Efficient Soft Error Vulnerability Analysis Using Non-intrusive Fault Injection Techniques

Vitor Bandeira, Felipe Rosa, Ricardo Reis, Luciano Ost

► To cite this version:

Vitor Bandeira, Felipe Rosa, Ricardo Reis, Luciano Ost. Efficient Soft Error Vulnerability Analysis Using Non-intrusive Fault Injection Techniques. 27th IFIP/IEEE International Conference on Very Large Scale Integration - System on a Chip (VLSI-SoC), Oct 2019, Cusco, Peru. pp.115-137, 10.1007/978-3-030-53273-4_6 . hal-03476605

HAL Id: hal-03476605

https://inria.hal.science/hal-03476605v1

Submitted on 13 Dec 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

Efficient Soft Error Vulnerability Analysis Using Non-intrusive Fault Injection Techniques

Vitor Bandeira[1], Felipe Rosa[1], Ricardo Reis[1], and Luciano Ost[2]

[1] PPGC/PGMicro — UFRGS
Porto Alegre, Brazil
[2] Loughborough University
Loughborough, United Kingdom
{vvbandeira, frdarosa, reis}@inf.ufrgs.br
l.ost@lboro.ac.uk

Abstract. Electronic computing systems are integrating modern multicore processors and GPUs aiming to perform complex software stacks in different life-critical systems, including health devices and emerging self-driving cars. Such systems are expected to experience at least one soft error per day in the near future, which may lead to life-threatening failures. To prevent these failures, critical system must be tested and verified while under realistic workloads. This paper presents four novel non-intrusive fault injection techniques that enable full fault injection control and inspection of multicore systems behavior in the presence of faults. Proposed techniques were integrated into a fault injection framework and verified through a real automotive case study with up to 43 billions instructions. Results show that compared to traditional methods, the new techniques can increase the efficiency of fault injection campaigns during early development phase by 32.28%.

1 Introduction

Leading companies in automotive, medical, consumer electronics, and high-performance computing (HPC) industry employ general-purpose multicore processors and graphics processing units (GPUs) in their applications. The rising demand for powerful computing capacity and energy efficiency of multicore components lead to high-frequency clock operation and multiple voltage domains within the same chip. In addition to that, the increasing number of internal elements (e.g., cores, memory cells, registers) is making multicore-based systems more vulnerable to both hard and soft radiation-induced errors [1, 2]. Managing the soft error occurrence is crucial to accomplishing a reliable and efficient operation in several domains. In an HPC system, an undetected soft error can impact on the efficiency of resource utilization (i.e., re-execution of applications/jobs), which may lead to financial loss. In turn, the occurrence of a soft error may cause a critical failure on a self-driving car, which can put human lives at risk.

Given trends for ever-increasing application/kernel code size and complexity, cost-effective tools to assess the soft error resilience of multicore-based systems

become of utmost importance to identify the most unreliable system functionalities early in the design phase. In this regard, the high cost and time inherent to hardware-based fault injection methods make more efficient simulation-based fault injection frameworks key to test reliability. Most fault injection simulators available in the literature offer a restricted number of fault injection exploration capabilities such as injection of bit-flips in memory [3], general-purpose registers and some other CPU components (e.g., load/store queue) [4, 5]. However, with the growing complexity of both processor and software architectures, more appropriate fault injection techniques and tools are required. The underlying techniques and tools must provide engineers with full fault injection control and inspection of the system's behavior under the presence of faults.

This paper proposes *four novel non-intrusive* fault injection techniques enabling engineers to perform in-depth and relevant soft error evaluation [3], addressing the gap between the available fault injection tools and the industry requirements. These techniques consider the particularities of each software stack component (e.g., kernel, hypervisor, or application function) running on the target system. To maximize this research impact, we adopt a new tool called **SOFIA** (Soft error Fault Injection Analysis) [6]. SOFIA integrates the proposed fault injection techniques along with several facilities (e.g., error tracer module), which enable to identify and classify the effects of soft errors on the system behavior, considering both hardware and software architectures. SOFIA is based on the Multicore Developer (M*DEV) virtual platform [4], and its implementation is highly autonomous and requires little human interaction, after its configuration.

This work is organized as follows, in Section 2 we review relevant works regarding simulation frameworks and soft error analysis. Section 3 presents our tool, its components and the simulation flow. Section 4 reviews two traditional fault injection techniques and introduces the four novel techniques, then Section 5 show results that support the consistency and runtime advantages of our tool. Sections 6 and 7 contain results of soft error analysis using SOFIA for a multicore benchmark and an automotive application, respectively. Finally, in Section 8 we conclude and discuss future works.

2 Related Works in Fault Injection Frameworks

Authors in [7] present the Relyzer, a hybrid simulation framework for SPARC core using Simics [8] and GEMS [9] simulators coupled with a pruning technique to reduce the number of injected faults. The Relyzer enables the injection of faults into architectural integer registers, and output latches of the address generation unit. In [10], a QEMU-based fault injection framework is proposed targeting general-purpose registers. Fault injection campaigns [10] consider an

[3] A soft error campaign (and thus the evaluation of said campaign) in the context of this paper is considered to be relevant when the result can either identify the existence of vulnerabilities or their source.

[4] www.imperas.com

X86 architecture running four in-house applications on the top of RTEMS kernel. F-SEFI is another fault injection framework that relies on QEMU [11, 12]. This work employs the QEMU using a hypervisor mode, i.e., it does not emulate the complete target system, which reduces both its fault injection and soft error analysis capabilities.

The authors in [3] propose the GeFIN and the MaFIN tools, which support the injection of faults in microarchitectural components such as general-purpose and cache control registers. Conducted experiments consider the execution of 10 bare metal benchmarks. Rosa *et. al.* [13] propose the OVPsim-FIM framework on which several fault injection campaigns were performed in Arm processors running FreeRTOS kernel. Authors in [5] propose a gem5-based framework that allows injecting faults in different microarchitecture elements (e.g., reorder buffer, load-store queue, register file). In [5], each element is subject to small 300-long fault campaign for each of the ten applications collected from both MiBench and SPEC-Int 2006 benchmark suites. A similar gem5-based fault injection framework is described in [4].

The reviewed frameworks only support the injection of bit-flips in memory and general single-core processor components, including registers, load/store queue, among others (Table 1). Another drawback of such approaches is the lack of detailed and customizable post-simulation analysis. Reviewed works classify the detected soft errors according to the inspection of the processor architecture context (i.e., memory and registers), disregarding the impact of software components (e.g., functions and variables) on the system reliability. Further, such approaches typically report low simulation performances of up to 3 MIPS [7], which restricts the number and the complexity of fault injection campaigns. While some works consider a single ISA [7], others use only in-house applications [10] or bare-metal implementations [5, 3, 4].

Different from the above works, SOFIA offers four novel non-intrusive fault injection techniques that provide engineers with flexibility and full control over the fault injection process, allowing to disentangle the cause and effect relationship between an injected fault and the occurrence of possible soft errors, targeting a specific critical application, operating system or API structure/function. Our contribution also differs from all previous projects by allowing users to define bespoke fault injection analysis and soft error vulnerability classifications, taking into account both software and hardware components particularities and the system requirements. SOFIA framework was developed based on M*DEV simulator, and it enables to inject faults at a speed of over 3,900 MIPS while running complete software stacks, allowing fast soft error reliability assessment during early design exploration phases. Further, distinctly from other reviewed works, the promoted tool does not alter the simulator engine by using already provided extension ports to access system hardware components.

Table 1. State-of-the-art in virtual platform (VP) fault injection simulators (Sim.), where 'N/A' means 'not available'.

Ref.	*VP Sim.*	*Kernel*	*Fault Injection Description*
[3]	MARSS gem5	N/A	• General-purpose registers • L1 and L2 cache • Load/Store Queue
[4]	gem5	N/A	• General-purpose registers • Pipeline and functional units registers • Load/Store Queue
[5]	gem5	N/A	• Eleven microarchitectural components
[7]	Simics+GEMS	Open-Solaris	• General-purpose registers • Address generation latches
[10]	QEMU	RTEMS	• General-purpose registers
[11]	QEMU	N/A	• General-purpose registers • L1 and L2 cache • Physical Memory
[13]	OVPsim	FreeRTOS	• General-purpose registers • Physical Memory
This Work	M*DEV	Linux	• General-purpose registers • Physical Memory • Virtual Memory • Variables • Function Code • Function Lifespan

3 SOFIA: Fault Injection Framework

To validate and demonstrate the potential of proposed techniques, the M*DEV simulator was selected due to its support to more than 170 processor model variants (e.g., MIPS, Arm, single-core, dual-core) including state-of-the-art multicore processors and ISAs. Note that proposed techniques can be implemented in any virtual platform or simulation environment that provides access to the system memory management unit (MMU) translation tables. This section details the SOFIA fault injection framework and its main features.

3.1 Fault Model

SOFIA emulates the occurrence of single-bit-upsets (SBUs) by injecting faults into pre-selected register or memory locations during the execution of a given software stack. This paper focus on SBUs for brevity, nevertheless, the tool applies a 64 bit-wide to each target locations enabling any arbitrary multiple-bit upset fault injection. The default fault injection configuration (e.g., bit location, injection time) relies on a random uniform function, which is a well-accepted fault injection technique since it covers the majority of possible faults on a system at a low computation cost [14]. Fault injections occur during the target application lifespan (i.e., the operating system (OS) startup is not subject to faults), which includes OS system calls and parallelization API subroutines arising during this period. This approach allows identifying unexpected application execution errors (e.g., segmentation fault), which are associated with adopted OS components or API libraries.

3.2 Fault Injector Module

SOFIA incorporates a fault injector module (FIM) with five main components: (1) configuration, (2) fault monitor, (3) fault injector, (4) error analysis, and (5) exception handler. The configuration component (1) starts the simulation, reads the configuration file (i.e., fault list), and setups the monitor component (2), which is responsible for controlling the simulator flow. When the simulation reaches the injection time, the fault injector (3) is invoked, and it alters the microarchitectural elements (e.g., register file, physical memory) according to the adopted fault injection technique. After the application execution, the soft error analysis component (4) compares the simulator context against the reference execution (i.e., the application execution without fault injection) to classify the application behavior under fault presence. The analysis considers memory, register context (including the program counter), and the number of executed instructions. Additionally, component (5) automatically terminates the fault injection simulation after an execution time threshold defined by the user and captures unexpected termination events in the target application and OS.

3.3 Fault Injection Simulation Flow

The SOFIA fault injection flow (Fig. 1) comprises four phases: The faultless execution (phase 1) first cross-compiles the application source code and then simulates the target application without fault influence, aiming to verify its correctness and extract reference information, i.e., registers context and final memory state. During the simulation, SOFIA acquires additional information based on the selected fault injection technique. The second phase deploys the fault generation tool considering the injection time, the register name, and the target bit for each fault injection technique. In the third and most complex phase, the SOFIA tool starts by configuring an instruction counter event, which is defined according to the insertion time. Then, the FIM reads the fault characteristics and introduces a bit-flip according to the adopted fault injection techniques. After the application conclusion, the fault injection module compares the application outcome (e.g., the number of executed instructions, registers context, and memory state) under fault influence with the information acquired during phase 1. In the last phase, SOFIA assembles all the individual reports to create a single file, performs several statistical analysis (e.g., average, worst, and best cases) and generates individual plots.

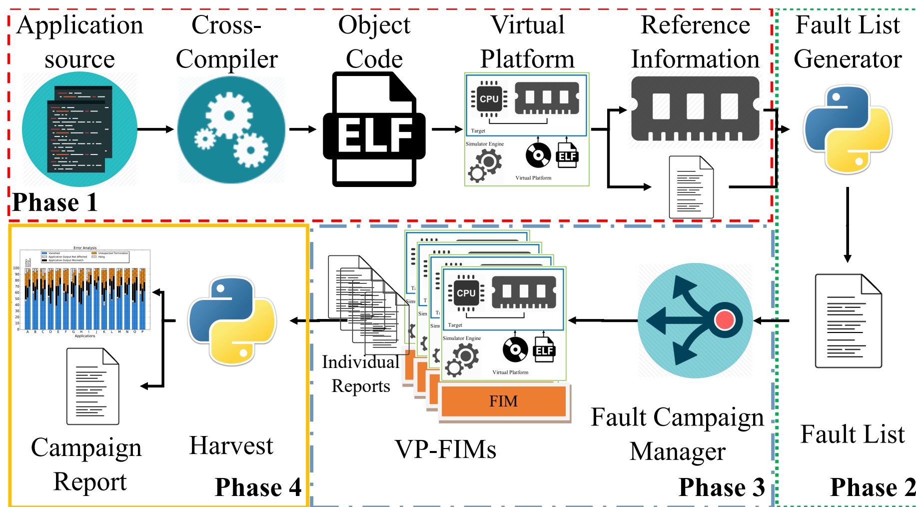

Fig. 1. SOFIA fault injection flow.

3.4 Soft Error Analysis and Classification

The *soft error analysis module* investigates the target platform software stack (i.e., application, drivers, OS under the fault injection influence) after each simulation to expose discrepancies against an identical software stack in a faultless

execution. Fault injections campaigns must be followed by a customizable and flexible soft error analysis, which provides engineers with appropriate means to isolate and identify not only the occurrence but also the system characteristics (i.e., software and hardware) contributing to the error. SOFIA enables the addition of customizable inspections based on application code, execution pattern, or even final results without any modification on the original target software code. For instance, the tool can be used to check a critical variable against a predefined value or another internal variable (e.g., data duplication) ensuring the application correctness. All the soft error analysis conducted in this paper rely on a well known [15], and on a customized classification proposed by the Authors of this work.

Cho *et. al.* [15] classification considers: **Vanished**, if no fault traces are left. **Application Output Not Affected** (ONA), when the resulting memory is not modified; nevertheless, one or more remaining bits of the architectural state is incorrect. **Application Memory Mismatch** (OMM), the application terminates without any error indication; however, the resulting memory is affected. **Unexpected termination** (UT), the application terminates abnormally with an error indication. **Hang**, the application does not finish, requiring a preemptive removal after a threshold execution time. Depending on the application's nature, Cho's classification may be inadequate to express possible misbehavior. SOFIA enables the creation of new classifications to achieve a customized soft error analysis. This feature is fully explored in the Section 7.1, Automotive Results Analysis.

4 Fault Injection Techniques

The SOFIA framework supports six fault injection techniques **(A-F)**, which are illustrated in Fig. 2. These techniques make SOFIA suitable for fast and detailed soft error vulnerability analysis at an early design space exploration stage. Obtaining early indications of soft errors enable reliability software developers to adjust the application code (or portion of it) as needed. Note that the *six* techniques target the register file or physical memory without altering the target software stack (i.e., application, OS, and related libraries).

4.1 Register File

Random register file fault injection is a well-accepted mechanism that homogeneously covers the majority of soft errors, striking both application and operating system codes. This approach ignores distinct regions of criticality (i.e., function and data structures), leading to reduced code coverage that narrows the number of errors that can be detected during the development phase. The SOFIA framework can access the processor model register file to inject faults in any visible register without altering the application under test.

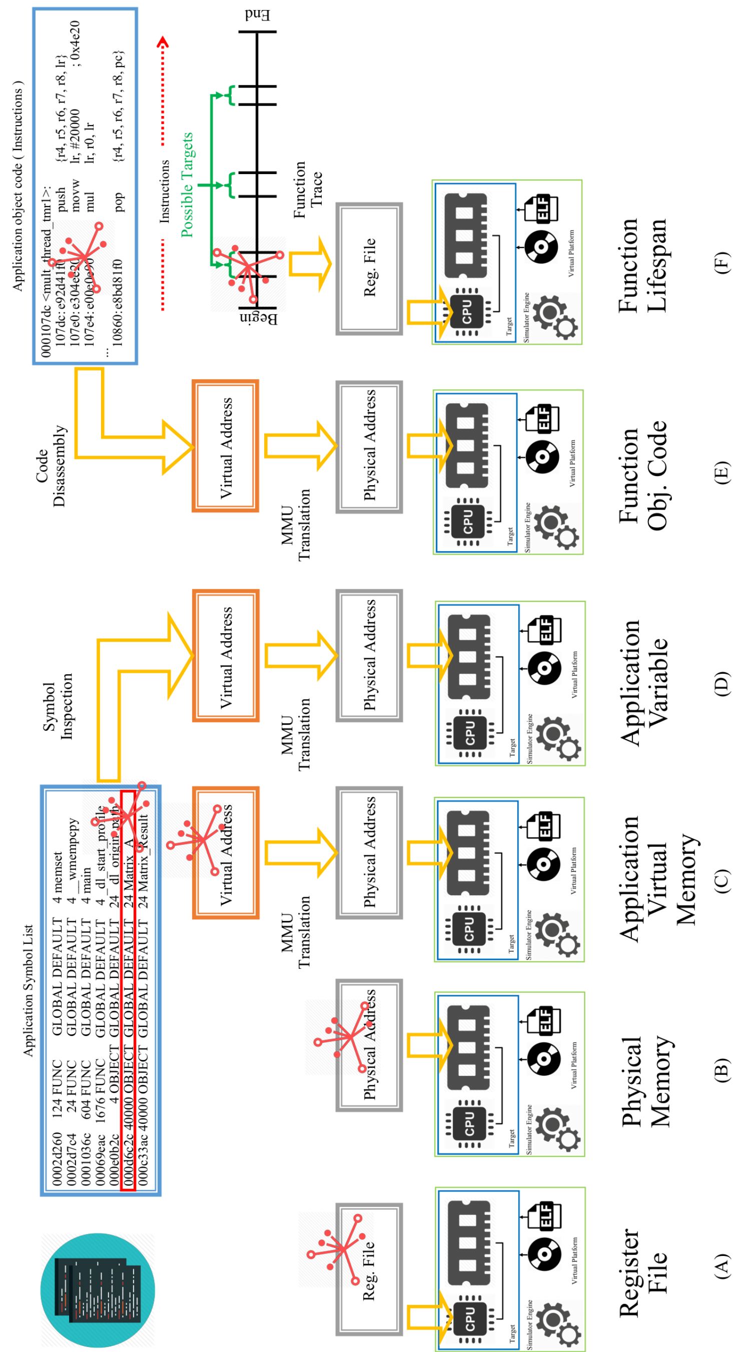

Fig. 2. Fault injection techniques targets (Section 4). The red mark indicates where the bit-flip occurs and the yellow arrows its propagation.

4.2 Physical Memory

Another well-known fault injection technique relies on the inversion of single-bits into the system physical memory, which accurately reproduces its exposition to radiation particles. Nevertheless, the lack of correlation between the injected faults location in the physical memory and the application-level data structures may lead to inadequate error coverage. To enable this technique, SOFIA supports access to both the physical memory and the injection of bit-flips at any moment during the application execution.

Traditional fault injection frameworks only support the techniques **(A)** and **(B)**, which rely on a random selection in terms of fault injection location (e.g., one bit from the complete memory range or a register) and time. Although both are well-accepted mechanisms since they cover the majority of possible faults on a system, it lacks correlation between faults and errors. Faults are arbitrary distributed throughout the execution, striking both application and operating system codes. Underlying techniques may lead to low code coverage, which restricts the identification of soft errors due to the large number of errors that are masked. Fault injection techniques **(C-F)** aim to minimize such limitation while keeping the software stack unmodified.

4.3 Application Virtual Memory

Most operating systems abstract the physical hardware implementation of the memory from the user by making available a set of virtual address ranges while using a translation table to connect both virtual and physical ranges. The promoted technique **(C)** automatically extracts the virtual addressing ranges from the target application object code, including different segment addresses (e.g., data, code, read-only, debug) during its phase 1 (Section 3.3) to create an appropriated fault lists (phase 2). For each fault injection, SOFIA accesses the target OS virtual memory translation table, acquires the correspondent physical to a virtual address, and injects the bit-flip in the system physical memory representing the target application-level memory. The advantage of this technique over the purely physical memory fault injection relies on the fact that it targets the application virtual address space without affecting the OS, the execution of other applications, or libraries, reducing the number of faults campaigns that must be conducted since soft errors are more likely to manifest earlier. This approach enables the user to target a particular application running in a complex environment with multiple applications and libraries.

4.4 Application Variables and Data Structures

To precisely evaluate an application's vulnerability to soft errors the fault injection infrastructure should provide efficient means to correlate errors with particular application blocks or data structures. Technique **(D)** (Application Variable) enables the engineer to direct bit-flip injections into particular data structures, allowing to isolate and identify the most vulnerable ones with a lower number of

fault campaigns and higher precision. Further, this approach allows evaluating the impact of specific application variables on the soft error reliability without affecting the application control flow. For this purpose, the user is asked to inform the target variable name, enabling SOFIA to automatically capture the variable virtual address to create a set of faults targeting the data structure virtual addressing. During any point of the application execution, the variable will suffer a single bit-flip on its physical memory representation using the translation table.

4.5 Function Object Code

To explore the impact of errors on functions assembly code, this work proposes the technique **(E)** that limits the injection spectrum to the memory region which holds the target function code—instructions and local variables. In the real world, the probability of a particular function being hit by a transient fault depends on its size (i.e., number of instructions) in comparison with the complete memory range. This technique enables the user to investigate the soft error reliability of a particular function independent of its size or execution time.

4.6 Function Lifespan

The majority of frameworks rely on a random time generation scheme where faults are scattered over the entire application and OS execution. Consequently, the number of faults per function depends on its execution time and not on its criticality level. One can argue that the critical system functions are those with the most extended execution, however, any function can produce a system malfunction that can impact on the overall system reliability. *Function Lifespan* **(F)** technique enables to reduce the fault injection spectrum by limiting the insertion time to those intervals where the target function is active—in the register context. During the simulation, the fault monitor component (**(2)**, Section 3.2) traces the function execution at the instruction level and thus create a list of active ranges, including the processor core (s) that executed the underlying function. In this work, the lifespan technique implementation targets all the available registers: floating-point, general purposed registers (r0-r15), the program counter (PC), and the stack pointer (SP). However, this technique can be combined along with any other fault injection technique, e.g., **(C-E)**, to further narrow the fault target.

5 Techniques Consistency and Performance

To validate the proposed fault injection techniques as well as to demonstrate the effectiveness of SOFIA a set of experiments are described as follows. Section 5.1 investigates the soft error analysis consistency of SOFIA concerning a cycle accurate full system simulator (gem5). Whereas in Section 5.2, SOFIA simulation performance is compared to the gem5 fault injection implementation. Further,

Section 6 analysis the soft error reliability of a benchmark considering the use of a mitigation technique. In turn, in Section 7 a real automotive case study is used to investigate the proposed fault injection techniques.

For this section, the software stack comprises an unmodified Linux kernel (3.13) and 8 applications from the NASA NAS Parallel Benchmark (NPB) suite [16]. Each application has three versions, the base serial and two parallel implementations with different libraries (OpenMP, MPI), totaling 16 parallel scenarios. The target architecture includes an Arm Cortex-A9 processor model.

5.1 Accuracy

Aiming at evaluating the SOFIA accuracy the register file technique **(A)** has been integrated into gem5 full system mode. The gem5 simulator supports detailed cycle-accurate simulation of the system components (e.g., processor, cache, pipelines, arithmetic units), which justifies its adoption as the reference. Underlying evaluation comprises 32 fault injection campaigns (16 for each simulator) considering eight NASA benchmarks Block Tri-diagonal solver (BT), Conjugate Gradient (CG), Embarrassingly Parallel (EP), Discrete 3D fast Fourier Transform (FT), Integer Sort (IS), Lower-Upper Gauss-Seidel solver (LU), Multi-Grid (MG), and Scalar Penta-diagonal solver (SP) implemented in both MPI and OpenMP.

For each benchmark, 8,000 faults are injected in the registers of a quad-core Arm Cortex-A9 processor in random order, aiming to estimate the percentage of errors that are not masked during the execution of each benchmark. Such experiments deploy 256,000 fault injections through more than 400 thousand of simulation hours using a 5,000-core high-performance system.

Results illustrated in Fig. 3 show that the average mismatch of the SOFIA w.r.t gem5 fault injection implementation is only 5.84%, while the worst case is 23.35% for the OpenMP implementation of MG. Note that the mismatch between two fault injection campaigns is defined here as the sum of absolute differences between each soft error occurrence (e.g., ONA, OMM), divided by the total number of fault injections. Considering that the experiments have as reference a cycle-accurate simulator, which deploys a two-level detailed cache model, and eight high-performance applications implemented in both MPI and OpenMP parallelization libraries; the achieved mismatch is quite acceptable for early reliability explorations of multicore systems executing complex software stacks, specially when approximately 99% (396 hours) of the simulation time was devoted to gem5 simulation.

5.2 Simulation Speed

During early design space explorations the simulation time is an important factor as a lower simulation time allows for a more thorough evaluation. This experiment evaluates the SOFIA simulation performance in terms of millions of instructions per second (MIPS) when compared to the gem5 fault injection

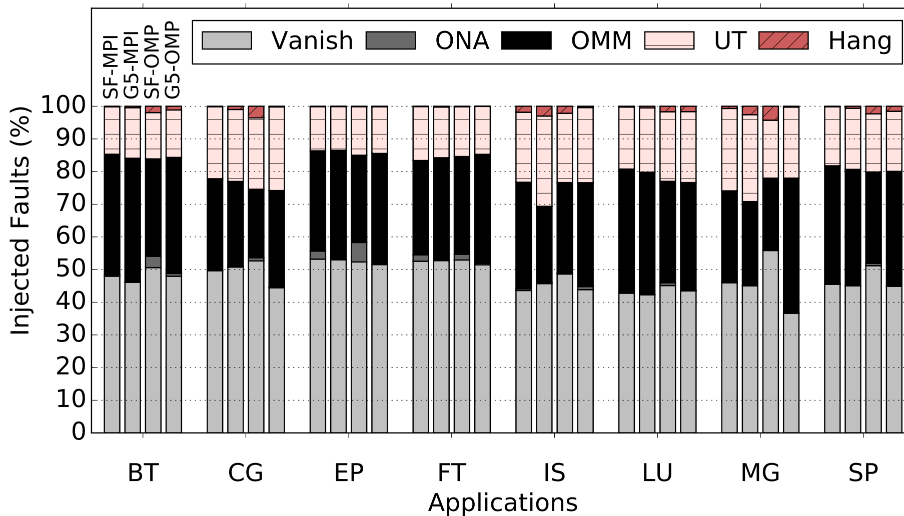

Fig. 3. Fault injection campaigns considering a quad-core processor using the SOFIA and the gem5.

implementation. Results were performed in a Quad-core Intel$^{(R)}$ Core$^{(TM)}$ i7–4790K CPU (32 GB DDR3 RAM) host machine. Fig. 4 shows the simulation performance of both the SOFIA and the gem5 frameworks as we increase the number of host cores from 1 to 4 considering the 36 fault injection campaigns.

Note that both (SOFIA and gem5) fault injection flows can perform and manage parallel fault injection campaigns. Considering the most significant benchmark (i.e., EP with 87 billion instructions) and using four host cores, SOFIA achieves up to 3,910 MIPS, which is approximately 325 times faster than the reference gem5 (12.52 MIPS). The obtained results also show that the simulation speed of SOFIA w.r.t gem5 fault injection implementation increases along with the application complexity, i.e., the more instructions, the higher is the SOFIA speedup.

6 Benchmark and TMR Case Study

To showcase SOFIA's applicability, this section presents a soft error analysis for a matrix multiplication (MM) kernel. First, we compare a sequential and a parallel implementation of the same kernel in Section 6.1. Leveraging from acquired information on the initial analysis, this work deploys two versions of the *Triple Modular Redundancy* (TMR) mitigation technique. In this regard, results are presented according to the classic TMR approach (MM-TMR — Section 6.2), and a refined TMR version (MM-TMR-I — Section 6.3).

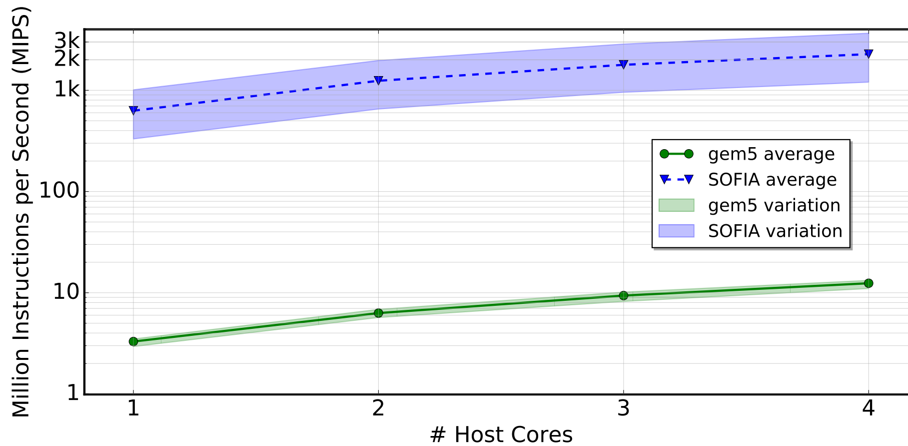

Fig. 4. SOFIA and gem5 simulation performance.

6.1 Sequential and Parallel MM

The first FI campaign deploys the MM kernel in two versions: (i) a sequential implementation, which uses a simple iteration-based algorithm, and (ii) a MM parallel kernel that relies on the Pthreads library to create two working threads. Fig. 5 shows the two MM implementations subjected to 8 fault campaigns of 8,000 fault injections each totaling 128,000 simulations, considering the six FI techniques:

1. Random registers **(A)**;
2. Physical memory **(B)**;
3. Virtual memory (VM) entire range **(C1)**;
4. VM code section **(C2)**;
5. VM data sections **(C3)**;
6. Result matrix **(D1)**;
7. Multiplication function object code **(E1)**;
8. Multiplication function lifespan **(F1)**.

The sequential MM under FI shows a higher occurrence of OMMs (Fig. 5) due to dirt registers used in the multiplication, which leads to a significant number of silent data corruptions. The underlying implementations are also susceptible to UTs as consequence of incorrect memory address computation caused by registers under fault influence, which may lead to errors such as *segmentation fault*. However, the parallel MM presents a substantially more significant number of UT when compared to the sequential version. This is explained because Pthreads scheduling algorithm increases the application control flow complexity, which might incur in more wrong address computation during the MM execution.

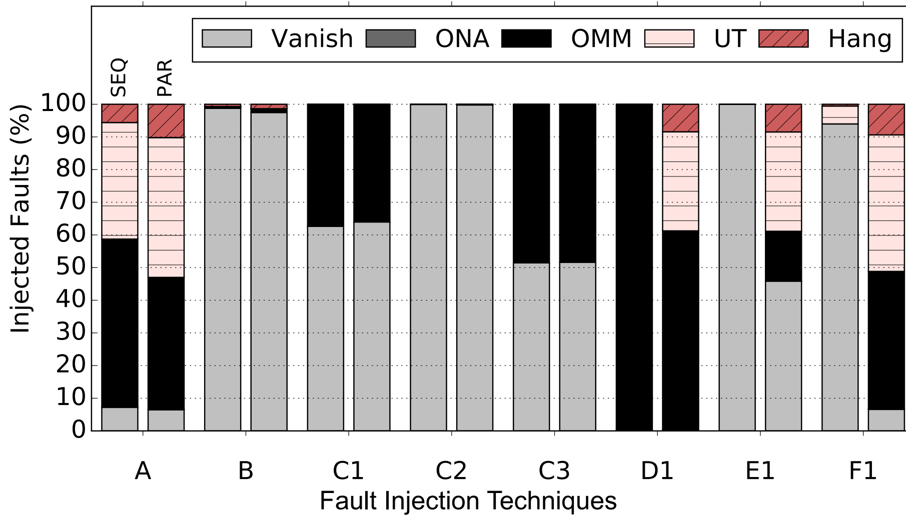

Fig. 5. Matrix Multiplication soft error vulnerability analysis. Sequential (SEQ) and parallel (PAR) for a single-core processor

6.2 Triple Modular Redundancy

As identified in the previous Section, the injection of bit-flips severely impacts on the matrix multiplication kernel operation due to its simplicity and high-density code. This work deploys the TMR to improve its reliability. Such a technique combines spatial (i.e., data duplication) and temporal (i.e., concurrent execution) replication to reduce the amount of detectable silent data corruptions. The new kernel (MM-TMR) executes three independent *parallel MM* instances (i.e., six working threads), enabling one incorrect execution to be masked by a voting process at the end of MM execution (i.e., a function vote the majority from the partial results). The TMR version corrects most of the errors originated from the input and output matrices. Nevertheless, the TMR implementation using the Pthread library increases the UT occurrence.

A custom soft error analysis step was included in the fault campaign flow to demonstrate the proposed tool efficiency. This additional module compares the four matrices (i.e., each TMR replica and the voter) alongside two other error classifications considering three possible outcomes:

1. All matrices are identical, in this case, the SOFIA classifies the detected error according to one of the five default classes (e.g., Vanish, UT, Hang). Note that OMM and ONA only occur if the result matrix is correct, and thus being considered benign errors in this context.
2. If one TMR matrix does not match the other replicas, the voter will mask the error and produce the correct result. Nevertheless, in this case, the simulation diverges in terms of the number of executed instructions from the faultless

run, which leads to a false-positive error (i.e., control flow error with incorrect memory) in traditional FI flows. The SOFIA classifies this execution context as **Corrected** to signal the appropriate behavior, i.e., even with the context mismatching the reference execution the final matrix is correct.
3. The third possible outcome originates from an incorrect voter execution (i.e., the three TMR matrices are identical and differ from the voter matrix) due to the FI being classified as **Voter Error**.

Table 2 describes 17 distinct FI scenarios targeting the MM-TMR, while Fig. 6 shows the results considering a single and quad-core ARM Cortex-A9 processor where each FI scenario comprises 8,000 faults. Register-based FI **(A, E, and F)** displays a considerable amount of UT (i.e., Linux OS segmentation faults in this context), around 40% due to the wrong address computation using registers under fault influence. In contrast, the memory-based technique errors depend on the stroke region, for example, targeting the 1 Gb physical memory (using technique B) would result to a minimal number of errors (i.e., masking rate of 99.95%) as the benchmark accesses a limited memory range (i.e., few dozen kilobytes). The complete VM range **(C1)** and data sections **(C3)** present a similar behavior as most of the faults hit the application 300-wide square matrices due to its size (i.e., each one possessing 360 kilobytes or 20% of application size). The code section **(C2)** contains, besides the application code, hundreds of unused Linux and C libraries functions added by the compiler, leading to greater masking rate. By individually targeting the matrix replicas **(D1–3)** we exercise the TMR main functionality resulting in an almost complete error coverage. The fault campaign **(D4)** leads to a 99.9% masking rate as the final result is composed of the voter function at the application end, which incurs in a narrow sensitive window (i.e., any faults previously present in this matrix are overwritten).

Table 2. Fault injection techniques targeting the TMR-based matrix multiplication.

Ref.	Target	Ref.	Target	Ref.	Target	Ref.	Target
A	Register file	D1	Matrix 1	E1	1st replication	F1	1st replication
B	Physical Memory	D2	Matrix 2	E2	2nd replication	F2	2nd replication
C1	Complete	D3	Matrix 3	E3	3rd replication	F3	3rd replication
C2	Code Section	D4	Matrix Result	E4	Voter	F4	Voter
C3	Data Sections						

Single and quad-core processors show a similar rate of correct results (i.e., vanish, SDC, and corrected) when targeting the function object code **(E1–4)**. However, their composition diverges while the single-core processor presents a more significant SDC rate (i.e., the MM result is correct with silent data corruptions on the memory) the multicore system displays a larger masking rate.

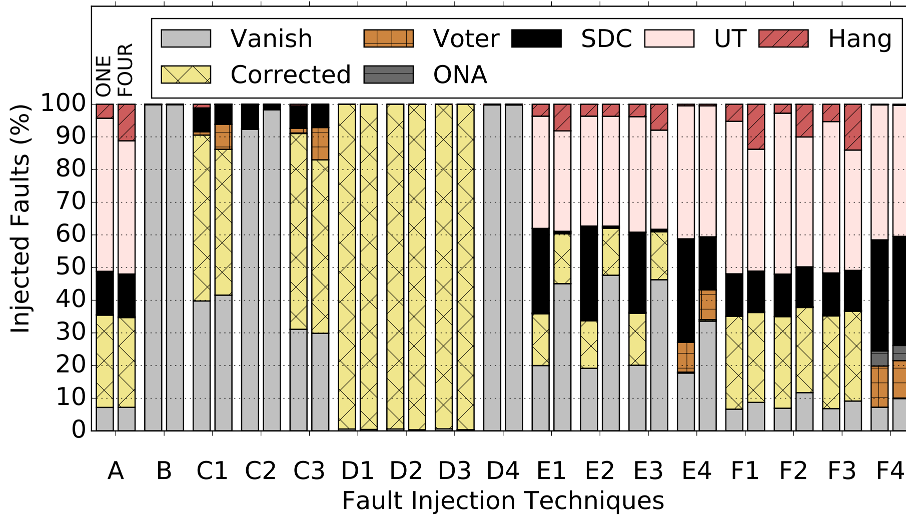

Fig. 6. Matrix Multiplication soft error vulnerability analysis. MM-TMR for a single and quad-core processor

Further, the multicore system reveals a higher number of *Hangs* due to the longer and higher executions of the PTHREAD scheduling policy leading to unrecoverable control flow. Random register **(A)** and Lifespan **(F1–3)** techniques show similar behaviors under FI as the MM application spends 95% on those multiplication functions. Directly targeting the voter function show a behavior not seen when targeting the complete application with random faults due to its short execution time, and thus, demonstrating the necessity of more detailed FI framework. Subjecting the voter code **(E4)** and lifespan **(F4)** to FI causes an erroneous matrix voting, which is a severe error in this context.

6.3 Improving the Triple Modular Redundancy

The initial MM-TMR solution provides complete coverage to fault injections for the replicated data (i.e., the partial matrices) while the control flows still prone to unexpected terminations. By using the promoted framework, it is possible to pinpoint the significant UT cause as OS segmentation faults in one of the thread replicas that terminates the complete application even if the other replicas had not experienced any errors. To mitigate this issue, we modified the application algorithm to include a segmentation handler for each replication, and consequently, the improved MM-TMR (MM-TMR-I) finishes correctly even if one of the replicas generates an OS segmentation fault. The experiments displayed in Fig. 7 reproduce the 17 FI scenarios mentioned above for the MM-TMR-I version using the single and quad-core processors.

The MM-TMR-I improves the MM kernel reliability by achieving of up to 90% of coverage (i.e., with correct final results) in contrast to the 50% of the

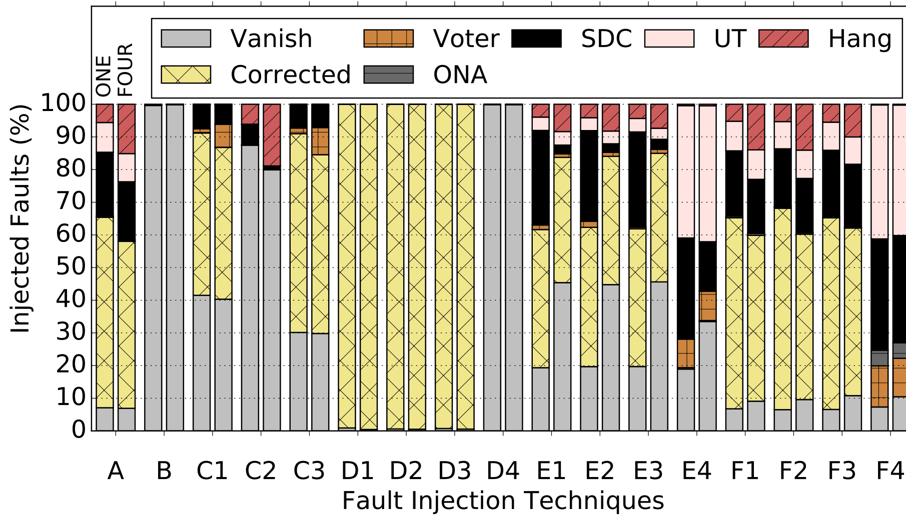

Fig. 7. Matrix Multiplication soft error vulnerability analysis. MM-TMR-I for a single and quad-core processor

traditional TMR considering register-based fault injections targeting the replicas working threads. FI techniques **(D4, E4, and F4)** targeting the voter function and data remains unchanged without any modification being made in its code.

7 Automotive Case Study

While benchmarks can help to validate the proposed tool, a real-world application provides more useful insights in terms of functionality and applicability of the promoted techniques. To demonstrate the proposed fault injection techniques soft error analysis capabilities, we select a complex software stack to be our second case study. The experimental setup comprises a commercially available Arm Cortex-A9 processor, an unmodified Linux kernel (3.7), the Darknet framework [17] for neural networks with NNPACK float-pointing acceleration package [18], and the YOLOv3 [19] real-time object detect system. Object detection in real-time is a paramount research topic for both academia and industry to achieve SAE Level 3+ autonomous vehicles. To supply a realistic scenario, this work adopts the KITTI Vision Benchmark Suite [20], which uses challenging computer vision sets to extract data from a real-world urban environment. The YOLO algorithm takes KITTI suite images, each one measuring $1,242 \times 375$ pixels, and outputs a list of detected objects and their *confidence's degree in percentile*.

After some experiments, we identified that Cho's classification falls short when it comes to defining object detection algorithms, which produce outcomes based on probabilities, not on just absolute "yes" or "no". To improve the soft

error analysis, a bespoke classification was defined according to the following conditions: **correct output** when the outputs (golden and fault injection) match, i.e., true vanished; **incorrect** if at least one object or probability is different. [5] Further, the incorrect result can be divided into **incorrect probability** when all objects are correct, but at least one has a different percentile of confidence—in most cases this would not influence the action of an autonomous vehicle; **wrong detection**, i.e., false positive or missing of an object; and **no prediction**, if no object is in the image. The last two can represent a life-threatening failure, by forcing a full stop in a highway (false positive, an object in the path) or crash (missing of an object).

We subject the selected application to multiple fault scenarios employing the fault injection techniques (see Section 4) targeting distinct software components alongside a customized error classification module. Each technique covers different aspects of the application considering its variables and critical functions in an isolated manner, demonstrating the importance of providing engineers with appropriate means that enable to identify not only the soft error occurrence but also the specific software characteristics that contribute more directly to their appearance.

The fault scenarios consider commonplace techniques in the literature (**(A)** and **(B)**) as reference to evaluate the benefits and drawbacks of proposed fault injection techniques (Section 4): application virtual memory **(C)**, two variables **(D1, D2)**, and four functions—code **(E)** and lifespan **(F)**. The technique **(C)** is used to target the application data structures, excluding kernel and other workloads from the system. The select two variables **(D1, D2)** for this experiment are: `colors` and `windows`; which are globally available and are used to plot the image overlay that delimit each detected object. The chosen functions represent the target application critical portions: detection and probability calculation of objects, `add_bias`, and `scale_bias`; convolutional layer creation of the neural network `make_convolutional_layer`; and matrix multiplication, as is orthogonal to this and other applications, `compute_matrix_multiplication`. Each of the individual 13 scenarios comprises three fault injections campaigns (with different input images), each campaign has 800 simulations with a single bit-flip, totaling 31,200 fault injections.

7.1 Result Analysis

This work uses traditional fault injection techniques such as targeting general propose registers (GPRs) and physical memory as a baseline for comparison. This section aims to show how these techniques can mask certain behaviors. We want to show that bespoke techniques can better guide developers towards a more efficient soft error analysis that considers the critical application elements, which reduces the number of simulations needed to extract relevant error/failure-related data.

[5] All input images used in this work have six to ten objects detected in the reference execution.

Register File Register file fault injection campaigns display a higher masking effect in small applications, as is the case for many benchmarks. For instance, the ArmV7 architecture has 48 registers (16 GPRs, 32 floating-point) and presents a twofold consequence: (i) bit-flips in unused registers lead to ONA as the fault remains untouched until the application ends—common in short benchmarks as they use a reduced register subset, and (ii) the usage of floating-point registers depend on the application and compiler support. Even considering a real-world application, the fault injection technique **(A)** has a masking rate of 62.32%.

The customized classification (Section 3.4) combined with the techniques allows more insightful investigations. While 37.68% of the campaign **(A)** has some kind of lingering difference w.r.t. Golden execution (Fig. 8), only 1.27% had their outcome affected (Figs. 9 and 10). The high number of masking and low incorrect outcomes attest to the reliability of the application. However, one can also say that 98.73% of the fault injections were "not helpfull" if the goal is to observe the behavior of the application in the presence of error.

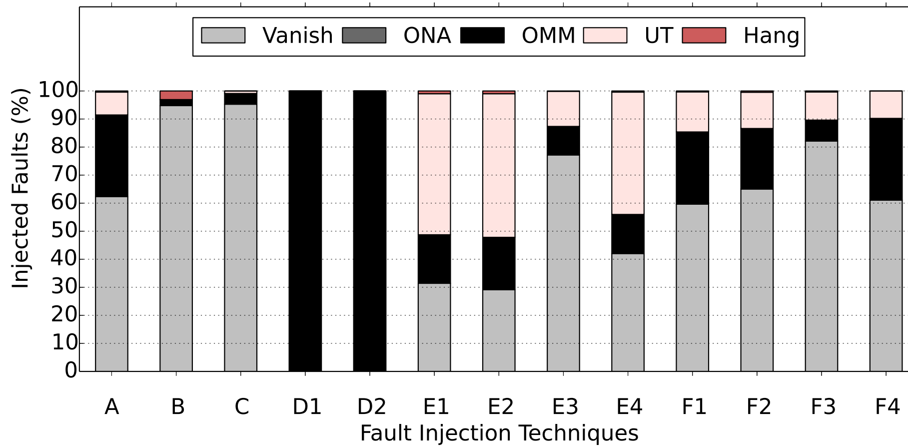

Fig. 8. Results for fault injection campaigns considering the classical (i.e., Cho's) Fault Injection classification.

Physical Memory Physical memory fault injection masking is higher than the ones targeting the register file. There are 42.8×10^{12} targets (i.e., the number of instructions multiplied by either the register, or the memory bit count) for **(A)** versus 65.7×10^{15} in **(B)**, and thus the Vanish rate increases to 95.09% (and 100% correct outcomes) due to the number of possible targets for each bit flip. This order of magnitude difference severely impacts the soft error analysis cost since longer fault campaigns are needed to extract a meaningful amount of error/failure-related data. A large number of campaigns becomes impractical as the complexity and size of the application increases. With this experimental

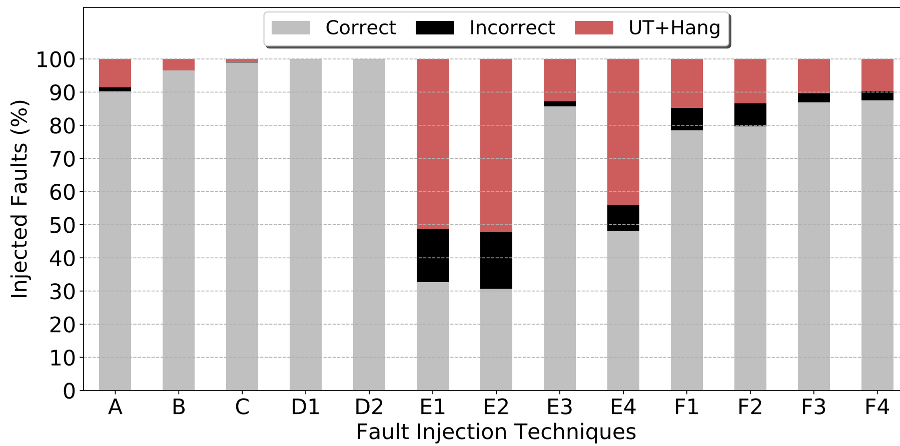

Fig. 9. Results for fault injection campaigns considering this work case study custom fault classification.

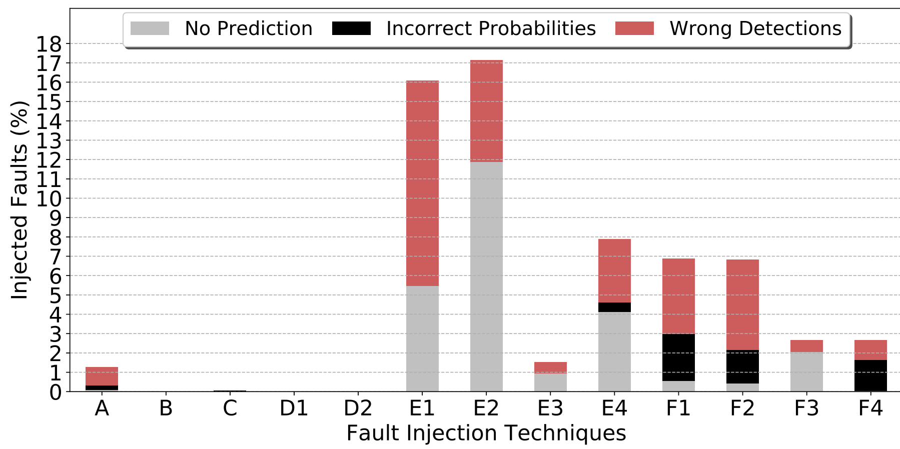

Fig. 10. Results for fault injection campaigns considering the only the incorrect results from Fig. 9.

setup (Section 7), each fault injection takes on average 10 minutes to execute, over 5,200 hours for all campaigns in this section.

This work main objective is to explore the erroneous application behavior and not establish the number of faults when exposing the target application to a particular fluence. Bespoke techniques allow narrowing the focus on critical aspects of the application, e.g., functions, variables, data structures, and more. Further, Figs. 8 to 10 also show results for campaign targeting two variables, four functions (code and lifespan).

Application Virtual Memory The target Linux kernel (v3.7) allocates 488 MB for this application base virtual memory, mostly comprised of libraries and reserved space. This context yields 20.9×10^{15} possible fault targets, a similar order of magnitude when compared to physical memory. Thus resulting in 95.29% of the fault injection resulted in a **Vanish** and all results were **correct**.

Application Variables and Data Structures The need for the auxiliary classification becomes even more evident for campaigns **(D1, D2)**, where 100% **OMM** leads to no **incorrect** results. Targeted variables are used during the construction of the visual output. Meaning that the output image with the box drawn is incorrect (e.g., position, color) but the algorithm output (i.e., classification) is correct.

Function Code Fault injection on the assembly code will change the function behavior for future calls. Fig. 8 shows that **E (1, 2, and 4)** have a high susceptibility to **UT**, as these functions rely on `for` loops which are sensible to wrong address (e.g., reading after a vector), and control flow (e.g., number of iterations). Further, the predominant type of prediction errors (Fig. 10), is **no prediction** and **wrong detections**. With a 32.28% reduction in the Vanished (technique **(A)** vs **(E2)**), there is more relevant data regarding erroneous application behavior. However, **E3** that creates the layers of the neural network (NN) that perform the object detection, leads to less **UT** and **incorrect** outcomes.

Function Lifespan Even though this technique may lead to more masked faults than **(A)**, it has at least twice (at most six times) the number of incorrect/wrong results. Fig. 8 shows that the occurrence of **UT** is lower while **OMM** is higher than code fault injection, when considering the lifespan of a function (i.e., fault injection injected in register file only when the function is running). Considering the prediction errors (Fig. 10), the trace of the target functions are more susceptible to change the objects detected, thus **F1** is a prime candidate for fault tolerance techniques.

Techniques Evaluation/Remarks From this experimental setup, and case study, it is possible to observe the importance of a targeted fault injection approach to finding critical elements of a given application. While fault injection in

the register file shows the average behavior considering all functions, variables, and routines; it is possible to see that some variables do not change the outcome (Section 7.1). Furthermore, this type of analysis can improve the efficiency of the fault injection campaign. Techniques targeting Function Code and Lifespan can reduce the number of masked faults in up to 32.28%, Fig. 8. The cases where the number of vanished increased, the analysis still has more meaningful results than **(A)** as the number of **incorrect** outcomes is higher, Fig. 10. This type of analysis can show the developer where the cost$\times$benefit of implementing a fault tolerance technique (e.g., TMR, ECC) is more profitable according to the propose of the application. Indeed, if the most crucial aspect of the application is to provide the correct object independent of its degree of confidence, protecting the function code (e.g., ECC) may yield better results then its execution (e.g., TMR for function lifespan).

8 Conclusion

This paper proposed the SOFIA framework, which supports detailed soft error vulnerability investigation considering complex software stacks in a non-intrusive manner. By combining novel fault injection techniques and robust analysis, the presented tool can help engineers to uncover otherwise hidden soft errors during early design space explorations. Two case studies with a resource intensive benchmark running on a quad-core platform showcase the kind of discovery and how SOFIA can aid during the early phase of development. Further, a real-world automotive application shows that SOFIA's exploration capabilities can scale beyond artificial workloads, thus validating the framework.

References

1. Baumann, R.: Radiation-Induced Soft Errors in Advanced Semiconductor Technologies. IEEE Transactions on Device and Materials Reliability **5**(3) (2005) 305–316
2. Marc Snir, Robert W Wisniewski, Jacob A Abraham, Sarita V Adve, Saurabh Bagchi, Pavan Balaji, Jim Belak, Pradip Bose, Franck Cappello, Bill Carlson, Andrew A Chien, Paul Coteus, Nathan A DeBardeleben, Pedro C Diniz, Christian Engelmann, Mattan Erez, Saverio Fazzari, Al Geist, Rinku Gupta, Fred Johnson, Sriram Krishnamoorthy, Sven Leyffer, Dean Liberty, Subhasish Mitra, Todd Munson, Rob Schreiber, Jon Stearley, Eric Van Hensbergen: Addressing Failures in Exascale Computing. The International Journal of High Performance Computing Applications **28**(2) (May 2014) 129–173
3. Kaliorakis, M., Tselonis, S., Chatzidimitriou, A., Foutris, N., Gizopoulos, D.: Differential Fault Injection on Microarchitectural Simulators. In: International Symposium on Workload Characterization, Atlanta, GA, USA, IEEE (October 2015) 172–182
4. Didehban, M., Shrivastava, A.: nZDC: A Compiler Technique for Near Zero Silent Data Corruption. In: Design Automation Conference, Austin, Texas, ACM Press (2016)

5. Tanikella, K., Koy, Y., Jeyapaul, R., Kyoungwoo Lee, Shrivastava, A.: gemV: A validated toolset for the early exploration of system reliability. In: International Conference on Application-Specific Systems, Architectures and Processors, London, United Kingdom, IEEE (July 2016) 159–163
6. Bandeira, V., Rosa, F., Reis, R., Ost, L.: Non-intrusive Fault Injection Techniques for Efficient Soft Error Vulnerability Analysis. In: International Conference on Very Large Scale Integration (VLSI-SoC), Cuzco, Peru, IEEE (October 2019) 123–128
7. Hari, S.K.S., Adve, S.V., Naeimi, H., Ramachandran, P.: Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults. In: International Conference on Architectural Support for Programming Languages and Operating Systems, New York, NY, USA, Association for Computing Machinery (2012) 123–134
8. Magnusson, P., Christensson, M., Eskilson, J., Forsgren, D., Hallberg, G., Hogberg, J., Larsson, F., Moestedt, A., Werner, B.: Simics: A Full System Simulation Platform. Computer **35**(2) (Feb./2002) 50–58
9. Martin, M.M.K., Sorin, D.J., Beckmann, B.M., Marty, M.R., Xu, M., Alameldeen, A.R., Moore, K.E., Hill, M.D., Wood, D.A.: Multifacet's General Execution-Driven Multiprocessor Simulator (GEMS) Toolset. ACM SIGARCH Computer Architecture News **33**(4) (November 2005) 92
10. Geissler, F.d.A., Kastensmidt, F.L., Souza, J.E.P.: Soft Error Injection Methodology Based on QEMU Software Platform. In: Latin American Test Workshop, Fortaleza, Brazil, IEEE (March 2014)
11. Guan, Q., BeBardeleben, N., Wu, P., Eidenbenz, S., Blanchard, S., Monroe, L., Baseman, E., Tan, L.: Design, Use and Evaluation of P-FSEFI: A Parallel Soft Error Fault Injection Framework for Emulating Soft Errors in Parallel Applications. In: International Conference on Simulation Tools and Techniques, Prague, Czech Republic, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering) (2016) 9–17
12. Guan, Q., Debardeleben, N., Blanchard, S., Fu, S.: F-SEFI: A Fine-Grained Soft Error Fault Injection Tool for Profiling Application Vulnerability. In: International Parallel and Distributed Processing Symposium, Phoenix, AZ, USA, IEEE (May 2014) 1245–1254
13. Rosa, F., Kastensmidt, F., Reis, R., Ost, L.: A Fast and Scalable Fault Injection Framework to Evaluate Multi/Many-Core Soft Error Reliability. In: International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems, Amherst, MA, USA, IEEE (October 2015) 211–214
14. Feng, S., Gupta, S., Ansari, A., Mahlke, S.: Shoestring: Probabilistic Soft Error Reliability on the Cheap. In: International Conference on Architectural Support for Programming Languages and Operating Systems. ASPLOS XV, New York, NY, USA, Association for Computing Machinery (2010) 385–396
15. Cho, H., Mirkhani, S., Cher, C.Y., Abraham, J.A., Mitra, S.: Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design. In: Design Automation Conference, Austin, Texas, ACM Press (2013)
16. Bailey, D.H., Schreiber, R.S., Simon, H.D., Venkatakrishnan, V., Weeratunga, S.K., Barszcz, E., Barton, J.T., Browning, D.S., Carter, R.L., Dagum, L., Fatoohi, R.A., Frederickson, P.O., Lasinski, T.A.: The NAS Parallel Benchmarks—Summary and Preliminary Results. In: Conference on Supercomputing, Albuquerque, New Mexico, United States, ACM Press (1991)
17. Redmon, J.: Darknet: Open Source Neural Networks in C. https://pjreddie.com/darknet/ (2016)

18. Dukhan, M.: Maratyszcza/NNPACK (February 2020)
19. Redmon, J., Farhadi, A.: YOLOv3: An Incremental Improvement. arXiv:1804.02767 [cs] (2018)
20. Menze, M., Geiger, A.: Object Scene Flow for Autonomous Vehicles. In: Conference on Computer Vision and Pattern Recognition, Boston, MA, USA, IEEE (June 2015) 3061–3070