analyze the area overhead of hardware-based redundancy schemes, when applied at the selected critical components, and we compare it with the whole system protection area overhead. Our case study is a high reliable and execution time constrained computer vision application, which detects obstacles and tries to avoid them during the movement of a robot.

A. Approximate Obstacle Avoidance Use Case

Approximate computer vision applications can be found in various critical mission applications, such as space exploration, autonomous vehicles, and mobile robots. In particular, the application that we use for this evaluation performs Obstacle Avoidance (OA) that can be used by robots in autonomous navigation for detecting and avoiding obstacles in their path [29]. This algorithm is responsible for first detecting the distance of obstacles, and then decides the direction of the robot movement. To perform obstacle avoidance, Disparity Map Computation (DMC) is needed for the estimation of the depth of the obstacles. DMC is based on a high-performance, real-time disparity computation engine and it is suitable for real-time critical applications. After DMC is performed, the OA system selects the region of interest (ROI). The ROI is divided into three regions: left, straight, and right. For each region, a pixel-based analysis is performed, to determine the number of pixels that have a disparity value bigger than a predefined threshold. The final decision of the direction is determined by a Finite-State-Machine (FSM) that selects the safest direction, i.e., the one with the most distanced obstacles.

As the whole process depends on the disparity computation, it is evident that an erroneous disparity map value can result in errors in the selection of the direction of the robot and can potentially lead the robot to crash on an unrecognized obstacle. Due to DMC's criticality, fault injection is performed in this part of the system. The vulnerability analysis is performed for the entire system (DMC and OA) in order to reveal the impact

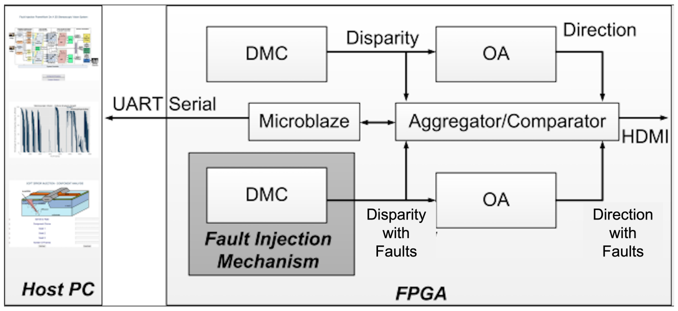


Fig. 3: Experimental Platform - System block diagram

of DMC errors in the OA application. The DMC architecture consists of 12 major components listed in Table IV, for a total number of 17,472 bits, where SEUs can be injected.

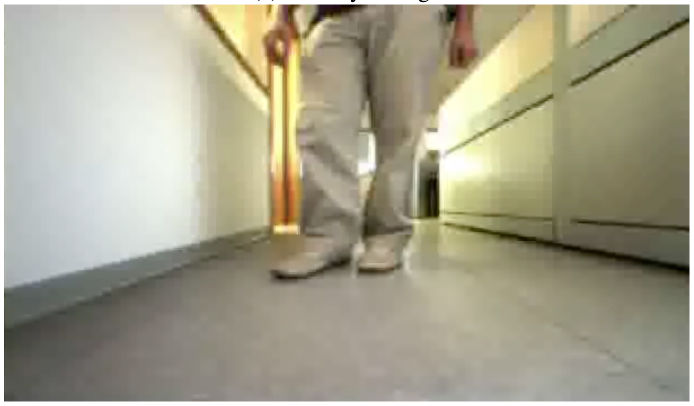
B. Underlying Fault Injection Mechanism

This section discusses the Fault Injection Mechanism (FIM) that is used to provide SEU vulnerability evaluation. The mechanism allows for coarse and fine-grained reliability analysis using hardware-based models for fault injection which are integrated with the system to be evaluated using FPGA-based emulation. Our FIM dynamically generates and targets the injection of faults, without the need of a fault-map, via four different fault injection mechanisms which target different granularity levels. The first mechanism controls the fault injection at the system level and randomly selects a clock cycle, where the fault will be injected, while the application is running. The second mechanism selects a component and the third mechanism selects a specific register inside the selected component. The fourth mechanism is the bit selection mechanism that selects the bit that will be inverted from the selected register. The selections regarding the clock cycle, component, register and bit are done in a pseudo-random manner, utilizing Linear-Feedback Shift Registers. For the analysis presented in this paper, we use all four mechanisms to inject faults. However, we present vulnerability analysis results only at the component level.

C. Selective Hardening

Fine-grained vulnerability analysis can be used to enhance the selective protection for the most vulnerable modules of the system. In this work we, analyze the underlying use case and identify the most critical components that experience SDC errors. SDC errors happen when there is an output discrepancy between the fault-free and faulty execution, which can manifest differently according to the exact impact on the application outcome. We analyze the area overhead of well-known fault tolerant techniques including Dual Modular Redundancy (DMR) and Triple Modular Redundancy (TMR), when applied at the selected critical components. DMR has duplicated components, which work in parallel, and a comparator that compares the two executions, whereas TMR has triplicated components and a voter for selecting the majority output from the three parallel executions.

(a) Left Eye Image


(b) Right Eye Image

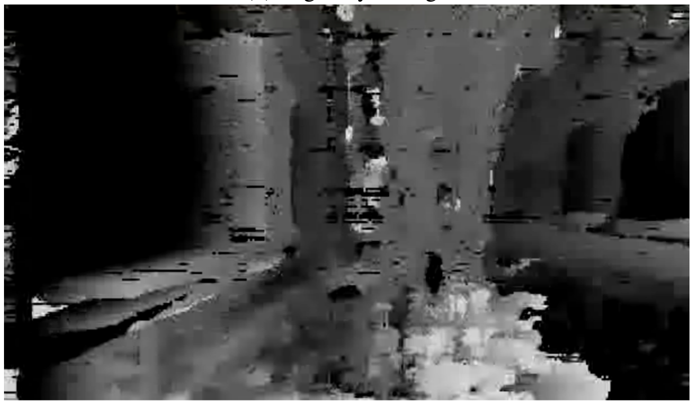

(c) Disparity Results

Fig. 4: Example of (a) input left eye image, (b) input right eye image, and (c) disparity results by DMC

D. Evaluation

The experiments were performed on a prototyped system implemented using Inrevium's Kintex-7 FPGA. The entire process was controlled through an in-house GUI allowing the user to specify the parameters and granularity of the whole process via a host PC connected to the FPGA. We have applied the fault injections at the DMC level and we evaluated the effects of the faults at both DMC and OA level. A database of 23 pairs of stereo images with 1280*720 (720p) resolution was used for our experiments, representing various obstacles in a typical office, such as doors, persons, and other office equipment. Each image in the stereo pair was loaded on two receiver ports of the HDMI Receiver in the Inrevium's Kintex-7 FPGA Display Kit through USB to HDMI cables. To investigate the effects of SEUs at the lower layer of the application (disparity map computation) and subsequently at the higher layer (decision of the obstacle avoidance system), we designed two DMC systems along with two OA systems on the Kintex-7 FPGA, as shown in Figure 3.

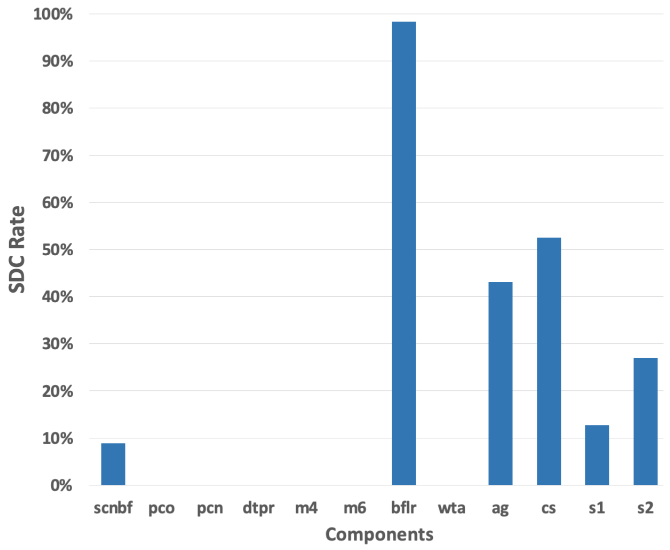


Fig. 5: SDC rate for different components using Obstacle Avoidance Algorithm

Figure 4 shows an example of the test images used in our evaluation (left and right eye images at the top) and the effects in the disparity (image below). In order to monitor and validate if a fault was injected correctly, we have added one more window in our GUI representing the fault injections. When a fault was injected, the color of the window changes. SEUs were injected in all the 12 components of the DMC part, listed in Table IV, at different registers and bits. A set of 19317 SEUs was generated for the whole system. To ensure the statistical correctness of the performed fault injection campaign, we used the method of [24] with a confidence level of $99,8\%$ and an error margin of $0,05\%$.

We first study the SDC rate for the different components by using the fault injection mechanism. To measure this, we inject a fault at the DMC algorithm and we compare the final fault free output of the OA with the faulty one, as shown in Figure 4. Thus, we evaluate the subsequent impact of the DMC on the decision of the OA model, when the different components were injected with SEUs. Figure 5 shows the SDC rate of this component-level analysis on the obstacle avoidance model. For example $scbf$ component shows an SDC rate around 10%. This means that 10% of the fault injected at the $scbf$ component propagate as SDC at some output of the OA algorithm.

The results in Figure 5 clearly indicate that from the 12 components only 6 experience SDC errors. The most vulnerable components are the *scnbf*, *bflr*, *ag*, *cs*, *s1*, and *s2*. This

TABLE V: Area overheads for extra hardware needed to provide DMR and TMR at different design levels (all in NAND gate equivalent)

Redundancy Technique	Redundant Gates	Voters / Comparators	Total Area Overhead
Full System DMR	69888	11	69899 (100.015%)
Full System TMR	139776	10	139786 (200.014%)
Selective Component DMR	31616	577	32193 (46.063%)
Selective Component TMR	63232	410	63642 (91.062%)

result is justified when considering the large number of bits in the *bflr* module, and the fact that, even thought they have a very small size, the *ag*, *cs*, *s1*, and *s2* are control units whose behavior affects the result of the application significantly. Moreover buffer *scnbf* is also more susceptible to errors. This analysis gives an indication of the number of components that need further exploration or, in this case, protection. To further reduce the number of critical components parameterized SDC rate thresholds could be used, which can be determined according to the criticality of the application. For example, a 10% SDC rate threshold would exclude component $scnbf$ from the list of critical components.

To show how the vulnerability analysis can be used to decide on effective detection/correction techniques, we consider DMR which can offer detection, and TMR which can offer both detection and correction. Table V compares DMR and TMR in terms of area overhead (based on NAND gate equivalents), when applying full system redundancy or selective redundancy at the 6 critical components identified in the previous analysis. The table shows for each redundancy technique (first column) the overhead for the extra hardware needed to provide DMR or TMR (second column), including the cost for comparators in DMR, or voters in TMR (third column). Additionally, we provide the total area overhead (fourth column) with the normalized overhead to the non-protected architecture (fourth column in parenthesis). A comparator consists of 4 NAND gate equivalents per bit plus cascading gate network for the output, whereas a voter consists of 5 NAND gates. Flip-flops (bits) require 4 NAND gates each. As the numbers depict in Table V, the selective component approach can provide around 2.17X (2.20X) less total area overhead, compared to full system DMR (TMR). Even thought the number of needed comparators (or voters) increases with selective redundancy, due to increase of outputs to be protected, this overhead is well compensated with the significant decrease of redundant gates needed by the selective protection (second column of V).

V. Time implications of transient faults in middleware for COTS GPUs

In this section, we explore the timing effects of SEUs in the next abstraction level, where a fault-tolerant middleware integrates information from error reporting mechanisms of modern processors and exposes this information to the software, in order to take propoper actions.

Such a feature is very important for systems that have relatively high probability to experience SEUs and, at the same time, need to guarantee availability and correct functionality, such as COTS (commercial off-the-shelf) systems employed in critical domains. This is the case of an emerging space sector, known as *New Space* [23], which is currently on the rise. Unlike institutional missions executed by Space Agencies, which rely on space-proven custom designs, such as radiation hardened and radiation tolerant processors (like Cobham Gaisler's LEON4/NGMP), New Space is characterised by low-cost, commercial missions. As a consequence, New Space uses COTS processors, which, apart from their lower cost, they allow the use of high performance processing technologies, such as embedded Graphics Processing Units (GPUs). GPUs can provide significantly higher processing power than space processors and are promising candidates for adoption in space.

Despite their lower cost, New Space missions have also real-time requirements, which need to be addressed in the presence of transient faults. For example, earth and space observation missions, acquire periodically images through their instruments which need to be processed before the next image is acquired, otherwise data are going to be lost. However, when COTS processors, such as GPUs, are used for processing, the radiation errors need to be considered in order to meet the required timing. In the next subsection we discuss how this can be achieved.

A. Image Processing Space Case Study

In order to demonstrate the concept in a realistic setup, we use an image processing space case study from ESA's OBPMark Benchmark suite [48]. OBPMark is an open source benchmarking suite targeting high performance devices for space. While it was originally developed to benchmark embedded GPUs in order to assess their applicability to space as part of the GPU4S ESA-funded project [39], it is currently featuring implementations of space related algorithms for multiple processing technologies, including multi-core CPUs and a portable sequential implementation in C. Although OBPMark Kernels (also known as GPU4S Bench) was restricted only on computational kernels, which are used across multiple space domains, OBPMark includes full, representative space applications [12]. A third variant of the suite, OBPMark-ML focuses on the implementation of Machine Learning space applications. All OBPMark benchmarking suites are co-hosted in github [12] and come with representative input and verification output in order to ensure the correctness of the algorithms, making them ideal for use in reliability methods.

In this work, we use the #1.1 Image Calibration and Correction application. This application represents the typical on-board processing tasks necessary for panchromatic imaging instruments in scientific remote sensing applications, such as deep-space telescopes with long exposure times.

Frames from several acquisitions are pre-processed individually, then co-registered and summed to create a final image output. After an image is acquired, the following stages are performed for each frame: a) image offset correction, b) bad pixel correction, c) radiation scrubbing, d) gain correction, e) spatial binning and f) temporal binning. These processing steps create the processing pipeline shown in Figure 6.

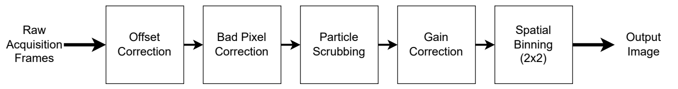


Fig. 6: General Pipeline

Although some processing steps only depend on a single image frame, some steps depend on previous/following frames. For example, the temporal binning step is performed on the current frame, two previous and two already processed frames, so a total of five consecutive frames are required. Figure 7 shows how many and which frames are required for each processing step to have a full image output, after processing eight complete frames. In fact, due to these dependencies, each of the processing stages of the application can be considered as individual, and independent processing pipelines, which have a length equal to the number of frames that they depend on.

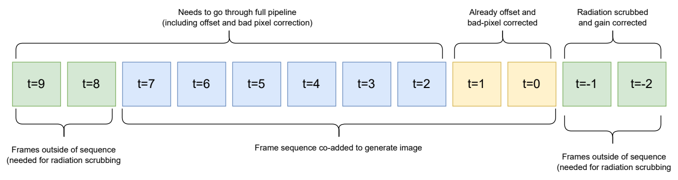

Needs to go through full pipeline (including offset and bad pixel correction)
Already offset and bad-pixel corrected
Radiation scrubbed and gain corrected
t=9
t=8
t=7
t=6
t=5
t=4
t=3
t=2
t=1
t=0
t=-1
t=-2
Frames outside of sequence (needed for radiation scrubbing
Frame sequence co-added to generate image
Frames outside of sequence (needed for radiation scrubbing

Fig. 7: Frame dependencies for each processing step.

The input frames and verification output for this application come from ESA's Herschel mission [17], although the implementation of the algorithms is generic and not tied to that mission, in order to be representative of several image processing space applications.

B. Fault tolerant middleware

We modified the application in order to include 3 points in which we interact with a custom fault tolerant middleware we have developed. The middleware queries the error reporting facilities of the platform, such as MCA, RAS, SMART etc. In case that an uncorrectable error is detected for a given stage, it's data are discarded and the re-execution of its computation is performed. Figure 8 shows the location of these error checks within the processing sequence.

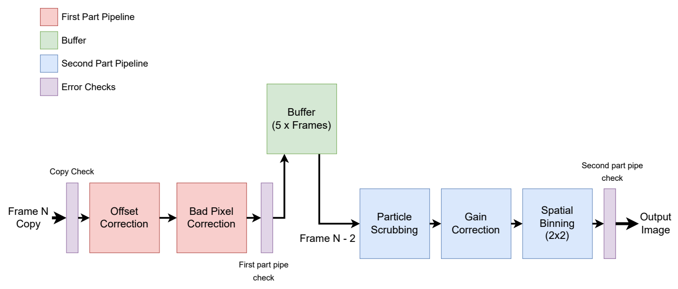


Fig. 8: Pipeline during execution.

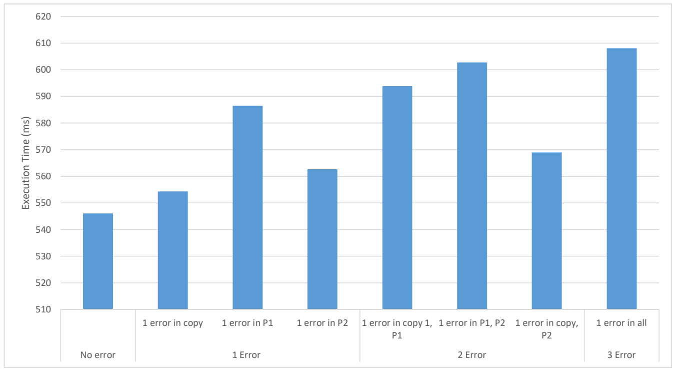


Fig. 9: Execution time with multiple combinations of unrecoverable error in the different middleware check points in the application.

The re-execution increases the execution time of the next output frame. Therefore, the latency between the output of two consecutive frames needs to consider, not only regarding the execution time which is required for the correct processing – as it would have happened on a radiation hardened processor – but also for the re-execution overheads.

Note, however, that the execution time increase is not linear. The reason for this is that, as mentioned before, each of the processing stages forms an independent pipeline. For example, if an uncorrectable error is detected during the image copy from the instrument, the copy operation is replicated. On the other hand, if an error is performed between the first and the second middleware error checks, which we call part 1 (P1) of the application, only the processing performed in the P1 needs to repeated, but not the copy operation. Similarly, if an uncorrectable error is detected between the second and third error checks, only the P2 computation needs to be recalculated.

C. Evaluation

We have evaluated our implementation on an NVIDIA Xavier NX embedded GPU SoC, which is a promising COTS embedded GPU platform for adoption in space, using the CUDA version of the application. That is the image processing steps are executed on the Volta-based embedded GPU of the platform. Since the GPU provides significantly higher performance than the space processors, we use images of 4K×4K size as an input, although the original Herschel instrument is processing 1K×1K sized images.

The deadline between two consecutive processed frames of the application is set to 800 ms. The optical instrument of the Herschel mission provides a new frame with a rate of 10Hz [27], and since the longest processing step of the image processing pipeline of the application requires eight frames, the processing of each output frame requires to be processed in below the 800ms time limit, in order to avoid a missing frame.

Figure 9 shows the timing results of our evaluation, both for the nominal execution without experiencing any unrecoverable transient faults, as well as for combinations of one, two and three errors in each of the error check points. The execution times are collected by measurements over multiple frames, and high watermark values are provided.

The execution time when no errors are experienced is below 550ms. When a single unrecoverable error takes place, the additional execution time overhead varies depending on the failure point. If the error is in the P1, the re-execution time overhead is longer, due to the fact that the bad pixel correction stage is the most time consuming one, while the impact of the other two is minimal. This is also evident when two faults are experienced, and one of them involves a re-execution of P1. Finally, even when three faults are experienced in all the middleware check points, the re-execution time overhead reaches its maximum value, but still well below the 800ms deadline.

In fact, the presented values are overly pessimistic, since simulations based on data obtained during radiation testing using protons on this device, predict that three faults are expected to happen within a single year of operation at a low-earth orbit (LEO) [40].

VI. Conclusions

In this paper we present three case studies of reliability analysis and mitigation at different levels of the system, and we explore their timing and functional impact.

The obtained results of the proposed functional and timing vulnerability analysis show that even when the application's output is correct, the application execution time can be significantly increased under the presence of faults, even up to almost 700%, compared to the application execution time without faults. This observation has a direct three-fold consequence: i) current fault-tolerant techniques, based only on the functional correctness of the application, are incapable of detecting this impact of faults on the application execution time, ii) current WCET estimation methods do not account for this timing impact, and iii) current fault-tolerant techniques, based on fault-free WCET estimations, may become unsafe. To deal with this timing impact, fault tolerant techniques should be employed in order to mitigate it at a lower level than the task level, and WCET estimation approaches should account for it. Our future directions is to analyse the software and hardware parts that lead to significant timing impact and provide countermeasures.

We also highlighted the need of having an FPGA-based emulation vulnerability analysis tool, which allows for in-field fault injections, improving the analysis time. This framework has allowed us to study the impact of SEUs from coarse-level to fine-grained system levels in reasonable time, of an approximate hardware implementation and its effect to the application running on top of it. Moreover, we analyze the area overhead of different redundancy schemes, when applied at the selected critical components. Future work will focus at the register-level vulnerability analysis, which can provide even finer-grained insights that can lead to further area savings for selective redundancy.

Last, we have experimentally measured the timing impact of re-execution required in a space application due to SEUs in a COTS embedded GPU, when a fault tolerant middleware informs the application for non recoverable transient faults. We have seen that the actual overhead depends on the particular execution time cost of the original computation and that factoring this overhead is not straightforward. Moreover, we observed that the performance benefit of a COTS GPU is high enough to allow meeting the deadline, even when accounting for the additional re-execution overhead.

Acknowledgments

This work has been partially funded by ANR-FASY (ANR-21-CE25-0008-01) and received funding by ESA through the 4000136514/21/NL/GLC/my co-funded PhD activity "Mixed Software/Hardware-based Fault-tolerance Techniques for Complex COTS System-on-Chip in Radiation Environments" and the GPU4S (GPU for Space) project. Moreover, it was partially supported by the Spanish Ministry of Economy and Competitiveness under grants PID2019-107255GB-C21 and IJC2020-045931-I (Spanish State Research Agency / http://dx.doi.org/10.13039/501100011033), by the European Union's Horizon 2020 grant agreement No 739551 (KIOS CoE) and from the Government of the Republic of Cyprus through the Cyprus Deputy Ministry of Research, Innovation and Digital Policy.

References

[1] J. Abella, E. Quiñones, F. J. Cazorla, M. Valero, and Y. Sazeides. Rvc-based time-predictable faulty caches for safety-critical systems. In *IEEE Int. On-Line Testing Symp. (IOLTS)*, pages 25–30, July 2011.

[2] Z. Al-bayati, J. Caplan, B. H. Meyer, and H. Zeng. A four-mode model for efficient fault-tolerant mixed-criticality systems. In *IEEE/ACM Design, Automation Test in Europe Conf. Exhibition (DATE)*, 2016.

[3] Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. High-integrity gpu designs for critical real-time automotive systems. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 824–829, 2019.

[4] Sergi Alcaide, Leonidas Kosmidis, Carles Hernandez, and Jaume Abella. Software-only diverse redundancy on gpus for autonomous driving platforms. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 90–96, 2019.

[5] Dario Asciolla, Luigi Dilillo, Douglas Santos, Douglas Melo, Alessandra Menicucci, and Marco Ottavi. Characterization of a risc-v microcontroller through fault injection. In *Applications in Electronics Pervading Industry, Environment and Society (APPLEPIES)*, Lecture Notes in Electrical Engineering, pages 91–101. Springer Open, 2019.

[6] A. Bhat, S. Samii, and R. Rajkumar. Practical task allocation for software fault-tolerance and its implementation in embedded automotive systems. In *IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 87–98, April 2017.

[7] I. Chadjiminas, C. Kyrkou, T. Theocharides, M. K. Michael, and C. Ttofis. In-field vulnerability analysis of hardware-accelerated computer vision applications. In *Int. Conf. Field Programmable Logic and Applications (FPL)*, pages 1–4, Sep. 2015.

[8] I. Chadjiminas, I. Savva, C. Kyrkou, M. K. Michael, and T. Theocharides. Emulation-based hierarchical fault-injection framework for coarse-to-fine vulnerability analysis of hardware-accelerated approximate algorithms. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 830–833, March 2016.

[9] C. Chang, S. Lym, N. Kelly, M. B. Sullivan, and M. Erez. Hamartia: A fast and accurate error injection framework. In *IEEE/IFIP Int. Conf. on Dependable Systems and Networks Workshops (DSN-W)*, 2018.

[10] Gang Chen, Nan Guan, Kai Huang, and Wang Yi. Fault-tolerant real-time tasks scheduling with dynamic fault handling. *Journal of Systems Architecture (JSA)*, 102:101688, 2020.

[11] P. Corneliou, P. Nikolaou, M.K. Michael, and T. Theocharides. Fine-grained vulnerability analysis of resource constrained neural inference accelerators. In *IEEE Int. Symp. Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6. IEEE, 2021.

[12] D. Steenari et al. On-Board Processing Benchmarks, 2021. http://obpmark.github.io/.

[13] A. Dixit and A. Wood. The impact of new technology on soft error rates. In *Int. Reliability Physics Symp.*, pages 5B.4.1–5B.4.7, April 2011.

[14] B. Du, S. Azimi, C. de Sio, L. Bozzoli, and L. Sterpone. On the reliability of convolutional neural network implementation on sram-based fpga. In *Int. Symp. Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT)*, pages 1–6, 2019.

[15] Veronique Ferlet-Cavrois, Lloyd W Massengill, and Pascale Gouker. Single event transients in digital cmos—a review. *IEEE Transactions on Nuclear Science*, 60(3):1767–1790, 2013.

[16] 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 (TNS)*, 66(7):1449–1456, 2019.

[17] M. Harwit. The herschel mission. *Advances in Space Research*, 34(3):568–572, 2004.

[18] Carles Hernandez and Jaume Abella. Low-cost checkpointing in automotive safety-relevant systems. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 91–96, 2015.

[19] P. Huang, H. Yang, and L. Thiele. On the scheduling of fault-tolerant mixed-criticality systems. In *ACM/EDAC/IEEE Design Automation Conf. (DAC)*, pages 1–6, June 2014.

[20] G. Hubert, R. Velazco, and P. Peronnard. A generic platform for remote accelerated tests and high altitude SEU experiments on advanced ICs: Correlation with MUSCA SEP3 calculations. In *IEEE IOLTS*, pages 180–180, June 2009.

[21] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba. Impact of Scaling on Neutron-Induced Soft Error in SRAMs From a 250 nm to a 22 nm Design Rule. *IEEE Trans. on Electron Devices*, 57(7):1527–1538, July 2010.

[22] J. Kim, G. Bhatia, R. Rajkumar, and M. Jochim. Safer: System-level architecture for failure evasion in real-time applications. In *IEEE Real-Time Systems Symp. (RTSS)*, pages 227–236, Dec 2012.

[23] L. Kosmidis, I. Rodriguez, A. Jover-Alvarez, S. Alcaide, J. Lachaize, O. Notebaert, A. Certain, and D. Steenari. GPU4S: Major Project Outcomes, Lessons Learnt and Way Forward. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1314–1319. IEEE, 2021.

[24] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert. Statistical fault injection: Quantified error and confidence. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 502–506, April 2009.

[25] D. Li, J. S. Vetter, and W. Yu. Classifying Soft Error Vulnerabilities in Extreme-Scale Scientific Applications Using a Binary Instrumentation Tool. In *Int. Conf. on High Performance Computing, Networking, Storage & Analysis (SC)*, pages 1–11, Nov 2012.

[26] A. Lotfi, S. Hukerikar, K. Balasubramanian, P. Racunas, N. Saxena, R. Bramley, and Y. Huang. Resiliency of automotive object detection networks on gpu architectures. In *IEEE International Test Conference (ITC)*, pages 1–9. IEEE, 2019.

[27] M. J. Griffin et al. The Herschel-SPIRE instrument and its In-flight Performance. *Astronomy and Astrophysics*, 518(L3), 2010.

[28] N. N. Mahatme, S. Jagannathan, T. D. Loveless, L. W. Massengill, B. L. Bhuva, S. Wen, and R. Wong. Comparison of Combinational and Sequential Error Rates for a Deep Submicron Process. *IEEE Trans. on Nuclear Science (TNS)*, 58:2719–2725, December 2011.

[29] Y. Miyajima and T. Maruyama. *A Real-Time Stereo Vision System with FPGA*, pages 448–457. Springer Berlin Heidelberg, Berlin, Heidelberg, 2003.

[30] Burcu O. Mutlu, Gokcen Kestor, Adrian Cristal, Osman Unsal, and Sriram Krishnamoorthy. Ground-truth prediction to accelerate soft-error impact analysis for iterative methods. In *International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 333–344, 2019.

[31] Marco Ottavi, Salvatore Pontarelli, Dimitris Gizopoulos, Cristiana Bolchini, Maria K Michael, Lorena Anghel, Mehdi Tahoori, Antonis Paschalis, Pedro Reviriego, Oliver Bringmann, et al. Dependable multicore architectures at nanoscale: The view from europe. *IEEE Design & Test*, 32(2):17–28, 2014.

[32] B. Ozcelik Mutlu, G. Kestor, J. Manzano, O. Unsal, S. Chatterjee, and S. Krishnamoorthy. Characterization of the Impact of Soft Errors on Iterative Methods. In *IEEE Int. Conf. on High Performance Computing (HiPC)*, pages 203–214, December 2018.

[33] Risat Pathan. Real-time scheduling algorithm for safety-critical systems on faulty multicore environments. *Real-Time Systems (RTS)*, 09 2016.

[34] Joseph Paturel, Angeliki Kritikakou, and Olivier Sentieys. Fast Cross-Layer Vulnerability Analysis of Complex Hardware Designs. In *ISVLSI 2020 - IEEE Computer Society Annual Symposium on VLSI*, pages 328–333, Limassol, Cyprus, July 2020. IEEE.

[35] Ilia Polian and John P Hayes. Selective hardening: Toward cost-effective error tolerance. *IEEE Design & Test of Computers*, 28(3):54–63, 2010.

[36] Alexis Ramos, Juan Antonio Maestro, and Pedro Reviriego. Characterizing a RISC-V SRAM-based FPGA implementation against Single Event Upsets using fault injection. *Microelectronics Reliability*, 78, November 2017.

[37] B. Reagen, U. Gupta, L. Pentecost, P. Whatmough, S.K. Lee, N. Mulholland, D. Brooks, and G.-Y. Wei. Ares: A framework for quantifying the resilience of deep neural networks. In *ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2018.

[38] S. Rehman, M. Shafique, and J. Henkel. *Reliable Software for Unreliable Hardware: A Cross Layer Perspective*. Springer, 2016.

[39] I. Rodriguez, L. Kosmidis, J. Lachaize, O. Notebaert, and D. Steenari. GPU4S Bench: Design and Implementation of an Open GPU Benchmarking Suite for Space On-board Processing. Technical Report UPC-DAC-RR-CAP-2019-1, Universitat Politècnica de Catalunya. https://www.ac.upc.edu/app/research-reports/public/html/research_center_index-CAP-2019,en.html.

[40] I. Rodriguez-Ferrandez, M. Tali, L. Kosmidis, M. Rovituso, and Da. Steenari. Sources of Single Event Effects in the NVIDIA Xavier SoC Family under Proton Irradiation. In *IOLTS*, 2022.

[41] Simon Rokicki, Davide Pala, Joseph Paturel, and Olivier Sentieys. What You Simulate Is What You Synthesize: Designing a Processor Core from C++ Specifications. In *IEEE/ACM Int. Conf. on Computer-Aided Design (ICCAD)*. IEEE, November 2019.

[42] P. K. Samudrala, J. Ramos, and S. Katkoori. Selective triple modular redundancy (stmr) based single-event upset (seu) tolerant synthesis for fpgas. *IEEE Transactions on Nuclear Science (TNS)*, 51(5):2957–2969, Oct 2004.

[43] N. Seifert, B. Gill, S. Jahinuzzaman, J. Basile, V. Ambrose, Q. Shi, R. Allmon, and A. Bramnik. Soft Error Susceptibilities of 22 nm Tri-Gate Devices. *IEEE Trans. Nuclear Science (TNS)*, 59, 2012.

[44] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN*, pages 389–398, 2002.

[45] Jayati Singh, Ignacio Sax00F1;udo Olmedo, Nicola Capodieci, Andrea Marongiu, and Marco Caccamo. Reconciling qos and concurrency in nvidia gpus via warp-level scheduling. In *2022 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1275–1280, 2022.

[46] S. Skalistis and A. Kritikakou. Timely fine-grained interference-sensitive run-time adaptation of time-triggered schedules. In *IEEE Real-Time Systems Symposium (RTSS)*, 2019.

[47] J. Song and G. Parmer. C'mon: a predictable monitoring infrastructure for system-level latent fault detection and recovery. In *IEEE Real-Time and Embedded Technology and Applications Symp. (RTAS)*, pages 247–258, April 2015.

[48] D. Steenari, L. Kosmidis, I. Rodriguez-Ferrandez, A. Jover-Alvarez, and K. Forster. OBPMark (On-Board Processing Benchmarks) - Open Source Computational Performance Benchmarks for Space Applications. In *European Workshop on On-Board Data Processing (OBDP)*, 2021.

[49] M. G. Trindade, A. Coelho, C. Valadares, R. A. C. Viera, S. Rey, B. Cheymol, M. Baylac, R. Velazco, and R. P. Bastos. Assessment of a hardware-implemented machine learning technique under neutron irradiation. *IEEE Transactions on Nuclear Science (TNS)*, 66(7):1441–1448, July 2019.

[50] I. Tuzov, D. de Andrés, and J. Ruiz. Accurate Robustness Assessment of HDL Models Through Iterative Statistical Fault Injection. In *European Dependable Computing Conf. (EDCC)*, pages 1–8, September 2018.

[51] Jiesheng Wei, Anna Thomas, Guanpeng Li, and Karthik Pattabiraman. Quantifying the accuracy of high-level fault injection techniques for hardware faults. In *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 375–382, 2014.

[52] Edward J Wyrwas. Proton Testing of AMD e9173 GPU. Technical Report GSFC-E-DAA-TN72682, NASA, 2019.