Using Machine Learning Techniques to Evaluate Multicore Soft Error Reliability

Felipe Rocha da Rosa, *Member, IEEE*, Rafael Garibotti, *Member, IEEE*, Luciano Ost, *Member, IEEE*, and Ricardo Reis, *Senior Member, IEEE*

***Abstract*— Virtual platform frameworks have been extended to allow earlier soft error analysis of more realistic multicore systems (i.e., real software stacks and state-of-the-art ISAs). The high observability and simulation performance of underlying frameworks enable to generate and collect more error/failure-related data, considering complex software stack configurations, in a reasonable time. When dealing with sizeable failure-related data sets obtained from multiple fault campaigns, it is essential to filter out parameters (i.e., features) without a direct relationship with the system soft error analysis. In this regard, this paper proposes the use of supervised and unsupervised machine learning techniques, aiming to eliminate non-relevant information as well as identify the correlation between fault injection results and application and platform characteristics. This novel approach provides engineers with appropriate means to investigate new and more efficient fault mitigation techniques. The underlying approach is validated with an extensive data set gathered from more than 1.2 million fault injections, comprising several benchmarks, a Linux OS and parallelization libraries (e.g., MPI and OpenMP), as well as through a realistic automotive case study.**

***Index Terms*— Multicore systems, fault injection, soft error, virtual platforms, machine learning techniques.**

I. Introduction

WHILE commercial multicore processors based on 10 nm process node are already available, processing components manufactured in 5 nm are expected to be released in the market by, approximately, the end of the decade. Such processors are likely to be integrated into many electronic computing systems from a diverse range of industrial sectors, including medical, automotive, and high-performance computing (HPC). The increasing number of internal elements (e.g., cores, memory cells), coupled with the high clock frequency operation of multicore processors is making the aforementioned systems more vulnerable to radiation-induced soft errors [1]. The occurrence of soft errors can lead to failures of critical parts of a system, which might ultimately incur in financial or human life losses. Thus, assessing and mitigating the occurrence of soft errors in such systems is key to accomplish their reliable and efficient operation.

The increasing software and hardware complexity of such systems imposes exploration challenges, including: (*ch1*) conduct a large number of fault injection campaigns within a reasonable time; (*ch2*) provide engineers with detailed observation of system's behavior in the presence of faults; and (*ch3*) identify relationships or associations between application profiling and specific platform parameters in large data sets resulting from the fault campaigns. Aiming to overcome the challenges *ch1* and *ch2*, researchers are incorporating fault injection capabilities into virtual platform (VP) frameworks [2]–[8], enabling the detailed observation and analysis of complex software stacks and multicore architectures under the presence of faults at early design phases.

The main *contribution* of this paper relies on the exploration of supervised and unsupervised machine learning (ML) techniques that can be used to identify the correlation between fault injection results and application and platform microarchitectural characteristics. The other contributions of this work are the following:

- Proposal of a completely automated soft error assessment flow that includes simulation-based fault injection, identification of typical and critical soft errors, and in-depth correlation analysis of detected soft errors and target system architecture using ML techniques;
- Extensive multicore soft error evaluation by using realistic Linux kernel, instruction set architectures (ISAs) and standard parallelization libraries, considering several benchmarks;
- Validation of the proposed soft error assessment flow through a realistic automotive case study.

The rest of this paper is organized as follows. Section II presents related works in virtual platform fault injection simulators. Section III details two fault injection frameworks that are employed in this study. Next, Section IV presents the proposed soft error assessment flow along with the adopted machine learning techniques. Section V explores the promoted automated tool capability, revealing insights that can be obtained at each exploration phase to improve system soft error reliability. Following, Section VI shows the effectiveness of our soft error assessment flow through a realistic automotive case study. Finally, Section VII points out conclusions.

Manuscript received July 29, 2018; revised January 11, 2019 and February 25, 2019; accepted March 11, 2019. Date of publication April 17, 2019; date of current version May 15, 2019. This paper was recommended by Associate Editor J. Apolinario. *(Corresponding author: Luciano Ost.)*

F. R. da Rosa and R. Reis are with the Institute of Informatics, Federal University of Rio Grande do Sul, Porto Alegre 90040-060, Brazil (e-mail: frdarosa@inf.ufrgs.br; reis@inf.ufrgs.br).

R. Garibotti was with the Institute of Informatics, Federal University of Rio Grande do Sul, Porto Alegre 90040-060, Brazil. He is now with the School of Technology, Pontifical Catholic University of Rio Grande do Sul, Porto Alegre 90619-900, Brazil (e-mail: rafael.garibotti@pucrs.br).

L. Ost is with Loughborough University, Loughborough LE11 3TU, U.K. (e-mail: l.ost@lboro.ac.uk).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2019.2906155

TABLE I
RELATED WORKS IN FAULT INJECTION FRAMEWORKS DEVELOPED ON THE BASIS OF VIRTUAL PLATFORMS (VPS)

Works	Year	Virtual Platforms	Operating Systems	Number of Benchmarks	Programming Model	Number of Fault Injections	Target Architecture Single-core	Target Architecture Multicore	Use of ML
Hari *et al.* [2][3]	2014	Simics + GEMS	OpenSolaris	12	Serial	$3.2 * 10^7$	✓		
Geissler *et al.* [4]	2014	QEMU	RTEMS	4	Serial	$3.2 * 10^4$	✓		
Kaliorakis *et al.* [5]	2015	MARSS + gem5	None	10	Serial	$3.0 * 10^5$	✓		
Tanikella *et al.* [6]	2016	gem5	None	10	Serial	$3.3 * 10^4$	✓		
Guan *et al.* [7]	2016	gem5	gem5 (Syscall Mode)	7	Serial and MPI	$1.4 * 10^5$	✓		
Didehban *et al.* [8]	2016	gem5	None	11	Serial	$7.2 * 10^4$	✓		
Khosrowjerdi *et al.* [9]	2018	QEMU	None	2	Serial	$6.0 * 10^2$	✓		✓
This work	**2018**	**OVPsim + gem5**	**Unmodified Linux**	**30**	**Serial, OpenMP and MPI**	$1.2 * 10^6$	✓	✓	✓

II. REVIEW OF FAULT INJECTION APPROACHES USING VIRTUAL PLATFORMS

Virtual platforms simplify the development of fault injection modules and the subsequent analysis due to their design flexibility (e.g., several processor models available) and debugging capabilities (e.g., GDB support). Table I summarizes related works in virtual platform fault injection simulators. With the exception of [7] that includes benchmarks described in MPI, reviewed approaches neither consider parallel programming libraries nor multicore processors on their experiments. Further, the majority of these works consider either simple in-house (e.g., [4]) and bare metal applications (e.g., [8]) or small set of benchmarks (e.g., [5], [6], [9]), where only a specific ISA is considered. While the approaches presented by Hari *et al.* [2], [3] propose a hybrid simulation framework for SPARC core using Simics [10] and GEMS [11] simulators, remaining works rely on a single virtual platform simulator.

Machine Learning has been employed in different domains to recognize patterns and predict how a given system would react to unexpected circumstances. In the context of soft error assessment, Khosrowjerdi *et al.* [9] use ML to reverse engineer models of the system under evaluation, aiming to reduce the total number of required fault injections. To validate the ML-based pruning mechanism, QEMU was extended to enable the injection of faults in two automotive applications. While this approach focus on the reduction of total time to complete the fault injection campaigns, our approach aims at correlating large subsets of application profiles and architecture characteristics with fault injection results in order to pinpoint the most relevant parameters/traces on the target system. Vishnu *et al.* [12] evaluate the impact of multi-bit memory errors, both permanent and transient, on HPC applications. This work considers eight different ML algorithms (e.g., support vector machines, k-Nearest neighbors, three distinct decision trees), comparing their predictions (i.e., the error probability) with the ground-truth (i.e., fault injection results). In this work, training sets consist of fault injections targeting the application data structures while executing on a supercomputer. Proposed fault injection and analysis process are tightly coupled with the application, requiring an excellent understanding of its behavior and execution. ML techniques are also employed to create fault propagation models in [13]. This work considers the propagation of soft errors on large-scale MPI applications executing on up to 1000 cores. To inject faults and check for errors, this work introduces additional instructions in the application code using the LLVM Intermediate Representation (IR). Further, the LLVM instrumentation tracks the error propagation through several MPI process by monitoring communication messages, load-store operations, and function calls. This information is then applied to create fault propagation models using ML techniques, such as linear regression.

Another direction is the exploitation of compiler optimizations on application reliability. Ashraf *et al.* [14] exploit the effects of compiler optimizations on the reliability of HPC applications. Their results suggest that highly-optimized code is generally more vulnerable than unoptimized code. However, they not proposed any tools. Unlike, Narayanamurthy *et al.* [15] use a genetic algorithm to select the best optimization flag for a particular application. The proposed tool correlates a large data set of information from any source, related or not directly to the target application execution, and filters the most significant parameters for a given input. However, the paper specifies neither the compiler nor the particular optimization flags used on the tool. Although the evaluation of compiler parameters is not the main focus of this work, this exploration validates our proposed soft error assessment flow and shows engineers the advantage of having appropriate means that are able to investigate new and more efficient fault mitigation techniques.

Our contribution distinguishes from the previous works in four main aspects:

- First, this is the first work to use ML techniques to identify the correlation between fault injection results and multicore multi-parameters (i.e., application and platform characteristics), which is completely ignored in previous works;
- Second, the multicore soft error analysis requires complementary modeling and simulation mechanisms to manage other aspects such as resource sharing, memory allocation, and data dependencies. These important aspects are not considered in previous work;
- Third, this work employs not only realistic soft stacks including unmodified Linux kernel but also standard parallelization libraries, considering up to 29 benchmarks from both embedded and HPC domains. While reviewed works typically report fault injection campaigns with thousands of fault injections (e.g., [5], [7]), this work addresses an extensive soft error evaluation with millions of fault injections;
- Fourth, in addition to the 29 benchmarks, we also employ a realistic automotive case study to demonstrate the effectiveness of our proposed soft error assessment flow.

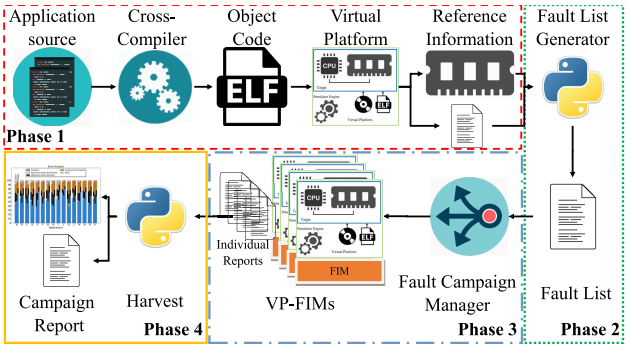


Fig. 1. Fault injection campaign flow applicable to both virtual platforms, gem5 and OVPsim.

III. Adopted Virtual Platform Fault Injector Framework Infrastructures

The growing susceptibility of multicore systems to soft errors necessarily calls for novel cost-effective tools to assess the soft error resilience of underlying systems early in the design phase. This paper employs two flexible fault injection (FI) frameworks, developed on the basis of two virtual platforms: OVPsim-FIM [16], and gem5-FIM [17]. The main difference between both FI frameworks is the simulation performance. A single execution of OVPsim-FIM, which is developed on top of the instruction-accurate OVPsim [18], can achieve simulation performance of up to thousands MIPS. In turn, event-driven simulators, such as the gem5 simulator [19], can typically report best-case simulation performances of up to 2-3 MIPS. The original gem5 distribution is devoted to microarchitecture exploration, thus gem5-FIM allows the user to inject faults in several components such as processor pipeline, cache control registers, among others. Both frameworks integrate a set of fault injection and simulation techniques, allowing fast soft error susceptibility exploration considering not only state-of-the-art multicore processor architectures (e.g., big.LITTLE) but also different ISAs (e.g., ARMv7, ARMv8). Both gem5-FIM and OVPsim-FIM follow a four-phase fault injection flow described as follows.

A. Workflow

Figure 1 shows the adopted four-phase fault injection flow.

In the *first phase*, Golden Execution, the target architecture is simulated in the absence of faults to extract the reference system behavior (i.e., the context of the registers and final memory state), while *phase two* creates a fault target list. In the *third phase*, each application behavior running under fault injection is compared to the golden execution in order to detect possible arising errors, including target system (e.g., soft stack, processor model) misbehavior regarding the number of executed instructions, register's context, and memory state. Lastly, *phase four* assembles all individual reports to create a single database.

B. Fault Model

The developed fault injection framework emulates the occurrence of single-bit-upsets (SBUs) by injecting one flipped bit in a single register or memory address during the execution of a given soft stack. In our setup, SBUs target only storage elements due to its higher susceptibility to radiation events when compared with logic elements [20]. The 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. Fault injections occur during the target application lifespan, i.e., the OS startup is not subject to faults but 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 correlate with adopted OS components or API libraries.

C. Fault Classification

We adopted Cho *et al.* [21] classification, which categorizes fault injection outcomes into five groups: *Vanished*, no fault traces are left; *Application Output Not Affected* (ONA), the resulting memory is not modified, however, one or more remaining bits of the architectural state is incorrect; *Application Output Mismatch* (OMM), the application terminates without any error indication, and the resulting memory is affected; *Unexpected Termination* (UT), the application terminates abnormally with an error indication; and *Hang*, the application does not finish requiring a preemptive removal.

D. Simulation Infrastructure

A fault injection campaign may comprise thousands of simulations, demanding a significant computational effort, which restricts the soft error analysis to small scenarios as reported in Section II. Developed simulation infrastructure (SI) includes a couple of features to boost up the fault injection simulation: checkpoint and distributed simulation [16], [17].

1) Checkpoint: A fault injection requires unnecessary code re-execution to provide the appropriated software context (i.e., fault injection time). The checkpoint technique reduces this redundant simulation by periodically collecting the platform components context during a faultless execution. The simulation infrastructure automatically calculates intervals between checkpoints, creates the data containers, and restores the appropriated one according to each fault injection time.

2) Distributed Simulation: This work extended the traditional fault injection flow (Figure 1) to take advantage of the multiprocessing capabilities provided by a supercomputer center, enabling users to conduct large fault campaigns and investigations. The simulation flow, shown in Figure 2, manages the creation, submission, and harvest of jobs (i.e., group of simulations) without any particular user configuration.

The *local* fault injection flow, which comprises phases one and two (Figure 1) is subdivided into four steps (A-D, blue in Figure 2). First, the SI (A, in Figure 2) compiles the application and submits a new job (B) with the golden execution (i.e., execution without faults). The golden execution (C) is performed in a node designated by the HPC manager system, and both the collected reference information (i.e., the context of the registers and final memory state) and the

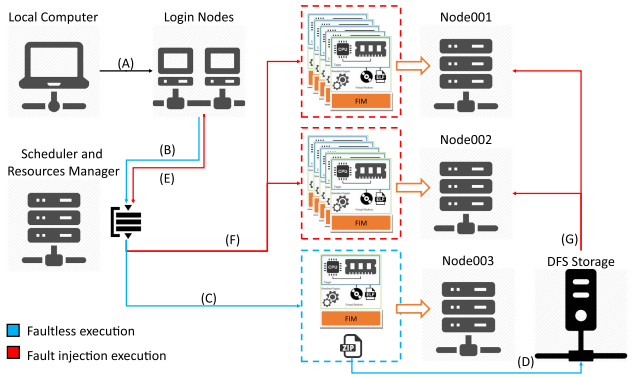


Fig. 2. Distributed fault injection campaign flow.

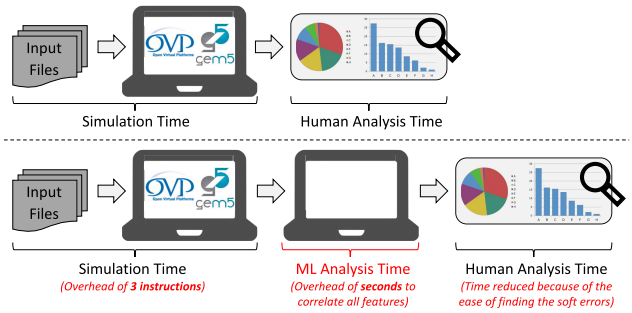


Fig. 3. Fault injection overhead.

fault list are saved (D, in Figure 2) in the distributed file system (DFS). Steps (E-G) comprises the distributed fault injection process (red) across hundreds of parallel executions. Jobs responsible for the fault injection are submitted to the scheduler queue (E, Figure 2). According to the system resource availability, those jobs are independently assigned to nodes (F). During each job initialization, the compressed file (G) on step "D" is locally copied and extracted. The entire fault injection and result analysis occur locally in each node. All generated individual reports are later merged into a final fault injection report.

Developed extension relies on commonly available HPC tools, such as Torque [22], ensuring compatibility with most of the available HPC environments. The distributed SI scalability has been validated through several scenarios, some of which with up to 1500 parallel simulations.

E. Fault Injection Overhead

The fault injection process does not change the target application source, instead, the fault injector interrupts the simulator a single time to change a single bit in the application execution. The underlying interruption occurs a single time during a simulation, having an unnoticeable impact on the overall simulation time whenever compared to an unmodified application execution (i.e., reference run).

Figure 3 shows the standard flow (top) and our soft error assessment flow (bottom). The overhead of the fault injection approach can be divided into two parts.

First, the overhead related to the increased simulation time. This overhead can be discarded because the simulations may have billions of instructions (e.g., VO application has up to 250 billion instructions) and 3 instructions will not affect the simulation time at all. Second, the addition of the ML analysis time, which takes seconds to correlate all features. Considering that the gem5 and OVPsim simulation times of VO application are 36 hours and 1 hour, respectively, this overhead is also marginal. Furthermore, the human analysis time required to understand the effects of the fault injection is reduced because our soft error assessment flow produces bespoke reports and charts that help in this process.

Further, it is important to mention that the proposed ML-based soft error correlation toolset was developed aiming to enable the processing of large amount of inputs collected from multiple sources, such as the adopted VP simulators. This work used such simulators because we believe it is essential to enable the execution of real software stacks considering, at least, basic architecture characteristics (e.g., ISAs, etc) when assessing the soft error reliability. However, other fault injection approaches can also be used to generate the necessary inputs to our toolset.

IV. Proposed Multicore Soft Error Assessment Flow Using Machine Learning Techniques

Figure 4 shows the traditional application soft error evaluation cycle (a, b, and e), where first, a software engineer describes the target application (a). The second phase (Figure 4b) requires a comprehensive soft error vulnerability investigation through one or more fault campaigns. Using this information, the software engineer may recommend modifications to the application, taking into account the target system (e) requirements.

The high simulation performance of both OVPsim-FIM and gem5-FIM allow users to generate and gather a significant amount of error/failure-related data, considering complex software stack configurations, as reported in Section III. The high-volume and the great variety of collected data lead to more reliable soft error analysis. However, the more data, the more difficult and time-consuming is to identify relationships or associations between fault injection campaign results and platform parameters. Aiming to overcome this challenge, we incorporated two additional steps (Figure 4c and d) to the traditional soft error evaluation flow: (*i*) application profiling, and (*ii*) soft error correlation analysis.

The *application profiling* (Figure 4c) facilitates the access to raw microarchitectural information, enabling software engineers to search for specific parameters of interest (e.g., number of loads and stores) during initial explorations. In turn, in the *soft error correlation analysis* phase, users can use machine learning techniques (e.g., linear regression) to speed-up the soft error evaluation process by pinpointing parameters with the most substantial impact on the software stack dependability. For this purpose, we developed a generic exploration tool that relies on supervised and unsupervised techniques to investigate correlations between fault injection results (i.e., Vanish, Hang, ONA, OMM, UT) and the application characteristics (e.g., cache statistics, number of branches) without the presence of faults (Figure 4d).

Machine learning algorithms can be categorized in three groups: (i) Supervised, (ii) Unsupervised, and (iii) Reinforced

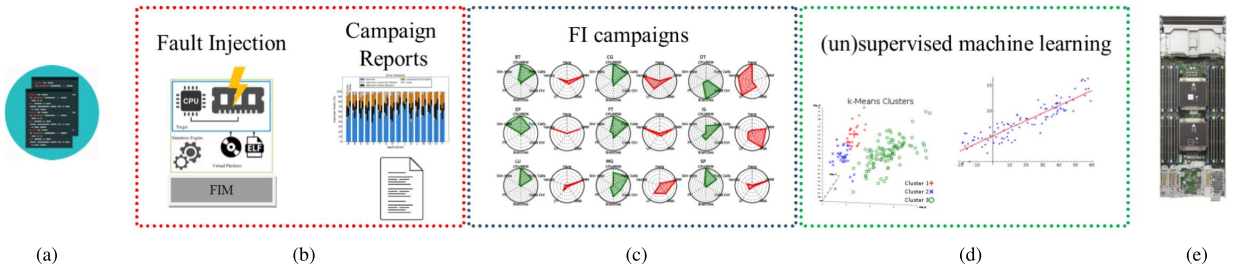


Fig. 4. Proposed soft error assessment flow using machine learning. (a) Application design. (b) Soft error assessment. (c) Application profiling. (d) Soft error correlation analysis. (e) Target system.

Learning. The supervised learning aims to predict the effect that one set of observations (i.e., input, features, attributes) has on another dependent variable (i.e., output, label, class). In this regard, supervised learning algorithms make predictions based on a set of training examples using regression and classification techniques. A regression technique models the target system using mathematical equations producing a continuous value output (e.g., linear regression) to approximate the target system behavior. In turn, classification algorithms are used to divide a data set into smaller subsets by evaluating its attributes (e.g., decision tree).

The unsupervised learning algorithms are used to search for patterns on unlabeled datasets (i.e., without a dependent variable output). While supervised learning correlates two groups of observations, the unsupervised learning describes the system features in more abstract levels of representations. For instance, the K-means is a well-known example of an unsupervised learning algorithm that enables to subdivide N observations into K clusters, where each observation belongs to the cluster with the nearest mean. Reinforced learning algorithms are employed to gain experience (i.e., knowledge) through a series of trial-and-error training sessions, where a cost function calculates reward or punishment values depending on its prediction. By minimizing the target cost function, the reinforced learning algorithm improves the system prediction accuracy until reaching a pre-defined quality threshold. Our article uses the first two types of techniques, as they provide the mathematical support to analyze the inputs. To proceed to reinforced learning algorithm requires a proactive framework, which is intended as future work by the authors.

As a proof-of-concept, we used the microarchitectural information provided by the gem5 simulator (e.g., memory usage, application instruction composition) as input values. Note that the proposed tool enables the adoption of any information sources, such as memory profiling tools (e.g., valgrind) or low-level architectural statistics (e.g., gate-level simulation). The analysis tool was developed using Python, taking advantage of available ML frameworks, in particular, the Scikit-learn module [23]. Since its release in 2007, Scikit-learn has become the most widely-used, general-purpose, open-source machine learning module. Further, this work adopts the pandas dataframe (i.e., two-dimensional size-mutable data structure) [24], which provides a data structure designed for large-scale explorations. Besides these two modules, several other libraries were employed to construct the exploration flow, such as matplotlib, numpy, scipy, multiprocessing.

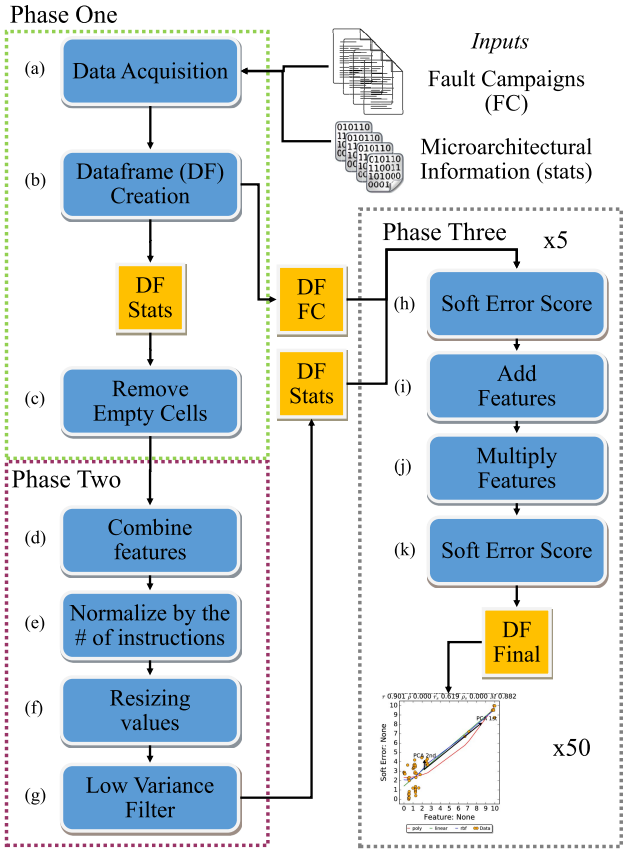


Fig. 5. Proposed soft error assessment tool automated flow.

A. Proposed Soft Error Assessment Tool Execution Flow

Figure 5 shows the proposed soft error assessment tool flow, which is organized in three main phases:

1) Phase 1 - Feature Acquisition and Data Homogenization: The tool collects the input files and extracts the necessary information (Figure 5a). This information is transformed into two distinct dataframes (Figure 5b) containing the fault injection campaigns (FC), and the microarchitectural information (i.e., gem5 Stats). Each dataframe line represents an individual fault campaign scenario (e.g., varying the number of cores, kernel, application) and columns represent the features. Following, the tool replaces all non-numerical (e.g., undefined, infinity, none) by zeros, as numerical methods do not handle it correctly.

2) Phase 2 - Unidimensional Feature Transformation and Selection: In this phase, the tool creates and filters new

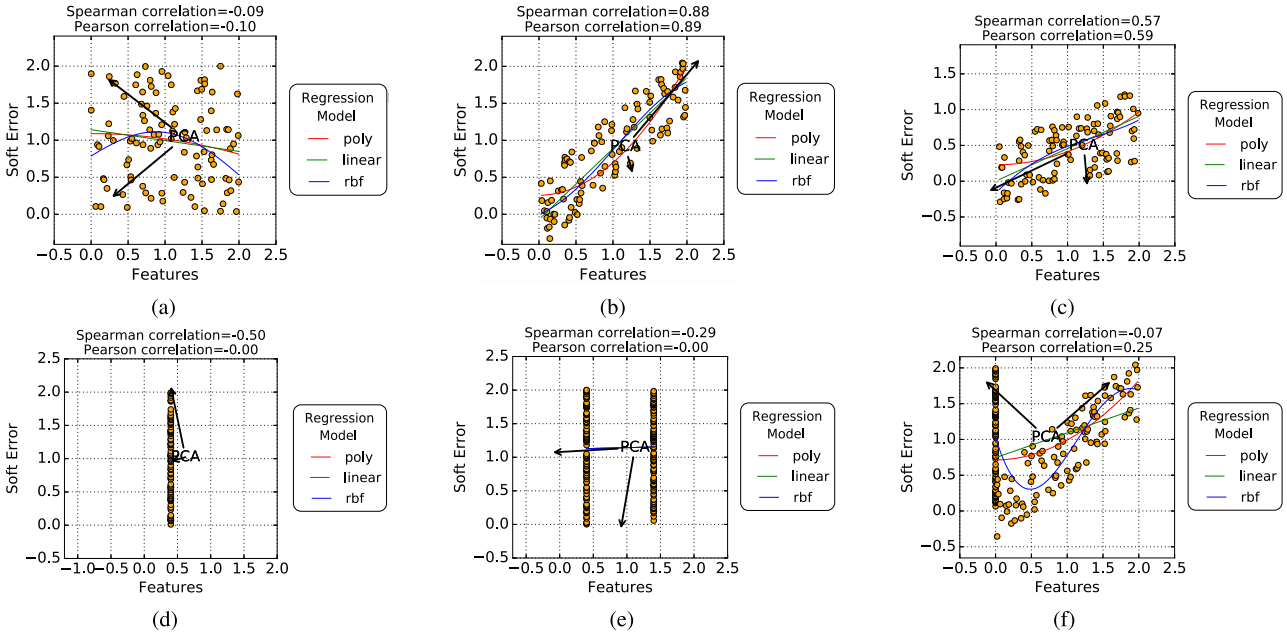


Fig. 6. Different of feature combination extracted from the gem5. (a) Random distribution. (b) Strong linear dependency. (c) Weak linear dependency. (d) Constant value. (e) Multiple constant values. (f) Usual feature shape.

features from the search space (i.e., all features) to improve the tool performance (i.e., smaller feature dimension) and accuracy (i.e., relevant information). First, the tool rearranges the collected data to create features with higher quality (i.e., knowledge content). For instance, the raw information provided by the gem5 divides the number of float instructions per individual core (e.g., system.cpu0.num_fp_insts, system.cpu1.num_fp_insts), and by merging them, it is possible to establish processor behavior as a whole (Figure 5d). Most of the raw data is described using nominal values (i.e., from zero to billions), depending on the target parameter and the application execution time. Due to this magnitude and discrepancy, a direct comparison between applications may not result in an evident relationship. To solve this problem, the tool normalizes all parameters (e) by the application length (i.e., number of instructions) in order to produce a relative value considering each application length. Next, the data is resized to range between 1 to 10 (f) to enable the comparison among distinct features. Thus, the tool reduces the total number of features (i.e., independently from the soft error) by eliminating the ones with low variance (g).

3) Phase 3 - Multidimensional Feature Transformation and Selection: Until this point, the exploration flow was restricted to a single dimension, where only microarchitectural features were considered. When considering the adopted soft error classification (i.e., Vanish, Hang, ONA, OMM, UT), the investigation becomes a five-dimensional problem. In this context, the tool performs five distinct investigations, one for each error class alongside the selected features. First, the proposed algorithm prunes the features from several thousand to 50 using the *Soft Error Score* (Figure 5h). Underlying score (see Section IV-B)) from 0 to 1 measures the feature impact (e.g., number of branches) on the target error class (e.g., Vanish, Hang). These outcomes are standardized in values between 0 and 1 to remove the magnitude parameter from the equation. Then, this reduced dataframe suffers two transformations: (i) addition, and (j) multiplication. In this regard, the tool adds and multiples the 50 columns in every possible combination leading to a total of 5050 features. Finally, the tool ranks the 50 most relevant features using the soft error score (k), revealing multi-dimension correlations between the microarchitectural parameters and fault injection campaigns.

B. Soft Error Score

Microarchitectural information (i.e., features) comes in multiple ways and they represent a wide range of parameters such as cache misses, number of float instructions, or virtual memory page size. Each one has a different distribution, for example, the virtual memory page size has few possible constant values while the number of float instructions ranges from zero to billions. Sometimes, parameters are constant for one application while fluctuates in another benchmark execution. Figure 6 shows six common feature arrangements found in the raw data from the gem5 microarchitectural statistics. For example, Figure 6a displays a random value distribution while Figures 6b and 6c exhibit two linear relationships with the first being the stronger one. In turn, Figures 6d and 6e display constant features (i.e., independent from the soft error). Finally, Figure 6f demonstrates the average behavior of raw features, mixing some linear behavior, outliers, and constant values.

Applying one feature filtering algorithm does not provide the target results (i.e., the microarchitectural features with higher impact on the system soft error reliability). For instance, the correlation coefficients perform poorly on noise dataset (Figure 6f) where the relationships exist among other types of data (e.g., constant values). The PCA can find the maximum growth direction in complex features but it can result in false-positives depending on the data distribution (Figure 6e).

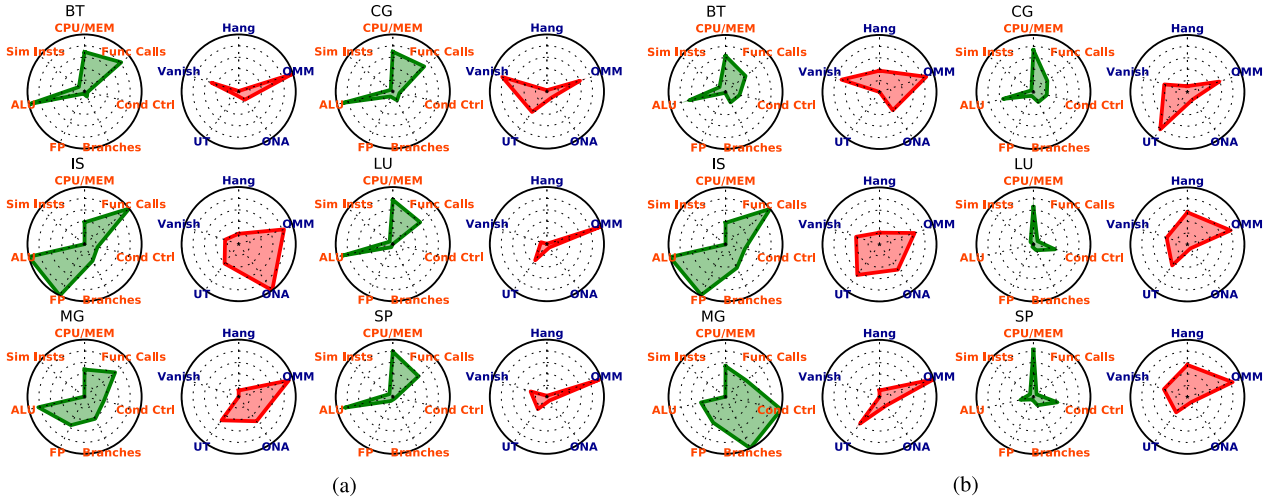


Fig. 7. The application profiling phase reveals common features that can be used to reduce the most frequent soft errors among similar applications. (a) MPI applications. (b) OpenMP applications.

The regression model provides a more robust solution without presenting a general solution. Further, applying multiple filters in sequence results on a small dataset with a significant amount of false-positive solutions.

To avoid false-positive results, this work proposes a filter score (i.e., from 0 to 1) to measure the feature quality according to the target soft error problem. This score results from simultaneous execution of several algorithms, i.e., the PCA, regression, and other techniques are computed in a single step. These outcomes are standardized in values between 0 and 1 to remove the parameter magnitude from the equation. The score computes the PCA first component, the Pearson's correlation coefficient, variance, dispersion score, and linear regression. Pearson's is preferable due to its smaller sensibility to false-positive linear relationships. Linear regression models with a 45° (from the origin) represent the optimal correlation between two dependent variables, increasing or decreasing this angle shows a weaker correlation.

The selection process computes the Soft Error Score for all columns in the target dataframe. The tool ranks the features by the score selecting the first N most significant values. This technique provides both filtering and ranking in a single technique, reducing the number of feature selection steps.

V. Results

This section demonstrates the effectiveness of the proposed soft error assessment flow based on ML techniques to detect the most relevant microarchitectural parameters to the system software stack dependability.

A. Experimental Setup

To properly evaluate the proposed soft error assessment flow, this paper considers two different Linux kernels (i.e., 4.3 or 3.13), and the NAS Parallel Benchmark suite [25] counting with 29 distinct applications (i.e., 10 Serial, 10 OpenMP, and 9 MPI benchmarks). Each application has distinct characteristics, some are CPU intensive while others are memory-bounded. Further, the execution time of a single simulation ranges from 300 million to 86 billion instructions. Experiments were conducted considering an architectural level simulator (i.e., gem5) to inject faults in general purpose registers [17] because it is highly used by academics as well as accepted in many industrial sectors. Configurations including single, dual, quad, and octa-core ARM Cortex-A9 (ARMv7 Architecture) and Cortex-A72 (ARMv8) processors model with 1GB of memory RAM, and two-level cache memory configured as: L1 instruction and data 32kB 4-way associative, and L2 512kB 8-way associative. Due to this high number of configurations, we were able to evaluate more than 150 scenarios (1.2 million failure injections, which require over than 1,050,000 simulation hours) considering not only the architecture (e.g., number of cores), but also software variations (e.g., kernel versions), among several others.

B. Application Profiling

This section presents a fault injection analysis similar to those commonly exploited in the literature. Researchers evaluate applications through their characteristics and present simple results, considering few benchmarks/applications [4]. We here aim at conducting solid experiments in order to create hypotheses and group applications that share common behaviour patterns that might lead to similar soft error results.

First, we exploited the access to the raw information to perform the profiling of both soft error results and the microarchitectural parameters (Figure 4c). The application profiling phase takes place after the gem5 microarchitectural information, collected from all fault injection campaigns, is retrieved and cleaned by the machine learning tool (Figure 5c).

Note that this phase exposes: (*i*) the application characteristics, such as CPU-bound or memory-bound trends, the number of branches, and others; and (*ii*) the fault injection campaigns results. Figure 7 compares the profiling of six NAS benchmarks implemented in MPI and OpenMP. The gathered information comprises the behavior of each benchmark implementation as it executes in the gem5 ARM processor model. This application profiling phase reveals trends that may be useful to group applications according to their behavior or removing outliers. The underlying profiling also reduces

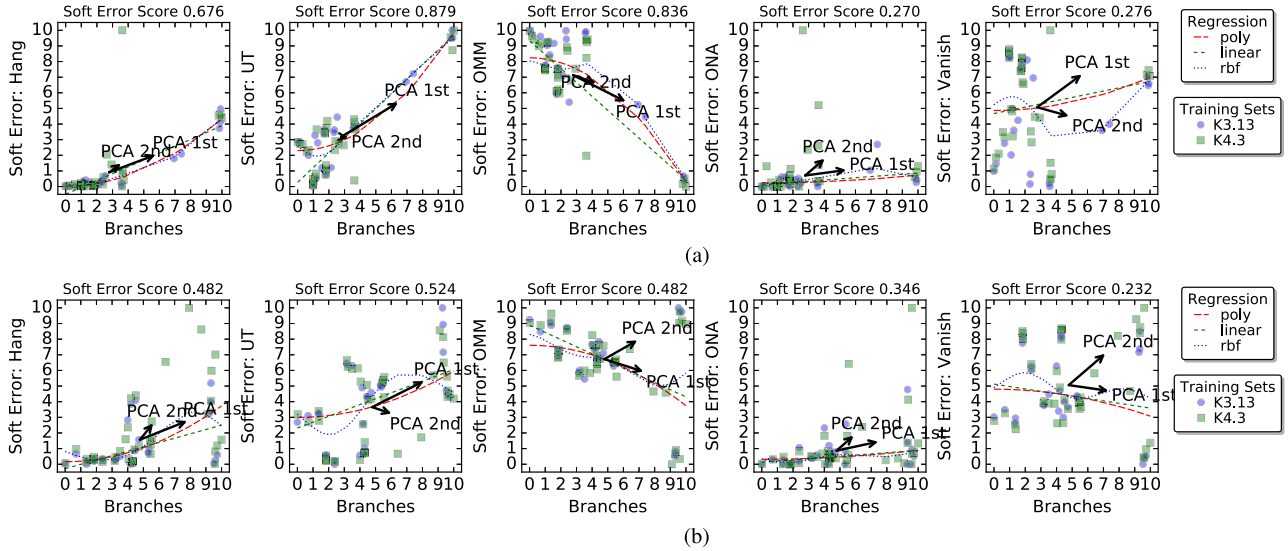


Fig. 8. Branch instructions impact on the soft error classification targeting parallel applications. (a) MPI. (b) OpenMP.

the number of future fault injection campaigns and enables users to conduct bespoke investigations.

From the conducted parallel application profiling, i.e., MPI (Figure 7a) and OpenMP (Figure 7b), it is possible to identify that the benchmarks implemented in OpenMP present a more massive branch concentration. OpenMP supports shared memory parallelism by executing code portions in parallel, usually loops, using a fork-and-wait approach, which increases the thread control complexity (i.e., more branches). On the other hand, MPI applications consist of one or more independent communication processes as denoted by the higher number of function calls (e.g., sending and receiving messages). From this initial exploration, it is possible to suggest two main patterns: (*hypothesis 1*) the increase of the feature branches, exposed by OpenMP leads to more occurrence of UT, Hang and Vanish; and (*hypothesis 2*) memory-bound applications (e.g., `IS`) present more ONA. Following, proposed ML techniques are used to reveal more complex soft error patterns considering both identified hypothesis.

C. Individual Feature Analysis

This section explores how a single feature (phase three, the output from Figure 5h) impacts on the soft error reliability of parallel applications. In this regard, the soft error assessment flow is used to evaluate the number of branches and memory transactions to provide solid results that may help to prove the hypotheses found in Section V-B.

1) Branches: Control flow statements (e.g., if, else, while, for) enable the creation of more complex algorithms, where jump, branch, and function call instructions fulfill this role at the assembly level. Any faults in such control flow instructions typically cause unrecoverable errors (e.g., segmentation faults). Figure 8 displays the branch instruction impact on the soft error reliability of the MPI (Figure 8a) and OpenMP (Figure 8b) implementations of the NAS benchmarks.

Note that the NAS benchmarks include from 300 million to 87 billion. Thus, due to the high number of instructions, only a small fraction of the total system simulation is devoted to execute the adopted Linux kernels (i.e., 4.3 or 3.13). Consequently, direct faults injections in the Linux inner workings are extremely rare (i.e., it accounts for less than 0.001%), resulting in faithful picture of the injected faults in the applications behaviour.

The hypothesis 1 states that the increase of branches exposed by OpenMP leads to more occurrence of UT, Hang and Vanishes. OpenMP relies on loop (i.e., for-while) parallelizations, which incurs in a greater branch presence than MPI. Figure 8b shows that OpenMP presents a low soft error score for branch instructions, which means that the hypothesis created from these results are not conclusive.

Unlike the *hypothesis 1*, the Figure 8a shows that although the number of branches in the MPI implementation of the NAS benchmarks is lower, they have a direct influence on the occurrence of Hang, UT, and OMM classes. Hang errors arise due to severe control flow errors (e.g., incorrect iteration counter), UT is an unexpected application termination (e.g., wrong address calculation), and an OMM results from an application finishing with an incorrect memory. Both Hang and UT show a direct and positive correlation (as correctly stated in *hypothesis 1*), while OMM a negative correlation (as erroneously stated in *hypothesis 1*). Therefore, the exact correlation is that the occurrence of Hang and UT increases at the same time that the OMM reduces.

2) Memory Transactions: Memory transactions (e.g., loads and stores) employ few registers (e.g., R0–3 and SP) to perform address calculations, consequently, the same registers are frequently reused. Although this feature apparently show an increase in ONAs, results contradict *hypothesis 2*. As the soft error score is not relevant, our proposed tool indicates that the occurrence of faults is due to the application itself. For example, the number of ONA in the IS benchmark is caused by the high utilization of the ULA for integer computation.

Figure 9 shows the soft error results related to memory instructions, where Vanish, OMM, and ONA are clustered into a single group to investigate the UT and Hang incidence.

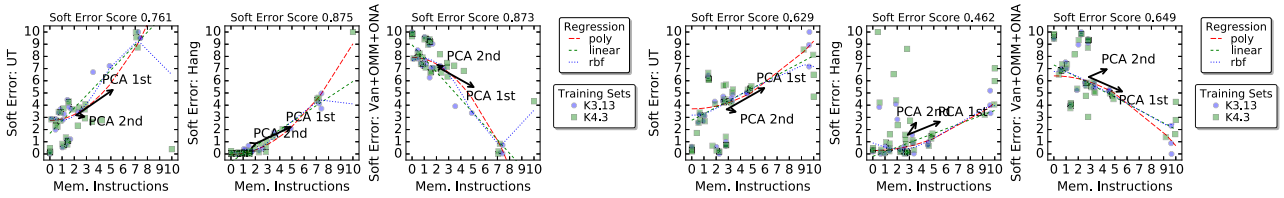


Fig. 9. Memory instructions impact on the soft error classification targeting parallel applications. (a) MPI. (b) OpenMP.

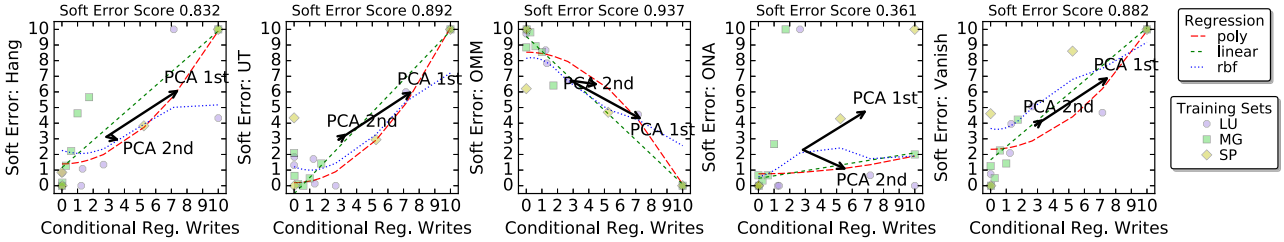


Fig. 10. The proposed ML tool reveals that the num_cc_register_writes feature can improve the system reliability of *Group A* applications.

Increasing the number of load/store operations lead to a more significant UT and Hang occurrence in MPI applications (Figure 9a), making them more likely to not complete their execution correctly due to the increased severity of the soft errors. In contrast, the OpenMP-based applications exhibit no direct impact of the number of load/stores in the UT/Hang incidence (Figure 9b).

D. Individual Feature Analysis Considering Application Profiling Similarities

This section expands the previous hypotheses by considering the influence of individual microarchitectural parameters on the soft error reliability of MPI and OpenMP applications, which share profiling similarities.

1) MPI: The applications `LU`, `SP`, and `MG` are combined into *Group A* due to their high OMM incidence and low number of Vanishes, making them appropriated candidates to improve system reliability. Figure 10 shows the *Group A* behavior considering the *num_cc_register_writes* parameter, which represents the number of writes on the conditional control register. Results corroborate the previous analysis (*hypothesis 1* Section V-B), which indicates a strong correlation between the number of branches and the growing occurrence of Hang, UT and Vanish. Figure 10 also shows that the ONA has no direct relationship with this parameter, while it increases the occurrence of Vanish, and reduces the frequency of OMM in the applications of *Group A*.

The proposed assessment flow based on ML techniques indicates the cache memory activity (i.e., the number of reads and writes hits), impact on the *Group A* reliability. A higher hit frequency on the cache memory results on a greater number of Hang and Vanish while it decreases the OMM incidence, as shown in Figure 11a. The relative soft error axis (i.e., resized) exhibits only the correlation strength, masking the feature impact concerning the real values. Figure 11b shows the impact of using an identical parameter with nominal values on both Hang and Vanish classifications. In this

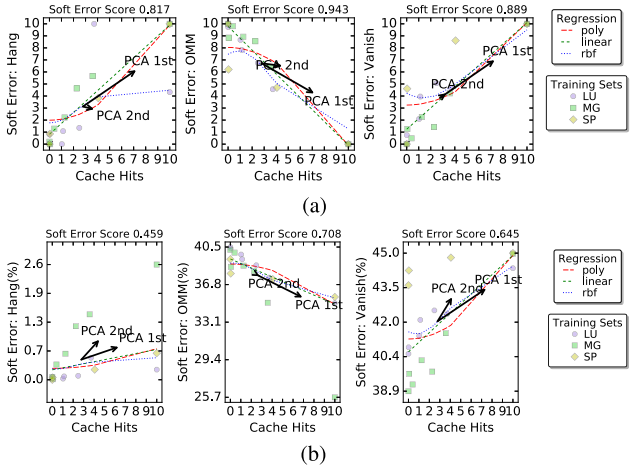


Fig. 11. The difference between (a) and (b) shows that some soft errors have more influence than others.

case, the same x-axis variation (feature) increases by 7% the occurrence of Vanish while Hang fluctuates around 1%.

Therefore, the greater Vanish rate caused by the larger cache activity benefits the system reliability, when considering the *Group A*. However, the number of data cache memory hits does not benefits all applications equally, e.g., the *Group B* composed of `BT` and `CG` applications. The *Group B* originally exhibits a high Vanish occurrence, and thus, this feature variation worsens the system reliability, as displayed in Figure 12.

2) OpenMP: The OpenMP and MPI parallelization libraries impacts the application profiling as discussed in Section V-B. However, applications with similar profiles might have similar features, which are suitable for optimization. In this sense, Figure 7 shows the OpenMP `MG` profile similarity with *Group A*, which relies on the MPI standard. Figure 13 shows an analogous behavior of the `MG` application (OpenMP standard) as the *Group A* (MPI standard), considering the data cache feature.

As a result, the increase of this feature results in a greater number of Vanish and, consequently, an improvement in the

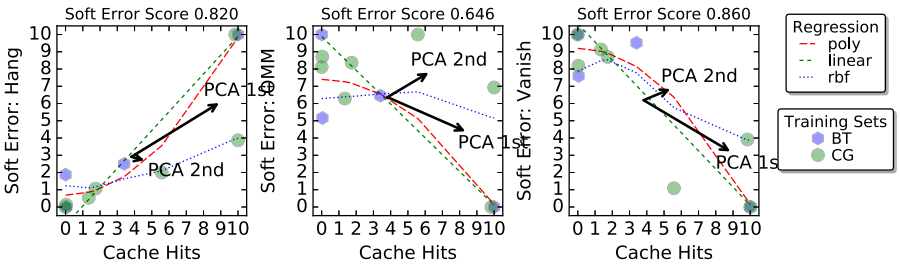


Fig. 12. The same feature that benefits *Group A* (Figure 11), harms *Group B* by increasing two soft errors (Hang and UT).

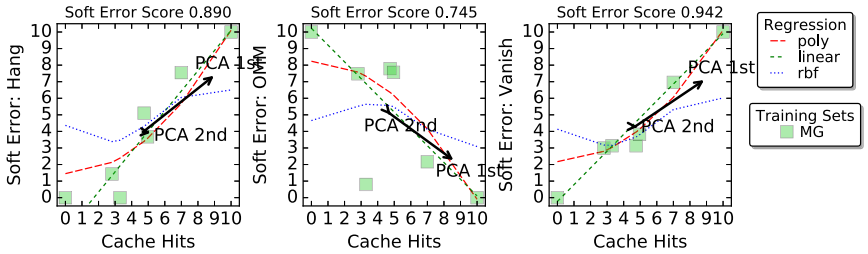


Fig. 13. OpenMP and MPI applications with similar behaviors may have improvements in system reliability using the same feature.

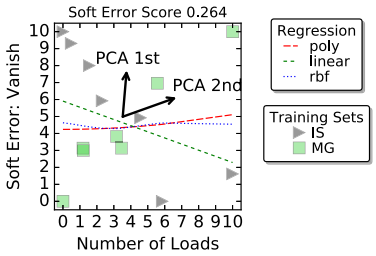


Fig. 14. Different applications can have distinct behaviors with the same feature.

system reliability. Our tool also indicates other influential features, e.g., the load instruction concentration in the application impacts on its comportment under fault injection. For instance, the `MG` and `IS` are affected in distinct manners as observable in Figure 14. While the `MG` Vanish rate increases with the number of loads, the `IS` has the opposite behavior. Note that each application responds differently to the modifications of this particular parameter, independently of the parallelization library. Therefore, although the application parallelization (MPI or OpenMP) affects the overall application behaviors, the resilience to soft errors rely on the initial application profiling. In this sense, our automated flow provides a flexible and robust mean to reduce human analysis time, as shown in Figure 3.

E. Combined Feature Analysis

After exploring the impact of one single feature on the software stack, this work investigates how the combination of two features affects the system dependability. Note that this is the first work that provides hints in such complex scenario, demonstrating the effectiveness of the proposed soft error assessment flow. It is worth mentioning that our machine learning tool supports the correlation of more than two features. However, it would be necessary to consider other data sources (i.e., microarchitectural parameters) than those exposed by gem5, such as Valgrid, VHDL, and others to produce more meaningful and complex correlations.

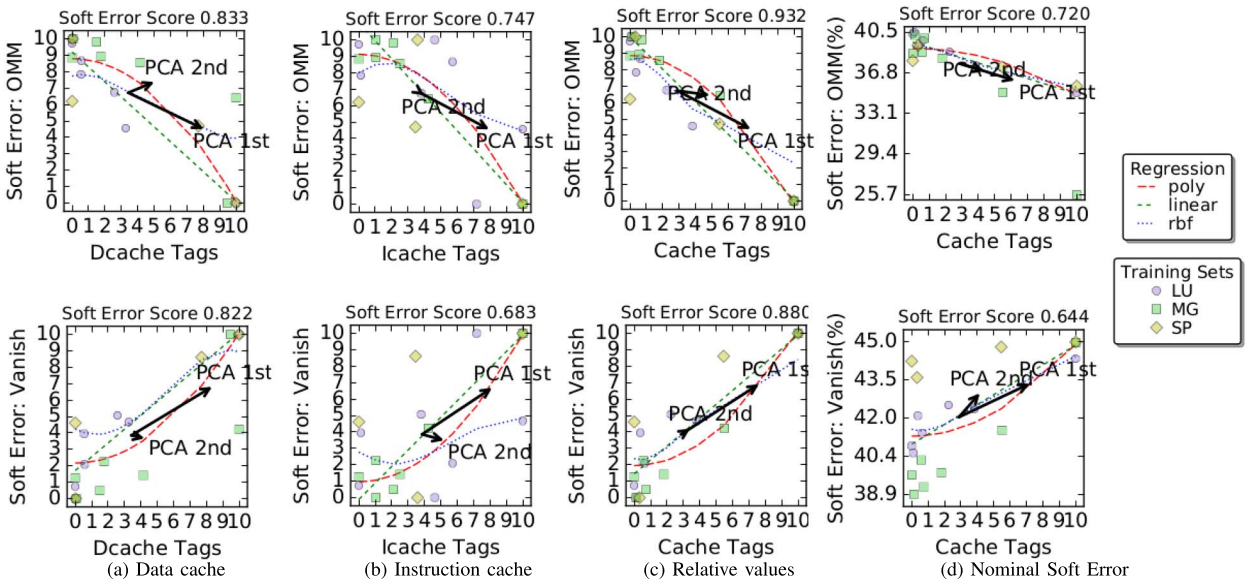
For combined analysis, the promoted tool automatically adds and multiplies the dataframe columns in all possible combinations (phase three, Figure 5). In this way, enabling the search for more complex correlations among the microarchitectural parameters and fault injection campaigns. For example, Figure 15 shows two distinct features the (a) data and (b) instruction cache valid references, while Figure 15c shows the combined features. *Group A* is CPU-bound and the few memory accesses have a direct impact on the soft error.

In this experiment, we observe that invalid references in cache memory cause CPU stall, and consequently increase the number of context switch, i.e., storing the state of a process or of a thread in the memory, which might be restored and executed later. If the bit-flip is injected into a register that will be used later, the memory is considered dirty. Even if the soft error is masked, the memory would remain dirty, causing an increase in OMM. In this regard, our tool reveals that the increase of valid references in both data and instruction cache memories causes a reduction in OMM and an increase in Vanish, i.e., the increase in valid references of each CPU's cache memory positively influences the reliability of *Group A*.

VI. Case Study

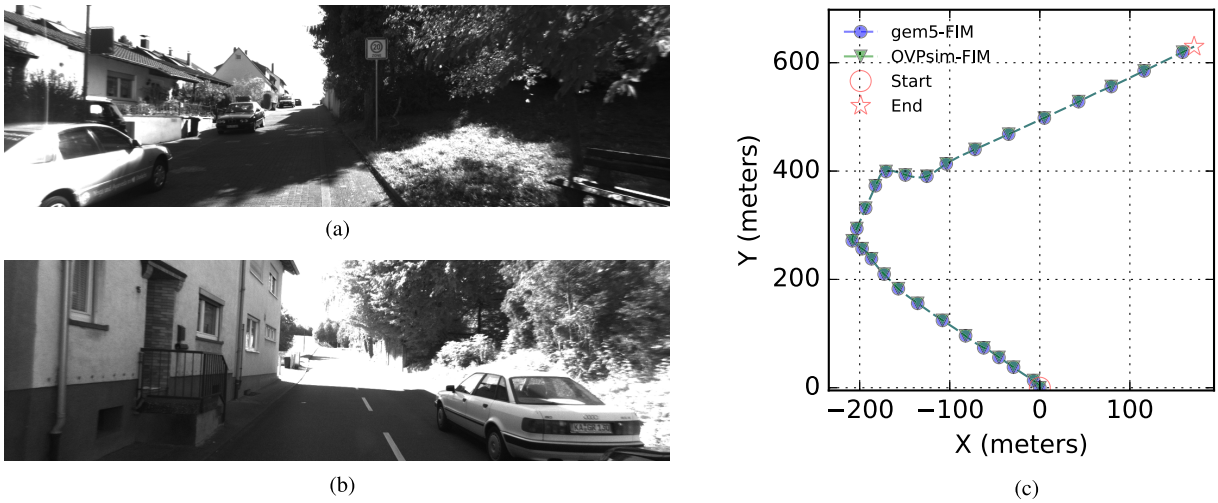
Automotive and technology market leaders are investing heavily to make self-driven cars commercially available by 2021. Such systems are integrating artificial intelligence, complex algorithms and software stacks aiming to analyze the real word, make decisions and perform actions without human input. Among the required algorithms there is the visual odometry (VO), which is used to determine the vehicle position and orientation by analyzing a series of images.

For this case study, we selected the LIBVISO2 [26] visual odometry library developed by the Karlsruhe Institute of



(a) Data cache (b) Instruction cache (c) Relative values (d) Nominal Soft Error

Fig. 15. Although individual feature analyses (a and b) may have unique benefits, combined feature analyses (c and d) reveal complex patterns that clarify system behavior, increasing the soft error score and, consequently, the system reliability.



(a) (b) (c)

Fig. 16. Selected visual odometry (VO) application. (a) Input Frame from KITTI Benchmark 11. (b) Input Frame from KITTI Benchmark 11. (c) VO application output path using distinct virtual platforms.

Technology (KIT). To use it, the library and its dependencies were compiled using the arm GCC cross-compiler with hard float-pointing and single instruction, multiple data (SIMD) flags enabled. Our LIBVISO2 setup uses the KITTI Vision Benchmark Suite [27], which is composed of 22 stereo odometry sequences captured from real-life vehicle trajectories.

This work uses the proposed soft error assessment flow, and its machine learning tool to evaluate the soft error reliability of a complex and real application, the KITTI Vision Benchmark Suite. Benchmarks integrating KITTI Vision have a high execution time. For instance, one single execution of KITTI benchmark 11 with 920 frames (up to 250 billion instructions) can take up to 36 hours with gem5-FIM, making unfeasible its use for the soft error assessment (Figure 4b). For that reason, we decide to use the microarchitectural information provide by the gem5-FIM, while performing the fault injection campaigns using the OVPsim-FIM, which requires less than one hour to execute the same scenario.

To illustrate the adopted case study, Figure 16 shows two input frames of the benchmark 11 input set from the KITTI suite alongside the prediction path algorithm using two virtual platforms. Figure 16c shows that the entire benchmark 11 traveled path produced by both OVPsim-FIM and gem5-FIM is identical, which shows the consistency between both simulators.

A. Visual Odometry Application Reliability

This section explores how compiler optimization flags impact on the execution of the visual odometry application. The initial exploration comprises four fault injection campaigns targeting the benchmark 11 input set from the KITTI suite, each one compiled with a different main optimization flag (i.e., O0-3). Further, we have restricted the stereo odometry sequence to 100 out of the 920 frames to reduce simulation time.

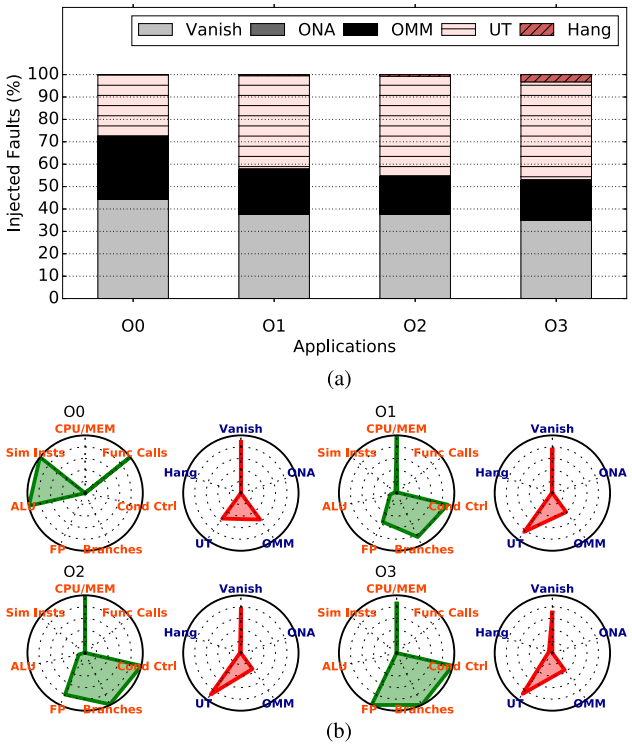


Fig. 17. Exploring distinct optimization flags using 100 frames of the benchmark 11 from the KITTI Vision Benchmark Suite. (a) Fault injection. (b) Application profiling.

Figure 17a shows the soft errors related to the fault injections in each optimization flag. Results show that the higher the optimization flag, the less reliable the system becomes. To complement these results, Figure 17b shows the application profiling considering each optimization flag. The application without any optimization flag (O0) has an execution 4.7x times larger than any optimization flag (i.e., O1-3), which means a reduction from 150 (i.e., O0) to 32 (i.e., optimized) billion instructions respectively. This extended execution time results in a more significant number of register operations (ALU). The compiler optimization reduces the number of instructions between control flow statements (e.g., if), which increases the probability of Hang and UT, as also highlighted in Figure 17a. Besides, when comparing applications with code optimizations (i.e., O1-3), it is noticeable the growing number of Hang occurrences due to the increasing modification in the loop-based snippets. For example, Figure 17b shows the growing number of branches with more aggressive GCC optimizations.

The visual odometry benchmark was chosen due to its real-life applicability in self-driven cars, which enables a physical representation of the system reliability (i.e., the traveled path). In this evaluation, the vehicle travels around 70 meters after processing 100 frames, revealing how compiler optimization flags can affect the traveled path.

The "Error (A)", shown in Figure 18, depicts the deviation between the correct and predicted stop point for each algorithm simulation under fault injection. The error reaches up to 70%, where larger values are due to algorithm halts (i.e., UT or Hang), making the car to stop a little bit far from the correct point. Figure 18a exhibits the results considering no GCC optimization (O0). In this scenario, for instance, 73.09% of the executions finished up to 1 meter away from the destination point. In contrast, when using compiler optimizations, the number of completed simulations closer to the correct point decreases from 73.09% to 54.82%. The "Error without UTs (B)", also shown in Figure 18, represents completed execution scenarios, where the algorithm evaluated all frames. The error considering only completed executions achieves up to 0.50m when compiled with optimizations while the O0 flag has no substantial deviation from the correct path, i.e., only four executions with a difference of fewer than eight centimeters.

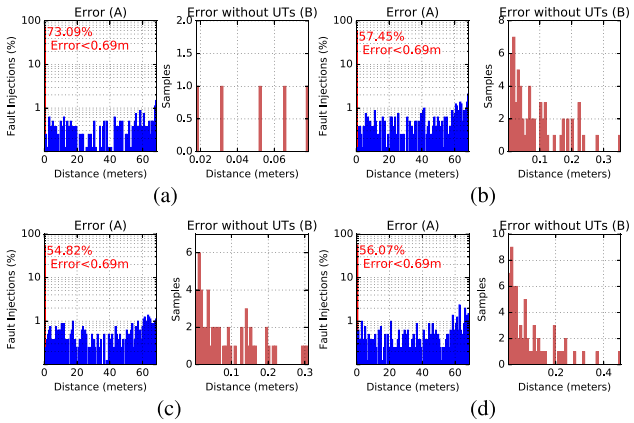


Fig. 18. VO application traveled route using 100 frames of the benchmark 11. (a) O0. (b) O1. (c) O2. (d) O3.

B. Exploring Individual Optimization Flags

As identified in Section VI-B, the main optimization flags (i.e., O0-3) affect the reliability of the visual odometry application. In this scenario, the objective of this section is to explore the individual GCC optimization flags that can potentially reduce the occurrence of soft errors. The GCC compiler has optimization flags enabled by default even when using the argument "-O0". The first fault campaign as depicted in Figure 19a shows the benchmark 11 compiled without any optimization flag under the influence of fault injections. Between the O0 flag and no optimization at all, the final compiled code has not meaningful changes leading to almost identical simulations (i.e., both application execute 150 billion instructions). Figure 18a and Figure 19a display this similarity considering both parameters (A and B).

Results shown in Figure 17 present a considerable reliability degradation when compiled with any other main optimization flags (i.e., O1-3). In turn, a notable speed reduction (i.e., 4.7x) is achieved when compiled with the O0 flag. As the most significant effect occurs when transitioning from O0 to O1, as presented in Figure 18, the next exploration investigates how to improve the O0 performance by adding individual optimization flags.

The stereo odometry sequence heavenly depends on loops to analyze the images, for this purpose, some unrolled loops options have been added to its compilation (Figure 19b). Results continue to show similar errors. Therefore, the next experiment (Figure 19c) mimics the O3 option by using

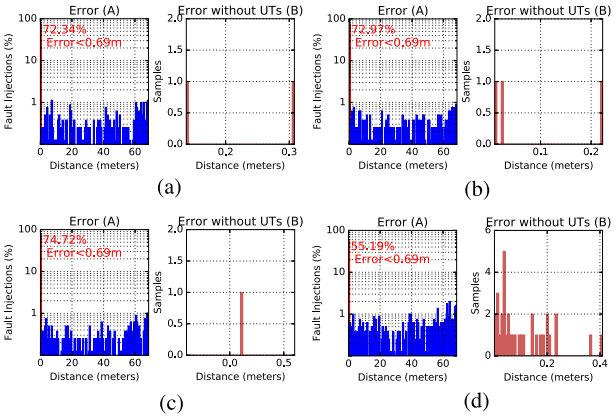


Fig. 19. Exploring the GCC optimization individual flags. (a) No optimization flags. (b) O0 plus unrolling loop. (c) Synthetic O3. (d) O3 plus unrolling loop.

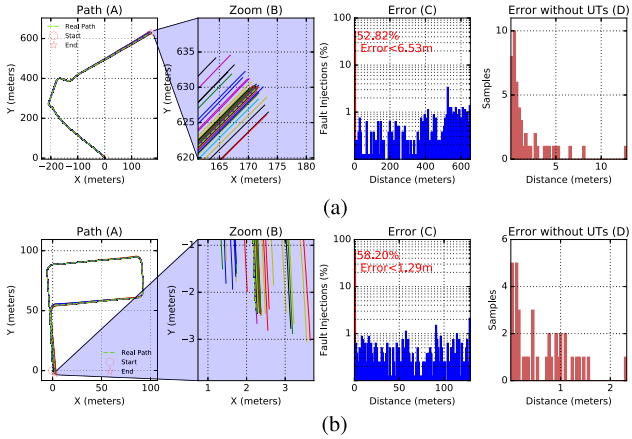


Fig. 20. VO application complete execution considering two distinct inputs. (a) Benchmark 11 (920 frames). (b) Benchmark 14 (630 frames).

individual optimization flags instead (e.g., -falign-jumps -falign-loops -falign-labels -fcaller-saves -fcrossjumping). This investigation aims to replicate the O3 effect on the code without directly adding the "-O3" flag. Individual optimization flags have minimal impact on the final system reliability and performance as they have a minor effect on the compiler behavior, as seen in the difference between Figure 18d and Figure 19c. Therefore, primary optimization flags (e.g., O0-3) are not just group alias, they have a direct impact on the compiler algorithm. That means that GCC does not provide full control of these optimizations, leaving the most relevant options hardcoded in the compiler source code. For example, the benchmark 11 without any optimization simulates 150 billion instructions while the O3 recreation requires 120 billion, still four times longer than the default O3 flag.

Most of the existing software projects use O2 as standard shipping optimization flag because a four times slowdown is not acceptable. In the impossibility of improving the software compiled with O0, the last experiment attempts to reduce the number of unexpected terminations when using the O3 optimization flag. This fault campaign (Figure 19d) uses aggressive loop and branch optimizations, given more freedom to the compiler to allocate the instructions orders. Results demonstrate the lack of support provided by GCC to algorithm customization's, as most of the code transformations are locked to hardcoded arguments.

C. Complete Path Execution

After analyzing the impact of individual GCC flags on the visual odometry application behavior, we extended this exploration to the effects of the traveled path on the accumulated error, aiming to reveal the real impact that the application suffers under the presence of injected faults. This exploration uses two input sets from the KITTI suite, the benchmark 11 featuring 920 frames and the benchmark 14 with 630 frames, and both compiled with the O3 optimization flag.

Figure 20 shows the traveled path for each fault injection scenario (A) with a zoom on the final stopping point (B). The benchmark 11 follows a more straight path leading to horizontal errors, i.e., the vehicle diverges either to the right or to the left (Figure 20a). In contrast, the benchmark 14 after each curve the car leans towards the correct way (dashed green line) as right and left accumulated errors attenuate each other. However, the vertical error accumulates in a greater magnitude, as shown in Figure 20b. Figure 20 also displays the error from the correct stopping point considering every fault injection (C) and only the ones with complete (i.e., neither Hang or UT) application (D). Note how the number of completed simulations (in red) remains at the same levels of the reduced simulations (i.e., 100 frames), around 55% when compiled with an identical GCC flag (i.e., O3).

VII. Conclusions

This paper described a novel soft error assessment flow that enables software engineers to not only identify the occurrence of soft errors in complex software stacks but also determine the correlation between multicore platform microarchitectural characteristics and detected soft errors using supervised and unsupervised machine learning techniques. Proposed flow and tools provide a step change in understanding multicore software stacks behavior under the presence of faults at early design phase.

The effectiveness of our soft error assessment flow was evaluated through an extensive data set gathered from more than 1.2 million fault injections, considering realistic and sophisticated software stacks including Linux kernel and benchmarks with up to 87 billion instructions. Aiming to demonstrate the applicability of our proposal in a realistic environment, we investigate the soft error reliability of a visual odometry algorithm commonly used in self-driven cars. Results show that the occurrence of soft errors affects the vehicle travel.

References

[1] M. Snir *et al.*, "Addressing failures in exascale computing," *Int. J. High Perform. Comput. Appl.*, vol. 28, no. 2, pp. 129–173, May 2014.
[2] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," in *Proc. ASPLOS*, 2012, pp. 1–12.
[3] S. K. S. Hari, R. Venkatagiri, S. V. Adve, and H. Naeimi, "GangES: Gang error simulation for hardware resiliency evaluation," in *Proc. ISCA*, 2014, pp. 61–72.

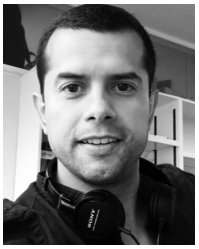
[4] F. de Aguiar Geissler, F. L. Kastensmidt, and J. E. P. Souza, "Soft error injection methodology based on QEMU software platform," in *Proc. LATW*, 2014, pp. 1–5.
[5] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos, "Differential fault injection on microarchitectural simulators," in *Proc. IISWC*, 2015, pp. 172–182.
[6] K. Tanikella, Y. Koy, R. Jeyapaul, K. Lee, and A. Shrivastava, "GemV: A validated toolset for the early exploration of system reliability," in *Proc. ASAP*, 2016, pp. 159–163.
[7] Q. Guan *et al.*, "Design, use and evaluation of P-FSEFI: A parallel soft error fault injection framework for emulating soft errors in parallel applications," in *Proc. SIMUTOOLS*, 2016, pp. 9–17.
[8] M. Didehban and A. Shrivastava, "nZDC: A compiler technique for near zero silent data corruption," in *Proc. DAC*, 2016, pp. 1–6.
[9] H. Khosrowjerdi, K. Meinke, and A. Rasmusson, "Virtualized-fault injection testing: A machine learning approach," in *Proc. ICST*, 2018, pp. 297–308.
[10] P. S. Magnusson *et al.*, "Simics: A full system simulation platform," *IEEE Comput.*, vol. 35, no. 2, pp. 50–58, Feb. 2002.
[11] M. M. K. Martin *et al.*, "Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset," *ACM SIGARCH Comput. Archit. News Lett.*, vol. 33, no. 4, pp. 92–99, 2005.
[12] A. Vishnu, H. van Dam, N. R. Tallent, D. J. Kerbyson, and Adolfy Hoisie, "Fault modeling of extreme scale applications using machine learning," in *Proc. IPDPS*, 2016, pp. 222–231.
[13] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, C.-Y. Cher, and P. Bose, "Understanding the propagation of transient errors in HPC applications," in *Proc. SC*, 2015, pp. 1–12.
[14] R. A. Ashraf, R. Gioiosa, G. Kestor, R. F. DeMara, "Exploring the effect of compiler optimizations on the reliability of HPC applications," in *Proc. IPDPSW*, 2017, pp. 1274–1283.
[15] N. Narayanamurthy, K. Pattabiraman, and M. Ripeanu, "Finding resilience-friendly compiler optimizations using meta-heuristic search techniques," in *Proc. EDCC*, 2016, pp. 1–12.
[16] F. Rosa, F. Kastensmidt, R. Reis, and L. Ost, "A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability," in *Proc. DFTS*, 2015, pp. 211–214.
[17] F. da Rosa, V. Bandeira, R. Reis, and L. Ost, "Extensive evaluation of programming models and ISAs impact on multicore soft error reliability," in *Proc. DAC*, 2018, Art. no. 178.
[18] Imperas. (2017). *Open Virtual Platforms (OVP)*. [Online]. Available: http://www.ovpworld.org/
[19] N. Binkert *et al.*, "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, May 2011.
[20] N. Seifert *et al.*, "Soft error susceptibilities of 22 nm tri-gate devices," *IEEE Trans. Nucl. Sci.*, vol. 59, no. 6, pp. 2666–2673, Dec. 2012.
[21] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *Proc. DAC*, 2013, Art. no. 101.
[22] G. Staples, "TORQUE resource manager," in *Proc. SC*, 2006, Art. no. 8.
[23] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.
[24] W. Mckinney, "pandas: A foundational python library for data analysis and statistics," White Paper, 2011. [Online]. Available: https://www.dlr.de/sc/Portaldata/15/Resources/dokumente/pyhpc2011/submissions/pyhpc2011_submission_9.pdf
[25] D. H. Bailey *et al.*, "The NAS parallel benchmarks summary and preliminary results," in *Proc. Supercomputing*, 1991, pp. 158–165.
[26] A. Geiger, J. Ziegler, and C. Stiller, "StereoScan: Dense 3D reconstruction in real-time," in *Proc. IV*, 2011, pp. 963–968.
[27] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? The KITTI vision benchmark suite," in *Proc. CVPR*, 2012, pp. 3354–3361.

Felipe Rocha da Rosa received the bachelor's degree in computer engineering and the Ph.D. degree in microelectronics from the Federal University of Rio Grande do Sul.

For the past six years, he has been researching and developing tools for performance and reliability analysis of arm-based processors.

His current research involves applying data science and machine learning techniques to improve the soft error investigations during early design space explorations process, simulation, system design, computer architecture, multi/many-core systems, network-on-chip, fault tolerance, and soft errors.

Rafael Garibotti (M'14) received the B.Sc. degree in computer engineering from the Pontifical Catholic University of Rio Grande do Sul (PUCRS), Brazil, in 2007, the M.Sc. degree from EMSE, France, in 2009, and the Ph.D. degree in microelectronics from the University of Montpellier in 2014. He was a Post-Doctoral Fellow with the Federal University of Rio Grande do Sul and Harvard University. He is currently an Associate Professor with the School of Technology, PUCRS. His research activity focuses on the programmability of multi/many-core systems and the design of parallel architectures, such as accelerators.



Luciano Ost received the Ph.D. degree in computer from the Pontifical the Catholic University of Rio Grande do Sul, Brazil, in 2010. During his Ph.D., he was an Invited Researcher with the Microelectronic Systems Institute, Technische Universitaet Darmstadt, from 2007 to 2008, and with the University of York in 2009. He was a Research Assistant and then an Assistant Professor with the University of Montpellier II/LIRMM, France, until joining the University of Leicester as a Lecturer in 2014. He is currently a Faculty Member with Loughborough University, U.K. He has authored more than 70 scientific papers, published in peer-reviewed international journals and conferences. His primary research directions comprise the design and exploration of reliable and performance-efficient multi/many-core systems.

Ricardo Reis (M'81–SM'06) received the Electrical Engineering degree from the Federal University of Rio Grande do Sul (UFRGS), Porto Alegre, Brazil, in 1978, and the Ph.D. degree in informatics and option microelectronics from the Institut National Polytechnique de Grenoble, Grenoble, France, in 1983. He has been a Full Professor with UFRGS since 1981. He is at research level 1A with the Brazilian National Science Foundation (CNPq), Brasília, Brazil, and the Head of several research projects supported by government agencies and industry. He has published over 550 papers in journals and conference proceedings and has authored or co-authored several books. His current research interests include physical design, physical design automation, design methodologies, digital design, EDA, circuits tolerant to radiation, and microelectronics education. He is member of the IEEE CASS BoG (2018–2020). He was a recipient of the IEEE Circuits and Systems Society (CASS) Meritorious Service Award in 2015. He received the Doctor Honoris Causa from the University of Montpellier, France, in 2016. He is the Chair of the IFIP TC10. He was the Vice President of the IEEE CASS representing Region 9 (Latin America) and the Brazilian Computer Society (SBC). He has also organized several international conferences. He was a member of the CASS Distinguished Lecturer Program 2014–2015.