AVGI: Microarchitecture-Driven, Fast and Accurate Vulnerability Assessment

George Papadimitriou Dimitris Gizopoulos
Department of Informatics and Telecommunications
National and Kapodistrian University of Athens
{georgepap | dgizop}@di.uoa.gr

***Abstract*—We propose AVGI, a new Statistical Fault Injection (SFI)-based methodology, which delivers orders of magnitude faster assessment of the Architectural Vulnerability Factor (AVF) of a microprocessor chip, while retaining the high accuracy of SFI. The proposed methodology is based on three key insights about the way that faults traverse complex out-of-order microarchitectures: (1) the distribution of the different ways that hardware faults manifest at the software (i.e., the first effects of faults to the software layer) is relatively uniform across workloads, (2) the *final* effects of faults in a specific hardware structure (i.e., their effect on the program execution) is relatively uniform for different workloads and depends on the distribution of the above fault manifestations, and (3) the majority of first manifestations occur in certain timeframe from the fault occurrence, which is significantly s*horter* than the complete execution of the workload, and depends on the type of hardware structure. Based on these insights, the proposed AVGI methodology accurately estimates the complete cross-layer vulnerability (i.e., AVF) for every hardware structure in fine granularity (SDCs and Crashes). Our experimental analysis shows that preserving high levels of accuracy, the proposed AVF assessment methodology is up to 337x and 440x faster than an accelerated exhaustive SFI, for two different microarchitectures of 64-bit Armv8 and 32-bit Armv7 CPU models, respectively.**

***Index Terms*—microarchitecture; reliability; hardware/software interface; microprocessors; microarchitecture-level fault injection**

I. Introduction

Reliability evaluation in early stages of microprocessor designs varies in the level of hardware modelling accuracy, the speed of the evaluation, and the granularity of the assessment report. Determining the Architectural Vulnerability Factor (AVF) of each individual hardware structure of the microprocessor throughout full end-to-end program execution is the most comprehensive way to measure the vulnerability of the entire system stack, including the microarchitecture, architecture, and software layers. The AVF of a hardware structure is the probability that a transient fault in it will affect the program's output [1] [2]. Typically, designers rely either on Statistical Fault Injection (SFI) [1] or on analytical methods, such as the Architecturally Correct Execution (ACE) analysis [2] [3], to provide insights into the programs' resiliency toward transient faults, because both methods aim to report the AVF of hardware structures. However, both techniques come with pros and cons: SFI is easier to implement, provides very accurate AVF but is significantly slow since it required several runs to reach high confidence; on the other hand, ACE analysis is fast (few runs), but requires very high development effort and provides pessimistic AVF overestimations. For example, Fig. 1 shows the AVF for SFI and ACE analysis on our infrastructure for the physical register file of an Arm Cortex-A72-like CPU. The AVFs reported by ACE analysis are constantly larger than SFI by 1.2x to 3x.

At the cost of multiple simulation runs of SFI, AVF measurements provide useful and accurate insights on the susceptibility of programs across the entire system stack (from the microarchitecture to the software layer). In an effort to accelerate SFI, several methods work at higher (thus faster) levels of abstraction (architecture or software), and evaluate the program's vulnerability assuming that the origin of a flipped hardware bit is an architecturally visible location (e.g., the Program Vulnerability Factor – PVF) [4]-[13]. Vulnerability evaluation methods that rely on the software or architecture level of abstraction are clearly much faster than the end-to-end AVF estimations that account for all hardware bits and operate at the microarchitecture cycle-accurate detail. Due to their speed benefit, high-level vulnerability evaluation methodologies have become common practice. However, a major pitfall in microprocessor reliability assessment has been recently demonstrated [14]: high-level fault injection approaches (i.e., microarchitecture agnostic) *mislead* the reliability assessment of microprocessors and may instruct designers to take *wrong* decisions for error protection.

In this paper we propose AVGI[1], a novel microarchitecture-level SFI-based methodology, which accelerates the AVF assessment for every hardware structure of a microprocessor and the entire chip (up to 337x faster per structure compared to an accelerated typical SFI flow, and 22x for an

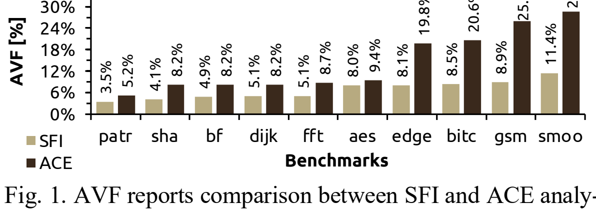


Fig. 1. AVF reports comparison between SFI and ACE analysis for the physical register file of Arm Cortex-A72-like CPU.

[1] AVGI is a Greek word for dawn (in Greek: *Αυγή*): the first appearance of light in the morning followed by sunrise; or figuratively: something new or the beginning of a great development.

entire 64-bit CPU), while retaining the high accuracy of SFI. The proposed methodology accurately delivers fine-grain cross-layer AVF assessment for every fault effect, including the most critical class of Silent Data Corruptions. The contributions of this paper are the following:

1. We categorize the hardware faults manifestations to the architecture (ISA – Instruction Set Architecture) layer, providing eight *complete and mutually exclusive ISA Manifestation Models* (IMMs). The unique set of IMMs is a fundamental knob that can refine the accuracy of high-level assessment methods.
2. We experimentally explore, *for the first time in the literature*, the relation of the IMMs to the final effect on the program output (i.e., what is the eventual "fate" of a fault after it is classified in one of the IMM classes), for 12 major CPU hardware structures. We conduct a microarchitecture-dependent analysis and a cross-layer AVF analysis, which reveals two key insights of faults behavior: (i) the distribution of IMMs (i.e., first appearance of faults to the software) is relatively uniform across programs, and (ii) the final fault effects classification for each hardware structure (i.e., the effects of faults on the program output) depends on the IMM distribution.
3. We experimentally observe that most fault manifestations (i.e., IMMs) occur within a specific short timeframe from the fault occurrence, due to the effective residency time of faults in a specific resource. This timeframe differs among hardware structures but is significantly shorter than the end-to-end execution of the workload, which suggests further speedup.
4. We propose a novel microarchitecture-driven methodology for full stack vulnerability assessment that combines and leverages the previous insights with fast runtime profiling to eventually elicit the final fault effects without running time-consuming AVF evaluations. The proposed methodology provides all AVF classes, by considering any microarchitecture-level fault occurrence (i.e., there is no pruning of the initial fault list), including the most important one of Silent Data Corruptions (SDCs) and delivers extremely accurate vulnerability results in up to 337x shorter time than end-to-end AVF.
5. We conduct a case-study that considers a different ISA and microarchitecture to demonstrate the effectiveness of the proposed methodology on a different microarchitecture. In this context, we present AVF and FIT (Failures in Time) rates, which clearly demonstrate the accuracy of vulnerability evaluations for every hardware structure of the microprocessor and the entire CPU for both ISAs and microarchitectures.

II. Experimental Setup

A. Software Manifestation Models

Reliability evaluation through Statistical Fault Injection either at the hardware or at the software layers, is based on injecting faults (bit flips to model transient faults) and simulating to the end of execution to observe their effect on program execution. It is, therefore, important to model the way that faults propagate from the hardware layer and manifest to the software layer. For this purpose, we introduce a complete set of mutually exclusive ISA Manifestation Models (IMMs), listed in Table I, which define the interface between the hardware and the software as faults traverse the abstraction stack. We adopt the coarse-grained fault propagation models from [14], but refine them into more distinct groups, considering *any possible effect of a hardware fault at the software layer*. As we show in next sections, the set of IMMs is complete and mutually exclusive. The proposed methodology leverages this classification, and as we discuss in the following sections, it can effectively elicit the final fault effects (i.e., the effects of faults on the program output) through the unique properties of the IMM classes.

B. Final Fault Effect Classification

Any SFI campaign, assumes that the occurrence of a fault may influence the eventual output of the program. We classify the effect on the program output into the following fault effect classes (typically used in all fault injection studies at any layer of abstraction):

Masked: Simulation finished with no deviations from a fault-free execution, Thus, the fault did not affect the system or the application in any observable way.

Silent Data Corruption (SDC): Simulation finished normally, but the program output was different than the fault-free simulation, without any observable indication.

TABLE I
THE EIGHT MUTUALLY EXCLUSIVE ISA MANIFESTATION MODELS (IMMS) AND THEIR DESCRIPTION.

	IMM	Description
IFC	Instruction Flow Change	A different instruction is executed compared to the original program flow due to an incorrect instruction fetching
IRP	Instruction Replacement	A different instruction is executed compared to the original program flow due to a corrupted Opcode
UNO	Unknown Operand	One or more instruction operand fields are corrupted and are unknown to the ISA
OFS	Operand Forced Switch	Register operand(s) and/or immediate value(s) field(s) of the instruction format are corrupted
DCR	Data Corruption	The correct resource is used, but the content of the resource (register or memory word) is corrupted
ETE	Execution Time Error	The instruction is correct, but it was committed in a wrong clock cycle compared to the fault-free execution
PRE	Pre-Software Crash	The execution crashes before the fault affects the ISA due to a high-level condition which is ISA-undefined
ESC	Escaped	Faults that corrupt the program output without ever reaching the software layer (see details in section IV.D)

Crash: A simulation that neither reached the end of the program nor finished within a certain amount of time, because it was disturbed by a catastrophic event. As a result, no program output was produced.

C. Statistical Fault Injection Framework

Fault injection should ideally be based on a real system or a very detailed low-level simulator (e.g., RTL). Although low-level simulators may provide accurate fault effects, their simulation throughput is extremely low to be affordable and cannot model long running workloads with OS activity. Our methodology relies on microarchitecture-level simulation using the *gem5* simulator [15], which allows deterministic end-to-end execution of large workloads on top of an operating system that is impossible at lower levels. Even if an RTL model of a microprocessor was available and the full system injection on it was possible, it would only marginally augment our analysis [16] with the vulnerability of the combinational logic, which has very low raw failure rates compared to storage elements on which we focus [17].

For this reason, we present our methodology harnessing the high throughput of microarchitectural modeling in performance simulators, such as *gem5*. We employ GeFIN [18], the state-of-the-art microarchitecture-level fault injection framework built on top of the *gem5* [15]. GeFIN consists of a modified *gem5* version that allows fault injection along with instrumentation for running and controlling simulation campaigns on full-system setup [14] [18]-[20].

D. Hardware Structures & Benchmarks

In this study, we first employ a 64-bit Armv8 ISA, modeling an out-of-order microarchitecture, which is very similar to the Arm Cortex-A72 CPU. For a comprehensive analysis, our evaluations target 12 important hardware structures: L1 data and instruction caches (tags and data fields), L2 cache (tags and data fields), the Physical Register File, the Load Queue (LQ), the Store Queue (SQ), the Reorder Buffer (ROB), and the Instruction and Data TLBs. We employ a diverse set of 13 workloads consisting of: (a) 3 workloads from NAS benchmarks suite [21], and (b) 10 workloads from the MiBench benchmarks suite [22] using *the largest possible input datasets for all benchmarks*. The execution times of the benchmarks range from 100 million cycles to 2.2 billion cycles and our experiments include both integer and FP benchmarks as well as both compute and memory bound benchmarks.

In contrast to previous microarchitecture-level reliability studies that employ SPEC benchmarks either considering only Simpoints of 10 to 100 million cycles (i.e., a very short part of the benchmark) or interrupting the simulations during a few thousand of cycles after the fault injection [23]-[27], in this study, *we consider the end-to-end execution of benchmarks with significantly higher number of cycles (up to 2.2 billion cycles)*. MiBench suite is commonly used in reliability studies [18]-[20], [28]-[36] as it facilitates complete end-to-end executions. For each of the 12 components, 2,000 single-bit faults were randomly generated following the uniform distribution as defined in [1], resulting in 312,000 faults for all 13 benchmarks and the 12 different hardware components. We follow the widely adopted formulation of [1] for the statistical fault sampling calculations; our 2,000 fault samples correspond to 2.88% error margin with 99% confidence level.

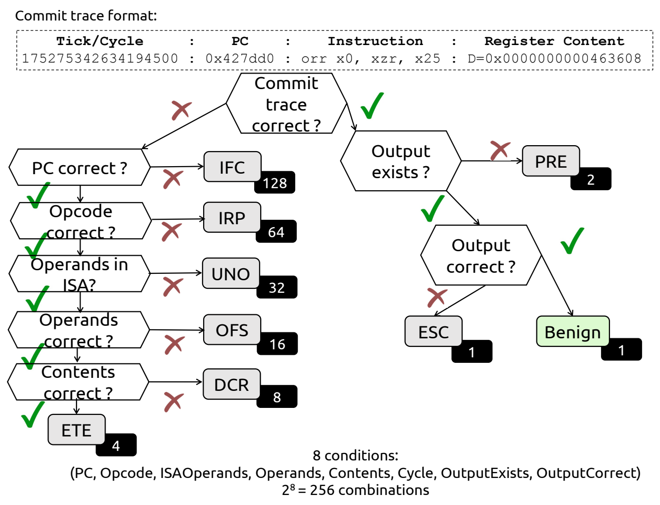
III. Understanding Faults Behavior

In this section we explore the "fate" of IMMs towards the final fault effect in the output of the program for all hardware structures. We conduct a complete fast HVF (Hardware Vulnerability Factor) analysis [37] (to extract the IMMs classes) and a cross-layer AVF analysis [1] (for the final fault effect classes) for all benchmarks and hardware structures of a microprocessor. Combining the insights of both analyses, we show how they can be used and influence the microarchitecture-driven assessment presented in Section IV.

A. Experimental Analysis

Our analysis employs two experimental steps for the vulnerability evaluation of different layers: the HVF assessment and the AVF assessment, providing the microarchitecture-dependent vulnerability and the cross-layer vulnerability, respectively. The microarchitecture-dependent evaluation (i.e., HVF) focuses on the effects of hardware faults only until they first "touch" the software layer and stops at that point. For the HVF analysis, we consider as *Benign* faults, those faults that eventually get masked by a microarchitectural operation (e.g., a misprediction), and thus, the fault occurrence never reach the commit stage. Since the fault occurrence did not commit, the fault is not architecturally visible, and it is categorized as *Benign*. On the other hand, any fault that reaches the commit stage (i.e., architecturally visible), is considered as a *Corruption*. Each Corruption is categorized to one (and only one) IMM, from those listed in Table I, depending on the type of corruption.

The diagram of Fig. 2 explains in detail the process we follow to classify a hardware fault into one of the IMM classes; any fault ends up *to one and only one* class. Each instruction is associated to the following parameters upon its retirement: (i) the committed cycle, (ii) the Program Counter (PC), (iii) the opcode, (iv) the register operands and/or an immediate field, and (v) the register contents. As shown in the diagram of Fig. 2, any corruption in the commit trace (i.e., the "Commit Trace Correct" check is False – left top branch of the diagram) can be categorized into only one IMM (six IMM nodes exist on the left branch). It is clearly shown by these conditions (i.e., six different conditions are checked for a potential commit trace corruption) that one and only one IMM class node can be reached (i.e., *mutual exclusiveness*). Also, since these conditions cover any parameter of the instruction, there can be no other class for IMM (i.e., *completeness*).


Commit trace format:

```
     Tick/Cycle      :     PC     :    Instruction    :   Register Content
175275342634194500 : 0x427dd0 : orr x0, xzr, x25 : D=0x0000000000463608
```

Fig. 2. Diagram for IMM classification process.

When a condition is checked and found false, the remaining (not checked) conditions are "don't care". For example, for the IRP class node to be reached, the PC must be correct, and the opcode must be corrupted. All other parameters that are associated with the instruction are "don't care" (the "64" label in the IRP node denotes the number of all combinations of the 6 "don't care" conditions). In another example, for the OFS class node to be reached, the PC, and the opcode must be correct, all operands must be defined in the ISA, and one of the operands must be wrong (but still in the ISA). The remaining 4 conditions are "don't care" (the "16" label in the OFS node shows the number of their combinations). The sum of all dark labels in the IMM class nodes of Fig. 2 is 256, which is the complete number of combinations of the 8 conditions (i.e., $2^8 = 256$ combinations).

One the other hand, if there is no deviation in the commit stage (i.e., the "Commit Trace Correct" check is True – right top branch of the diagram in Fig. 2), the categorization depends on the existence of output file. If no output file has been generated, this means that the fault may have caused a violation of a high-level condition, and a simulator assertion check has failed (i.e., PRE class). If, on the contrary, an output file has been generated there are only two options: if the output file does not differ from the fault-free one, the fault is categorized as Benign. If the output file is different from the fault-free one, the fault is categorized as ESC. Any fault that reaches the software belongs to *exactly* one of the 8 classes.

From the AVF point of view, an IMM may affect the program's operation or get masked. A fault that either reaches the software and gets masked by a program's operation (e.g., the corrupted register value is never be used) or it is Benign (it does not reach the software), it is a Masked fault for the AVF classification, since it does not affect the program. On the other hand, if the fault reaches the software (in an IMM class) and subsequently affects the program's operation, it is classified either as an SDC or as a Crash.

B. *IMMs Distribution*

A major finding of our HVF analysis and IMMs classes extraction, is that for each hardware structure, the distribution of IMMs is relatively uniform for every different program, which means that it is primarily an *invariant* of the hardware structure itself. Assume, for example, the data field array of L1 Data Cache or the Physical Register File. Most of the corruptions in these hardware structures result in DCR IMM, although there is a significantly low number of corruptions that lead to ETE and IFC IMMs, especially in the Physical Register File. This means that most fault occurrences affect the contents of registers (for the Register File) or memory words that are used for load or store instructions (for the L1 Data Cache), and there is a zero probability for any fault to manifest as any of the other IMMs (i.e., IRP, UNO, OFS, or PRE).

Fig. 3 shows the breakdown of IMMs for six hardware structures. Consider, for example, the leftmost graph of Fig. 3, which presents the IMM distribution for the data field of L1 Instruction Cache. It is clearly shown that all benchmarks (in the x-axis) provide a uniform distribution of IMM classes (the rightmost AVG bar in each graph, shows the arithmetic mean for each IMM of all benchmarks). Specifically, we can see that the IMM distribution of this hardware structure is as follows: 1% IFC, 11% IRP, 10% UNO, 66% OFS, 1% DCR, 11% ETE, and 0% PRE. In the next subsection we explore the probability of these IMMs distribution to result in a specific final fault effect (i.e., Masked, Crash, SDC). Similarly, we can see in the next three graphs of Fig. 3 that all benchmarks provide relatively uniform IMM distribution for each different structure. Very few benchmarks slightly deviate, but the difference is always less than the error margin of 2.88%. ROB/LQ/SQ are shown together since faults in these structures manifest only as PRE IMM.

The IMM distribution strongly depends on the operation of hardware structures and their role in the microarchitecture.

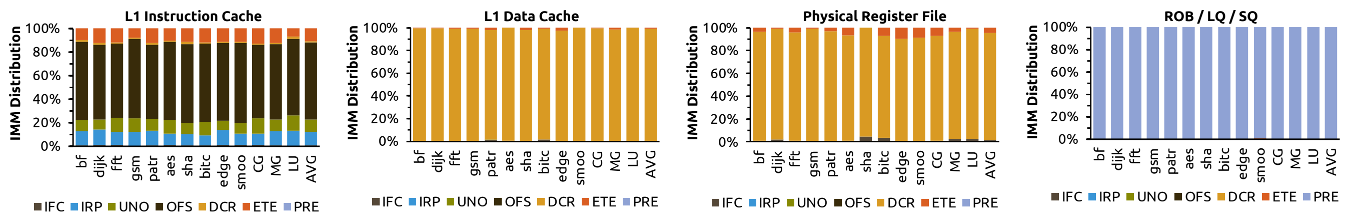


Fig. 3. Breakdown of IMMs (y-axis) for all benchmarks (x-axis) for (a) the data field of L1 Instruction Cache, (b) the data field of L1 Data Cache, (c), the Physical Register File, and (d) the Reorder Buffer (ROB), Load Queue (LQ) and Store Queue (SQ).

Faults in L1 instruction cache are most likely to corrupt the operands and/or the immediate field because they both occupy more bits than the opcode (i.e., high UNO and OFS IMM classes). Also, it is likely for a fault to corrupt the opcode (IRP), and the execution flow (ETE) (since both IMMs strongly depend on the executed instructions retrieved from the L1 instruction cache). Another example is the Physical Register File, which mainly consists of data values and memory addresses of inflight instructions. Thus, any corruption in it will primarily affect the content (i.e., DCR) and in very rare cases the execution flow. So, it is totally unlikely that a fault in the register file leads to any other IMM (e.g., IRP, UNO, OFS, and PRE practically cannot happen). Moreover, faults in the ROB/LQ/SQ lead to 100% PRE IMM class because faults in these structures cannot be architecturally visible, primarily due to dependence graph checks failures before the commit stage. Overall, there are two major insights: (1) different hardware structures provide different IMM distribution, and (2) different programs provide relatively uniform IMM distribution for a specific hardware structure, with slight differences.

An important observation is that the difference among benchmarks for faults on the same structure is only *the absolute number of corruptions* (faults that are not benign) *and not the distribution*, because the absolute number depends on the program (and on the microarchitecture, but the experiments are conducted for the same microarchitecture).

C. Final Fault Effects Distribution

Apart from the IMMs distribution, which, as we demonstrated, is relatively uniform across all benchmarks we studied for the same hardware structure, another important insight is that different benchmarks provide quite uniform fault effect classification (Masked, Crash, or SDC), depending on the IMM distribution and the hardware structure. Again, there may be slight differences among benchmarks, but they are around the statistical error margin.

In Fig. 4 we can see the final fault effects distribution for each IMM for L1 Instruction Cache and for all benchmarks.

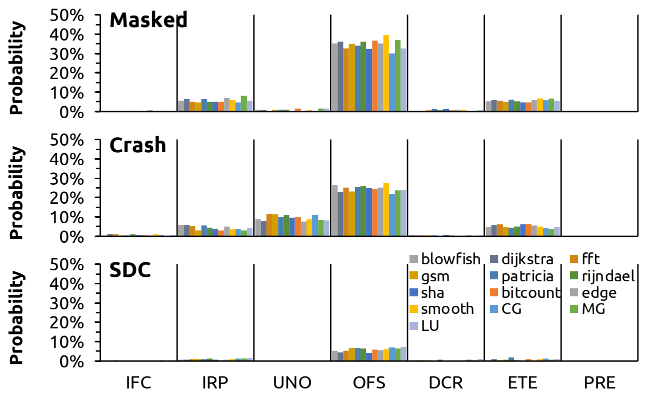


Fig. 4. The final fault effect probabilities for each IMM, for the L1 Instruction Cache and for all benchmarks.

Specifically, at the top graph of Fig. 4 we can see the probability of each IMM to result in Masked fault effect, at the middle graph we can see the probability of each IMM to result in Crash fault effect, and at the bottom graph of Fig. 4 we can see the probability of each IMM to result in SDC. It is clearly shown in these graphs that for every IMM (x-axis), any benchmark has practically the same probability to result in either Masked, SDC, or Crash. Specifically, the standard deviation among all benchmarks of any IMM category ranges between 0.1% and 2.4%. For example, approximately 35% of all IMMs in all benchmarks that are categorized as OFS results in Masked, while 25% of all IMMs in all benchmarks results in Crash. Note that the sum of Masked, SDC and Crash probabilities for all IMMs of each benchmark will naturally be 100%. Overall, this new major insight is that all benchmarks provide relatively uniform probability of the final fault effect classification, which depends on the IMM distribution.

D. Putting It All Together – IMM Weights

Our extensive HVF analysis reveals three major insights, as we discussed in the previous subsections, regarding the IMM distribution and their correlation to the final fault effect. The common point of all insights is that the observations we made rely on the fact that the IMM distributions and their corresponding final fault effects are relatively uniform for any benchmark when the faults reside in a specific hardware structure. This means that depending *only* on the hardware structure, it is feasible to elicit the final fault effect classification having only the IMM distribution and a predefined weighting factor for each IMM. Since different programs provide different absolute fault numbers for the IMM classes, but the same distribution and the probability of each fault effect class, it is straightforward to calculate the final AVF (i.e., the final fault effect classification) for an unknown workload with less that 2.88% accuracy loss (i.e., the same accuracy loss as in an exhaustive end-to-end SFI).

What makes this observation vital is that the simulation time for IMM classification (i.e., the HVF measurement) is significantly faster than running entire end-to-end AVF experiments. The reason is that for the IMM classification the simulation time is determined as the elapsed time between the fault injection and the fault manifestation (i.e., when the fault reaches the commit stage of an out-of-order microprocessor) or equivalently is based on fast HVF measurements. After that time, the simulation finishes. On the other hand, for an end-to-end AVF analysis, the simulation time is determined as the elapsed time between the fault injection and the end of the program's execution, to be able to classify not only Crash fault effects, but also any Masked or SDCs.

However, apart from this observation, which can accurately classify the fault effects of a program (i.e., the AVF estimation) and significantly accelerates the simulation process, we demonstrate in Section V.A that there is also another property that can further accelerate the simulation time for the IMM classification. Therefore, according to these insights, and since the fault effect classification is relatively uniform for each IMM and structure across all benchmarks, we can safely consider the IMM classification along with some weights per hardware structure and per IMM to elicit the final fault effect classification (i.e., the AVF). We simply consider as weights the arithmetic mean values (i.e., averages) for each fault effect of each IMM across all benchmarks. Fig. 5 presents the arithmetic mean for each fault effect (i.e., Masked, Crash, SDC) of each IMM across all benchmarks. Clearly, since all benchmarks provide relatively uniform probability of fault effect classification for each IMM with the standard deviation being less than 2.4% (see the discussion in subsection III.C and Fig. 4), the arithmetic mean shown in Fig. 5 is pretty close to the actual values for every benchmark.

Note that in each graph, the sum of all probabilities should be equal to 100%. However, the graph of L2 cache, sum up to less than 100% because faults in this structure can also result in the ESC IMM. The ESC IMM is considered in the AVF but is not shown in these graphs because the ESC faults cannot be identified as corruptions in the commit trace analysis (see discussion in subsection IV.D).

IV. Proposed Methodology

In this section, we built on top of the previous insights a novel microarchitecture-driven, fast, and accurate vulnerability evaluation methodology, which takes the full advantage of the SFI accuracy, but significantly increases the simulation speed (Section V discusses the speedup calculations). The proposed SFI-based methodology consists of five distinct phases: (1) Configuration phase, (2) Microarchitecture-Detailed Simulation phase, (3) IMM Classification phase, (4) Effects Classification phase, and (5) the Final Cross-Layer AVF Evaluation phase. The methodology flow is illustrated in Fig. 6 and described in the next subsections.

A. *Configuration Phase*

In this phase, the program, the initial fault list, and the target hardware structure are defined. The number of faults depends on the significance of the statistical sampling, i.e., the error margin and the confidence level. To our knowledge, *there is no other fault injection study or vulnerability evaluation methodology in the literature that considers all hardware structures of a modern microprocessor chip*. The proposed SFI-based methodology can effectively be applied to any structure of a microprocessor considering as the fault origin any hardware bit of a microarchitecture.

B. *Microarchitecture-Detailed Simulation Phase*

In this phase, the program runs in a microarchitecture-detailed simulation on top of gem5, considering all microarchitecture details of each microprocessor model (thus, any potential hardware masking effect of the fault). After the fault is injected, the simulation continues until the fault becomes visible to the software (i.e., the fault passes the commit stage). This corresponds to the clock cycle, in which the first instruction that is affected by the fault, commits to the architectural state. After that moment, the fault can be considered to have propagated to the software layer. Such faults, that eventually reach the software layer, are categorized through the IMMs as described in Section II.A. Faults that did not reach the software layer (thus, due to hardware masking, the program's execution will be eventually correct, and the fault cannot affect the program; the Fault Case 1 in Fig. 6) are categorized as *Benign* faults, since they cannot reach the software, and thus, the program's output.

To speed-up the simulation, several techniques have been implemented in the past to save simulation time in the absence of faults, both in pre- and post-injection periods without affecting the accuracy of the measurement. For example, checkpointing is used to skip the pre-injection period [18]. Note that these speed-up techniques are well-known in the literature and all the execution times reported in our paper *do not consider these techniques as a new contribution of ours*. This means that for a fair comparison between the proposed methodology and the exhaustive SFI, we consider in both cases these well-known speed-up techniques. Our speedups are further steps ahead.

C. *IMM Classification Phase*

When a fault reaches a visible point of the ISA layer, it is considered in the IMM classes, and there is no need to proceed with the simulation. For example, as we can see at the bottom of Fig. 6, every committed instruction is compared to its corresponding fault-free one (i.e., Fault Case 2). In this example, there is a deviation between the fault-free and faulty execution. The executed instruction (i.e., in Fault Case 2) has

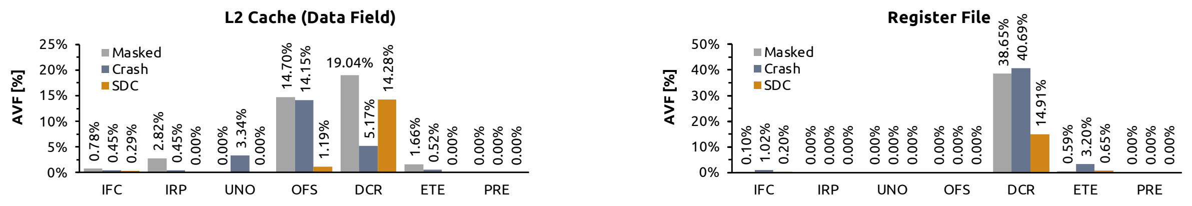
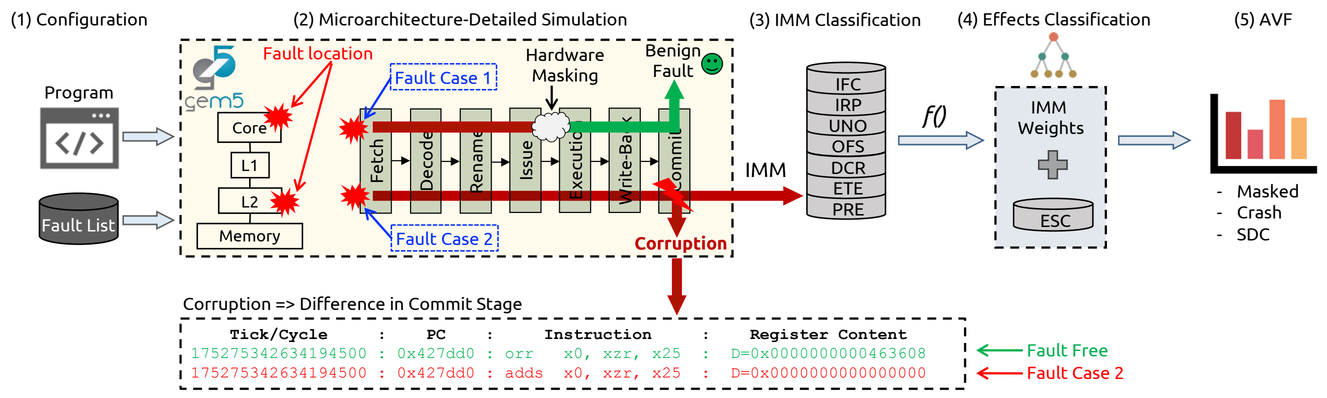


Fig. 5. Breakdown of the IMMs (x-axis) and their final fault effect. Numbers on bars are the averages across all benchmarks.



```
     Tick/Cycle       :    PC     :     Instruction     :    Register Content
175275342634194500 : 0x427dd0 : orr    x0, xzr, x25  :  D=0x0000000000463608
175275342634194500 : 0x427dd0 : adds   x0, xzr, x25  :  D=0x0000000000000000
```

Fig. 6. Microarchitecture-Driven methodology illustration.

committed in the correct cycle, with a correct Program Counter, but due to the fault occurrence, the committed instruction is different than the fault-free case. Apparently, the destination register contents are also wrong, but this is due to the execution of wrong instruction. In that case, this corruption in the commit stage will be categorized as an IRP (Table I). In this phase, the proposed methodology considers only the IMMs (i.e., the faults that eventually reach the software layer). The IMM classifier, organizes the faults into IMM groups as discussed in subsection III.A.

D. *Effects Classification Phase*

In this phase, the classifier will automatically categorize the fault effects as Masked, or Crash or SDC by applying to each IMM group, which was provided by the phase 3, the corresponding weighting factor depending on the target hardware structure (as discussed in subsection III.D). However, in this phase it is also required to consider the potential Escaped (ESC) IMM shown in the last row of Table I.

The ESC IMM was initially defined in [14] and determines the effect of a group of faults that hit a part of the program's output which is exposed in any cache level (only the cache arrays that store data) but will not pass again through the program trace. Assume that a fault happens on a modified cache line which contains data that are part of the program output. If the data of the cache line are not used again by the program (i.e., they are not read again by an instruction), they will be eventually written back without ever being read again by the microprocessor (i.e., they will not pass again through the program trace) and there is no further masking opportunity neither at the microarchitecture nor at the software layer. Since these data are part of the program output, the I/O device accesses this chunk of data, through a DMA controller, and thus, the program's output will be certainly corrupted (i.e., SDC). Therefore, any ESC IMM can only be considered in a cache level and *can only result in either Masked (i.e., if the fault does not affect the output data) or SDC (i.e., if the fault does affect the output data)*. Faults (in a cache level only) that result in ESC IMM are impossible to be identified during phase 3 of the proposed methodology. The reason is that faults that eventually result in ESC IMM are initially determined as *Benign* faults (since the fault occurrence will not pass through the program trace, so there is no detection capability at the hardware or software).

Therefore, to identify and consider any ESC IMM that its fault effect (i.e., SDC) will eventually affect the final fault effect classification, under normal circumstances, it would be necessary to run the program until the end for all Benign fault cases. On average, Benign faults in the L1 Data Cache are half (or more) of the total faults, and for the L2 cache they are 80% (or more) of the total faults. Therefore, a complete end-to-end execution of a workload for each Benign fault to detect any potential ESC IMM, which occur in the cache arrays of the microprocessor, would require extensively long simulation time. However, we experimentally found that the amount of ESC IMM that results in SDC, has a *strong* correlation to the output data size of each program, and apparently to the number of Benign faults that each program provides for each hardware structure.

When the program's output is very small (e.g., *sha* and *bitcount* benchmarks have total output size less than 1KB), the probability of an ESC IMM to affect the program's output is zero. The reason is that a cache level stores only a small part of the total output at every given moment, so since the total output size is relatively small, it is extremely unlikely for a fault to hit those (few) words that are stored in cache just before they are written in the output file. On the other hand, when the program's output is relatively large (e.g., *blowfish* and *rijndael* have output data greater than 3MB), the probability of an ESC IMM to affect the program's output is getting higher as the output size gets high (i.e., the probability is proportional to the output size). This process is performed offline during the phase 4 of the proposed methodology. We use the following empirical equation to estimate the percentage of Benign faults that will eventually turn into ESC faults. The equation shows that the ESC faults percentage depends on the output size (in KBs) and on the Benign faults counts for a pair of structure and benchmark.

$$ESC[\%] = \frac{Output_{Size}}{1024} \times \frac{Faults_{Total} - Faults_{Benign}}{(Faults_{Total} + Faults_{Benign})^2}$$

Consider, for example, *blowfish* and *rijndael* benchmarks which have virtually the same output data size (i.e., 3.16MB and 3.17MB respectively), however, *blowfish* has a larger number of Benign faults than *rijndael*. To this end, *blowfish* provides more ESC faults, which, in turn, result in SDCs. This is considered in phase 5 of the methodology to augment the final estimation considering the ESC IMM as well. We consider both the count of Benign faults and the output size of the benchmark together. Therefore, since the number of Benign faults strongly depends on the microarchitecture, given a specific benchmark, the accuracy of the equation above should hold true for every different microarchitecture. Fig. 7 demonstrates (in absolute fault numbers) the accuracy of predicted ESC faults for each benchmark for both the tag and data fields of L1 Data Cache and for L2 Cache (since the ESC faults can only occur in data cache arrays). Each dot in all graphs represents a benchmark. In the ideal scenario, all dots should be on the diagonal line. However, as we can see in Fig. 7, there are small divergences between the real and the predicted values. We will demonstrate in Section V that these small divergences do not affect the final AVF estimation.

E. Final Cross-Layer Vulnerability (AVF)

In this phase, the final AVF evaluation results are getting parsed and recorded to a results file.

V. Evaluation & Speed-Up

A. Further Speed-Up Opportunities

In the previous sections, we demonstrated how the proposed methodology can elicit the final AVF from the IMM categorization of each program for any target structure. Categorizing all fault occurrences that provide corruptions in the commit stage into IMMs (i.e., there is a deviation in a committed instruction compared to the fault-free execution), requires significantly less simulation time than end-to-end AVF calculation, since the simulations run until the first corruption in the commit stage and not until the end of the program's execution. This process significantly reduces the total simulation time. However, this process is still missing important portions of the fault simulation timeline. As we discussed, a fault occurrence can also be characterized as *Benign* during the IMMs classification phase.

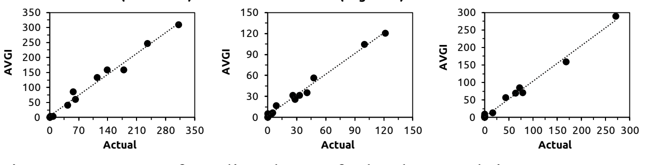


Fig. 7. Accuracy of predicted ESC faults that result in SDC.

However, since Benign faults do not corrupt any microarchitectural or software resource, there is no deviation in the commit stage (i.e., there is no IMM categorization). Therefore, the simulation should run until the end of the program. The number of Benign faults is relatively large, although it is different across workloads and hardware structures. This means that there are many injected faults that still require end-to-end simulated executions. An important aspect that may virtually eliminate the simulation time of any Benign fault for any hardware structure, is the *effective residency time* of a fault in a hardware resource: the time between the fault occurrence and its manifestation [18]. Faults in hardware structures that are located deep in the microprocessor's pipeline (e.g., register file, ROB, LQ, SQ, and TLBs) have significantly less effective residency time than in other structures that are out of the critical path (e.g., cache memories and especially the lower cache levels). Based on both the experimental study in Section III and the effective residency time, we gathered a set of important observations, which suggest opportunities to further speedup the assessment.

Consider, for example, the Physical Register File. Typically, two source and one destination registers are allocated for each instruction. The true dependencies are addressed when a static instruction is renamed, and its operands may change to temporal operand resources (i.e., register identifiers). During the commit stage, the order is restored, and the rename map is changed to reflect the current architectural state. So, the primary distinction between architectural and physical registers is the effective residency time. Architecturally mapped registers, which are part of the architectural state, may be used millions of cycles later since their allocation, or not at all. Physical registers, on the other hand, remain active during the lifetime of their corresponding instruction in the pipeline (usually a few clock cycles in total). This means that any fault in a physical register (which is the case in any microarchitecture-level reliability study) has a small probability (i.e., a short timeframe) to corrupt the architectural state (similar observations are made in [18]).

As a matter of fact, according to our experiments, we found that in the Physical Register File any potential corruption can be detected within a maximum interval of 1M cycles from the fault injection time. ROB, LQ, and SQ, which are also located deep in the pipeline, are assigned to support the order and the dependency resolution of all inflight instructions into the pipeline. However, we found that the residency time for these structures depends on the execution time of each benchmark, and it is at most 3% of its total cycles. Similarly, for TLBs any potential corruption can be detected during 5M cycles since the fault injection.

Similar observations are valid for the L1 instruction cache. During the fetch stage, an instruction word is fetched from the L1 instruction cache. If the fetched instruction is not a control instruction (i.e., there is a probability to be mispredicted), there is high probability to be used by the program

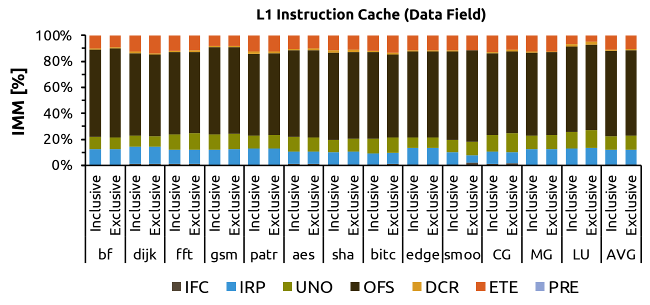


Fig. 8. IMM distribution for the entire execution (inclusive) and only for the necessary manifestation period (exclusive).

flow, and thus, to commit shortly. We experimentally found that any corruption in L1 instruction cache can be detected during 7M cycles since the fault injection. Fig. 8 presents a comparison of IMM distribution when considering the entire program's execution (shown with *inclusive* label in the x-axis of Fig. 8) and when considering only the necessary fault manifestation period of 7M cycles (shown with *exclusive* label in the x-axis of Fig. 8) for the L1 Instruction Cache. It is clearly shown in Fig. 8 that for each benchmark the IMM distribution is virtually the same between inclusive and exclusive cases.

Therefore, by stopping the fault injection simulations for each structure during the clock cycles that are suggested by the effective residency time analysis, there should not be any significant accuracy loss in the IMM distribution results. On the other hand, the effective residency time for the L1 Data Cache and L2 Cache is longer than the deep pipeline structures. The requests in these memory structures are primarily related to the memory operations of each workload, in which prefetching requests or unresolved memory access dependencies can also occur. For that reason, we found that any potential corruption in L1 Data Cache can be detected during up to 50M cycles, and in L2 Cache during up to 80M cycles. Fig. 9 shows an illustrative example for this process. As we can see, the simulation time after the defined residency time for each memory structure cannot offer any additional information for the corruption. It is only required for the exhaustive AVF analysis to consider all Masked and SDC cases. The proposed methodology does not require the simulation to run

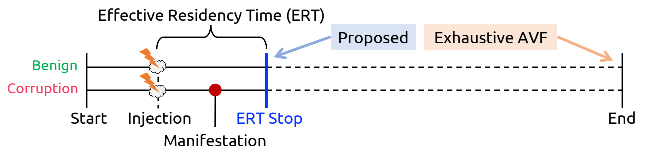


Fig. 9. Effective residency time illustration.

until the program finishes, because any fault manifestation (i.e., the first fault effect to the software layer) occurs until the defined effective residency time (ERT Stop).

Note that all the above timeframes for each structure (i.e., the maximum clock cycles needed until the fault commits based on the effective residency analysis; shown in Table II) constitute *pessimistic timeframes* and are related to programs with very large input datasets. This means that all previously determined timeframes cover a high percentage of any fault occurrence in any program used in this study, although we have found that most fault occurrences can be detected during significantly shorter intervals, which suggests even shorter simulation times. However, this study aims at high accuracy of IMM categorization and of AVF estimation, so we choose to consider the most pessimistic cases paying the price of a bit longer simulation time.

B. *AVF Evaluation Speed-Up*

Table II summarizes the findings discussed in the previous subsections. It presents the 12 hardware structures, the maximum simulation cycles for each structure (based on the effective residency time), the total AVF evaluation time considering the proposed methodology and the traditional (accelerated; see subsection IV.B) SFI flow, the speedup of the proposed methodology (separately, according to the contribution of each insight), and the speedup of the proposed methodology in orders of magnitude. All presented times are shown in days, using 192 computational cores of two servers with two AMD EPYC^{TM} 7402 microprocessors each (i.e., the rack servers where all experiments of this study have been conducted). Each row of the table consists of the time needed for all 13 benchmarks and 2,000 fault injections (i.e., for each structure 26,000 injections for the proposed methodology and

TABLE II
AVF ASSESSMENT TIME FOR THE PROPOSED METHODOLOGY AND AN ACCELERATED TRADITIONAL SFI-BASED METHODOLOGY.

	Maximum Sim Cycles	**AVGI (Days)**	**Traditional SFI (Days)**	**Speedup**		**Orders of Magnitude**
				Insight 1&2	*Insight 3*	
RF	1M	0.11	37.08	6.2x	330.8x	**2.5**
DTLB	5M	0.38	36.96	8.6x	88.4x	**2.0**
ITLB	5M	0.44	38.25	6.9x	80.1x	**1.9**
L1I (Data)	7M	0.60	30.23	7.2x	42.8x	**1.7**
L1D (Tag)	10M	0.76	27.91	7.1x	29.9x	**1.6**
ROB	3%	1.19	36.28	6.0x	24.0x	**1.5**
SQ	3%	1.19	35.67	5.9x	24.1x	**1.5**
LQ	3%	1.19	35.73	5.9x	24.1x	**1.5**
L1I (Tag)	1M	0.79	27.13	7.3x	36.7x	**1.5**
L2 (Tag)	20M	2.27	36.47	6.2x	9.8x	**1.2**
L1D (Data)	50M	3.23	33.59	6.5x	3.5x	**1.0**
L2 (Data)	80M	6.72	39.21	5.5x	0.5x	**0.8**
	Total (Days)	**18.9**	**414.5**			

26,000 for the traditional SFI). Consequently, we can see that in total for all 12 hardware structures for 13 benchmarks, we need 18.9 days based on the proposed methodology, while based on an already accelerated traditional SFI-based AVF assessment (see the related discussion in subsection IV.B), we needed about 14 months (414.5 days). Apparently, using more servers for the simulations, both numbers would be proportionally reduced depending on the available computational cores. Therefore, the effective speedup, without compromising the accuracy of the final estimations (section V.C), ranges between 6x to 337x depending on the structure, compared to the accelerated SFI, while for the entire CPU is 22x.

The first two insights (*Insight 1&2* in Table II) account for the speedup achieved because the simulation is stopped at the time when the first corruption occurs. This means that all Benign faults *do not take any speedup advantage* of both Insights 1 and 2 over the exhaustive SFI. The reason is that there is no deviation from the fault-free execution, and thus, the simulation needs to run until the end of the program (Insight 3 solves this problem). We have previously discussed that aspect in Section V.A. It is important to stress, however, that *all three insights constitute the entire proposed methodology and cannot be used separately*. Specifically, the main reason why we can leverage insight 3, is because the proposed methodology keeps track only the first corruption of faults at the software level, it categorizes them into IMM classes, and then it elicits the final fault effects through the IMM categorization. On the contrary, we cannot apply only insight 3 on the exhaustive SFI methodology, because there is no observation point during the assessment that could lead the estimation to stop after a few clock cycles. In the typical exhaustive AVF assessment, all experiments need to run until completion to provide the output file and decide for Masked or SDC outcome.

C. *Methodology Accuracy Evaluation*

In this section, we demonstrate the accuracy of results of the proposed methodology. Fig. 10 presents the accuracy evaluation between the real AVF assessment retrieved from the exhaustive traditional SFI-based AVF analysis ("Real" labels in the x-axes), and the AVF assessment provided by the proposed AVGI methodology ("AVGI" labels in the x-axes), as it is described in Section IV. Each graph in Fig. 10 represents the comparative results for each hardware structure and for all benchmarks. It is clearly shown in these graphs, that the accuracy of the proposed methodology is high, although *it does not consider any pruning technique* (every injected fault is individually considered – but faster). This means that the proposed methodology provides an exhaustive SFI, and thus, it retains the statistical significance of the experiments and keeps the statistical error margin at the lowest levels. As we can see in all graphs of Fig. 10, the Masked, SDC, and Crash fault effect probabilities are virtually the same between the results of the proposed AVGI methodology and the traditional SFI. Moreover, it is essential to note that the calculated SDC fault effect probabilities by the proposed methodology are very close to the real ones.

Fig. 11 shows the Failures in Time (FIT) rate for each structure and for the entire microprocessor chip (the rightmost graph). FIT rate of a device estimates the number of failures that can be expected in one billion (10^9) device-hours of operation. The FIT rate of the entire microprocessor chip is calculated by adding the individual FITs of the structures. For the calculation of the FIT, we use the raw FIT rate per bit, which is 9.39×10^{-6} FIT/bit for Cortex-A72-like, as proposed in [38]. FIT rates can provide a clearer view of the proposed methodology's accuracy since they combine all important reliability metrics of *all* hardware structures and the *entire* chip for all benchmarks consolidated. It is clearly shown in Fig. 11 that the accuracy loss of our methodology compared to the exhaustive AVF analysis is extremely low. Specifically, for any hardware structure the difference is at most 1.45%, and for the entire chip evaluation it is 0.2%.

VI. A CASE STUDY ON A DIFFERENT ISA & MICROARCHITECTURE

To further support the effectiveness of the proposed methodology across ISAs and microarchitectures, we conduct a case study employing a different ISA and microarchitecture. We model an Armv7 ISA, considering the Arm Cortex-A15-like microarchitecture, using the same 10 MiBench benchmarks (see subsection II.D). From the faults behavior and AVF point of view of each benchmark, there are several differences between the two CPU models. We conduct the case study for the same 12 hardware structures for Arm Cortex-A15, however, due to space limitations we present the graphs for only 3 but important structures, the L1 Instruction and Data Caches, and the Register File.

Fig. 12 shows the comparative AVF evaluation results for the 3 major structures for Armv7 ISA and for all benchmarks. Each graph represents one structure, showing the AVF results using the exhaustive SFI ("Real" label in the x-axes) and the

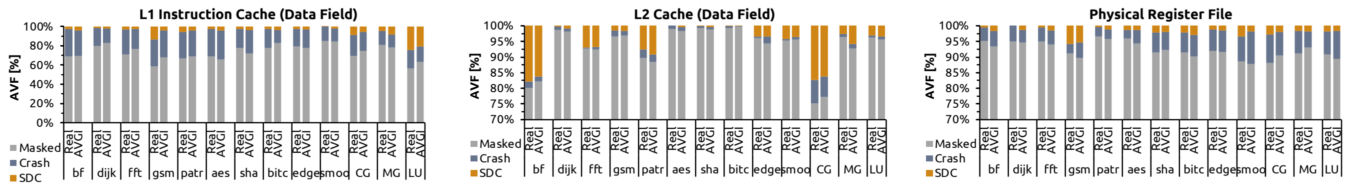


Fig. 10. Accuracy evaluation of the proposed methodology compared to the real AVF from exhaustive AVF analysis.

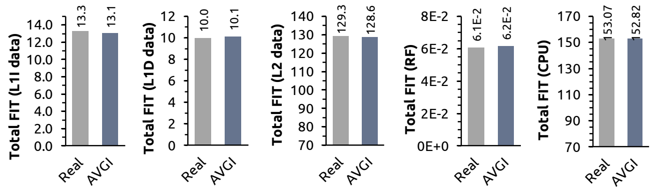


Fig. 11. Failure in Time (FIT) rates for each structure and the entire microprocessor chip (rightmost graph).

proposed SFI-based methodology ("Predict" label in the x-axes). For the proposed methodology's results, we use the proposed methodology as it is discussed in Section IV. As we can see in Fig. 12, all benchmarks for all hardware structures provide extremely high accuracy of the AVF results (i.e., the fault effects classification of the full stack vulnerability) using the proposed methodology.

There are some divergencies between the AVF results of the proposed methodology and the exhaustive AVF SFI-based flow, however, they are in most of the cases lower than the statistical error margin of SFI. More importantly, we can see that the important SDC fault effect class is virtually equal to the ground-truth value provided by the exhaustive AVF analysis. This means, that the proposed methodology can effectively assess the AVF for any benchmark and structure of a modern microprocessor chip with high accuracy levels, but in extremely shorter time. Particularly, the proposed methodology can provide the AVF in up to 440x shorter time (for the Physical Register File) than an already accelerated SFI-based flow regarding the Armv7 experiments. Note that the speedup depends on the underlying microarchitecture, which affects both the execution time of the benchmarks and the elapsed time between the fault occurrence and its first deviation in commit stage. This is the reason that the proposed methodology on Cortex-A15 provide higher speedup than Cortex-A72.

VII. Discussion

A. *Multi-Core & Multi-Bit Faults*

We claim that AVGI can be employed to estimate the vulnerability to spatial multiple-bit faults, as well as for multi-threaded applications running on multicore microprocessors. It has been observed through physical experiments of accelerated beam testing [20] [39] that on-chip storage arrays can suffer multiple-bit flips in adjacent areas. Therefore, multiple-bit faults can affect neighboring bits of a hardware structure. This means that, even if a multiple-bit flip occurs, it cannot affect two different instructions at the same time. For example, it is unlikely due to the geometry of multiple-bit faults to affect both the content of a register (i.e., DCR) and the opcode of that instruction at the same time (i.e., IRP). Once the corruption occurs, it will certainly be considered by AVGI with the same estimation accuracy, as in the single-bit fault case. The final estimation results for multiple-bit fault will be, of course, higher than the single-bit faults, because the probability of a corruption in increased. However, it is likely for multiple-bit faults to affect neighboring bits of two neighboring words (e.g., in a cache line). In such a case, it AVGI may not consider the effects of both corruptions. The reason is that the proposed methodology strongly relies on the distribution of fault manifestations (i.e., the effects of hardware faults until they "touch" the software layer). Overall, considering either single-bit or multiple-bit faults, or single-core or multicore configuration, the eventual fault manifestations will be considered by the proposed methodology and can accurately deliver the AVF measurements.

B. *Accuracy Discussion for Different Microarchitetures*

The proposed methodology for AVF estimation strongly depends on the distribution of fault corruptions that are architecturally visible (i.e., affect the software), which is, by definition, based on (i) the microarchitecture, (ii) the workload, and (iii) the faults distribution. A fault occurrence is architecturally visible, if and only if, it does not get masked at the microarchitecture layer (i.e., hardware masking – Benign Faults). Hardware masking can occur if any of the following three conditions holds: (i) the fault affects an "invalid" entry (e.g., a physical register which is not currently mapped, or a prefetched cache line which is never used, etc.), (ii) the fault in an entry gets overwritten by another normal operation before it is used, (iii) the fault affects a mis-speculated instruction (i.e., the fault is discarded due to a pipeline flush). In any other case, a fault in a hardware structure will eventually be architecturally visible.

Consequently, given a certain workload, different microarchitectures can only affect the population of the Benign faults (and thus the absolute number of corruptions, since benign faults are complementary to corruptions) and not the statistical distribution of IMMs (i.e., fault manifestations). The reason is that, for a given workload, different microarchitectures can only affect the hardware masking. However, since hardware masking is unavoidable and depends only on the

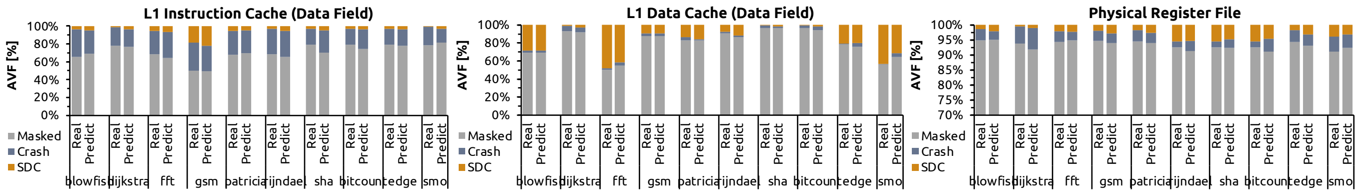


Fig. 12. Accuracy evaluation of the proposed methodology compared to exhaustive AVF analysis for Arm Cortex-A15.

microarchitecture (for a given workload and fault distribution), assume, for example, a microarchitecture "M1" with different branch prediction algorithm than microarchitecture "M2", and M1's branch prediction is less accurate than M2's (i.e., M1 has higher misprediction rate than M2). M1 will lead to more benign faults due to larger number of mispredictions, and thus, M1 and M2 have different hardware masking levels. However, even if the absolute number of corruptions is different between two microarchitectures, the statistical distribution of fault manifestations should not be changed due to the different hardware masking between M1 and M2 (within the range of the statistical error margin of 2.88%).

On the other hand, an architecturally visible fault may affect the program's output, if and only if, it does not get masked by the software [2] [3]. Software/Logical masking only depends on the program's flow and can occur when a corrupted register content or memory word do not affect the result of computations. Assume the following instruction: `and x0, x1, #0`. If the content of register `x1` was corrupted in any of the 32 least significant bits, the computation's result will be correct (i.e., Masked). Consequently, by categorizing the fault manifestations into IMMs, the final estimation is highly predictable (see sections III.C and III.D) because the distribution of the final fault effects (not the absolute number) of each IMM is relatively uniform for any workload of a certain hardware structure.

VIII. Related Work & Comparison

Microarchitecture-Level SFI: Kaliorakis *et al.* in [23] propose the MeRLiN methodology for accelerating AVF evaluation by using a combination of ACE analysis and fault injection. The main concept of MeRLiN is to use ACE analysis to get fast the cycle windows which are vulnerable, and then analyze the static instructions to even reduce the fault injection locations. To this end, MeRLiN can reduce the time needed for each fault injection campaign, since as the initial number of generated faults increases, the number of the remaining faults that will be used will be the same. This assumption, however, induces several errors in the final effects classification, since different number of loop iterations can provide different vulnerability effects, depending on the cycle, the dynamic instruction, the corrupted data, and the combination of them. On the other hand, in our methodology, in which no pruning or coarse-grained grouping is considered, the total simulation time needed is significantly low, and guarantees correct vulnerability results compared to the exhaustive AVF assessment, *for any number of faults*. However, our methodology could be complementary to MeRLiN's providing further speedups. Since our methodology only needs the potential deviations in the commit stage, it can complement MeRLiN to further reduce the simulation time by avoiding end-to-end executions that MeRLiN unnecessarily performs.

Another limitation of MeRLiN is that it requires *extensive development effort* for the implementation of ACE analysis not only for the register file, but also for the cache arrays, especially for their tag fields [40]. Its practicality and efficiency for address-based and instruction-based structures is questionable. On the other hand, our methodology requires significantly low development effort because only fault injection is required, and it can be easily applied to any hardware structure without any major modifications of the simulator and without any risky pruning phase.

ISA- and Software-Level SFI: ISA- or Software-level fault injection methods are widely used to evaluate the vulnerability at native speeds [6]-[13], [41]-[47]. To this end, several studies focus on detecting hardware faults by monitoring the software behavior [25] [48]-[50]. Although these techniques allow for long and realistic workload assessments, *they completely fail to capture the faults behavior considering the underlying hardware because the starting point of the experiment* (i.e., the origin of the injected fault) is a corrupted instruction (not a microarchitectural structure). It is recently demonstrated in [14] that any ISA- or software-level reliability evaluation method provides diverging vulnerability results. Therefore, even if such methods provide extremely fast the evaluation results, the final vulnerability measurements may be misleading and may instruct designers to take wrong decisions for protection schemes either at the software or at the hardware level.

IX. Conclusion

In this paper, we proposed AVGI, a novel AVF evaluation methodology, which is SFI-based and accurately delivers the cross-layer vulnerability assessment (i.e., AVF) for important microprocessor structures and every fault effect class. The proposed methodology is microarchitecture-driven and is based on three key insights of faults behavior: (1) the distribution of the introduced complete and mutually exclusive IMMs, (2) the final effects of faults on a specific hardware structure, and (3) the timeframe between the fault occurrence and its first corruption. We demonstrated that the proposed methodology delivers accurate per component and full CPU AVF measurements for two different ISAs, an Armv8 and an Armv7, in up to 337x and up to 440x, respectively, shorter time compared to an already accelerated exhaustive SFI. The proposed methodology also preserves the statistical significance of the experiments since no pruning of the initial fault list is employed.

Acknowledgment

This work was supported in part by the European Union's Horizon 2020 Research Programme under the Tetramax TTX Project under Grant 761349, in part by UniServer Project under Grant 688540, and by research gifts from Intel Research and Meta Research. The authors warmly thank the anonymous reviewers for their valuable comments and suggestions.

References

[1] R. Leveugle, A. Calvez, P. Maistri and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," 2009 Design, Automation & Test in Europe Conference & Exhibition, Nice, 2009, pp. 502-506, doi: 10.1109/DATE.2009.5090716.

[2] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36., San Diego, CA, USA, 2003, pp. 29-40, doi: 10.1109/MICRO.2003.1253181.

[3] S. Mukherjee, "Architecture Design for Soft Errors," Morgan Kaufmann Publishers Inc., San Francisco, CA, USA., 2008, [Online]. Available: https://doi.org/10.1016/B978-0-12-369529-1.X5001-0.

[4] V. Sridharan and D. R. Kaeli, "Eliminating microarchitectural de- pendency from Architectural Vulnerability," 2009 IEEE 15th International Symposium on High Performance Computer Architecture, Ra- leigh, NC, 2009, pp. 117-128, doi: 10.1109/HPCA.2009.4798243.

[5] V. Sridharan and D. R. Kaeli, "Quantifying Software Vulnerability," In Proceedings of the 2008 Workshop on Radiation Effects and Fault Tolerance in Nanometer Technologies (WREFT '08), Ischia, Italy, pp. 323–328, 2008, doi: 10.1145/1366224.1366225.

[6] Q. Lu, M. Farahani, J. Wei, A. Thomas, and K. Pattabiraman, "LLFI: An Intermediate Code-Level Fault Injection Tool for Hardware Faults," 2015 IEEE International Conference on Software Quality, Reliability and Security, Vancouver, BC, 2015, pp. 11-16, doi: 10.1109/QRS.2015.13.

[7] A. Jin, J. Jiang, J. Hu, and J. Lou, "A PIN-Based Dynamic Software Fault Injection System," 2008 The 9th International Conference for Young Computer Scientists, Hunan, 2008, pp. 2160-2167, doi: 10.1109/ICYCS.2008.329.

[8] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "FERRARI: a tool for the validation of system dependability properties," presented at the 1992 FTCS - The Twenty-Second International Symposium on Fault-Tolerant Computing, 1992.

[9] S. Jha et al., "ML-Based Fault Injection for Autonomous Vehicles: A Case for Bayesian Fault Injection," IEEE/IFIP International Confer- ence on Dependable Systems and Networks (DSN), 2019, doi: 10.1109/dsn.2019.00025.

[10] 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 Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVII), London, England, UK, pp. 123–134, 2012, doi: 10.1145/2150976.2150990.

[11] S. K. S. Hari, R. Venkatagiri, S. V. Adve and H. Naeimi, "GangES: Gang error simulation for hardware resiliency evaluation," 2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA), Minneapolis, MN, 2014, pp. 61-72, doi: 10.1109/ISCA.2014.6853212.

[12] R. Venkatagiri, A. Mahmoud, S. K. S. Hari, and S. V. Adve, "Approxilyzer: Towards a systematic framework for instruction-level approximate computing and its application to hardware resiliency," 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, 2016, pp. 1-14, doi: 10.1109/MICRO.2016.7783745.

[13] R. Venkatagiri et al., "gem5-Approxilyzer: An Open-Source Tool for Application-Level Soft Error Analysis," 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Portland, OR, USA, 2019, pp. 214-221, doi: 10.1109/DSN.2019.00033.

[14] G. Papadimitriou and D. Gizopoulos, "Demystifying the System Vulnerability Stack: Transient Fault Effects Across the Layers," 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), 2021, pp. 902-915, doi: 10.1109/ISCA52012.2021.00075.

[15] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The Gem5 simulator," ACM SIGARCH Computer Architecture News, vol. 39, no. 2, 1-7, Aug. 2011, doi: 10.1145/2024716.2024718.

[16] H. Cho, S. Mirkhani, C. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," 2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC), Austin, TX, 2013, pp. 1-10, doi: 10.1145/2463209.2488859.

[17] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim, "Robust system design with built-in soft-error resilience," in Computer, vol. 38, no. 2, pp. 43-52, Feb. 2005, doi: 10.1109/MC.2005.70.

[18] A. Chatzidimitriou and D. Gizopoulos, "Anatomy of microarchitecture-level reliability assessment: Throughput and accuracy," 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Uppsala, 2016, pp. 69-78, doi: 10.1109/ISPASS.2016.7482075.

[19] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris and D. Gizopoulos, "Differential Fault Injection on Microarchitectural Simulators," 2015 IEEE International Symposium on Workload Characterization, Atlanta, GA, 2015, pp. 172-182, doi: 10.1109/IISWC.2015.28.

[20] A. Chatzidimitriou, G. Papadimitriou, C. Gavanas, G. Katsoridas and D. Gizopoulos, "Multi-Bit Upsets Vulnerability Analysis of Modern Microprocessors," 2019 IEEE International Symposium on Workload Characterization (IISWC), Orlando, FL, USA, 2019, pp. 119-130, doi: 10.1109/IISWC47752.2019.9042036.

[21] NAS Parallel Benchmarks Suite, https://www.nas.nasa.gov/software/npb.html.

[22] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge and R. B. Brown, "MiBench: A free, commercially representative embedded benchmark suite," Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538), Austin, TX, USA, 2001, pp. 3-14, doi: 10.1109/WWC.2001.990739.

[23] M. Kaliorakis, D. Gizopoulos, R. Canal and A. Gonzalez, "MeRLiN: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," 2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), 2017, pp. 241-254, doi: 10.1145/3079856.3080225.

[24] K. R. Walcott, G. Humphreys, and S. Gurumurthi, "Dynamic prediction of architectural vulnerability from microarchitectural state," 2007 ACM/IEEE Annual International Symposium on Computer Architecture (ISCA), 2007, pp. 516-527, doi: 10.1145/1250662.1250726.

[25] N. J. Wang and S. J. Patel, "ReStore: symptom based soft error detection in microprocessors," 2005 International Conference on Dependable Systems and Networks (DSN'05), 2005, pp. 30-39, doi: 10.1109/DSN.2005.82.

[26] M. Li, P. Ramachandran, U. R. Karpuzcu, S. K. S. Hari and S. V. Adve, "Accurate microarchitecture-level fault modeling for studying hardware faults," 2009 IEEE 15th International Symposium on High Performance Computer Architecture, 2009, pp. 105-116, doi: 10.1109/HPCA.2009.4798242.

[27] X. Li, S. V. Adve, P. Bose and J. A. Rivers, "Online Estimation of Architectural Vulnerability Factor for Soft Errors," 2008 International Symposium on Computer Architecture, 2008, pp. 341-352, doi: 10.1109/ISCA.2008.9.

[28] A. Chatzidimitriou, P. Bodmann, G. Papadimitriou, D. Gizopoulos and P. Rech, "Demystifying Soft Error Assessment Strategies on ARM CPUs: Microarchitectural Fault Injection vs. Neutron Beam Experiments," 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Portland, OR, USA, 2019, pp. 26-38, doi: 10.1109/DSN.2019.00018.

[29] G. Papadimitriou and D. Gizopoulos, "Anatomy of On-Chip Memory Hardware Fault Effects Across the Layers," in IEEE Transactions on Emerging Topics in Computing, 2022, doi: 10.1109/TETC.2022.3205808.

[30] P. R. Bodmann, G. Papadimitriou, R. L. R. Junior, D. Gizopoulos and P. Rech, "Soft Error Effects on Arm Microprocessors: Early Estimations versus Chip Measurements," in IEEE Transactions on Computers, vol. 71, no. 10, pp. 2358-2369, 1 Oct. 2022, doi: 10.1109/TC.2021.3128501.

[31] Y. Sazeides, A. Gerber, R. Gabor, A. Bramnik, G. Papadimitriou, D. Gizopoulos, C. Nicopoulos, G. Dimitrakopoulos, and K. Patsidis, “IDLD: Instantaneous Detection of Leakage and Duplication of Identifiers used for Register Renaming”, ACM/IEEE International Symposium on Microarchitecture (MICRO 2022), Chicago, Illinois, USA, October 2022.

[32] P. Bodmann, G. Papadimitriou, D. Gizopoulos and P. Rech, "The Impact of SoC Integration and OS Deployment on the Reliability of Arm Processors," 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2021, pp. 223-225, doi: 10.1109/ISPASS51385.2021.00040.

[33] N. J. George, C. R. Elks, B. W. Johnson, and J. Lach, "Transient fault models and AVF estimation revisited," 2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN), Chicago, IL, 2010, pp. 477-486, doi: 10.1109/DSN.2010.5544276.

[34] D. S. Khudia and S. Mahlke, "Harnessing Soft Computations for Low-Budget Fault Tolerance," 2014 47th Annual IEEE/ACM International Symposium on Microarchitecture, Cambridge, 2014, pp. 319- 330, doi: 10.1109/MICRO.2014.33.

[35] A. A. Nair, L. K. John, and L. Eeckhout, "AVF Stressmark: Towards an Automated Methodology for Bounding the Worst-Case Vulnerability to Soft Errors," 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture, Atlanta, GA, 2010, pp. 125-136, doi: 10.1109/MICRO.2010.34.

[36] Z. Zhao, D. Lee, A. Gerstlauer, L. K. John, "Host-compiled reliability modeling for fast estimation of architectural vulnerabilities", SELSE 2015.

[37] V. Sridharan and D. R. Kaeli, "Using Hardware Vulnerability Factors to Enhance AVF Analysis," In Proceedings of the 37th annual international symposium on Computer architecture (ISCA '10), Saint-Malo, France, pp. 461–472, 2010, doi: 10.1145/1815961.1816023.

[38] G. Papadimitriou and D. Gizopoulos, "Characterizing Soft Error Vulnerability of CPUs Across Compiler Optimizations and Microarchitectures," 2021 IEEE International Symposium on Workload Characterization, 2021, doi: 10.1109/IISWC53511.2021.00021.

[39] E. Ibe, H. Taniguchi, Y. Yahagi, K. Shimbo, and T. Toba, "Impact of Scaling on Neutron-Induced Soft Error in SRAMs from a 250 nm to a 22 nm Design Rule," in IEEE Transactions on Electron Devices, vol. 57, no. 7, pp. 1527-1538, 2010, doi: 10.1109/TED.2010.2047907.

[40] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee and R. Rangan, "Computing architectural vulnerability factors for address-based structures," 32nd International Symposium on Computer Architecture (ISCA'05), 2005, pp. 532-543, doi: 10.1109/ISCA.2005.18.

[41] J. Carreira, H. Madeira and J. G. Silva, "Xception: Software Fault Injection and Monitoring in Processor Functional Units," in Conference on Dependable Computing for Critical Applications, Sept. 1995, pp. 135-149.

[42] H. Madeira, D. Costa, and M. Vieira, "On the emulation of software faults by software fault injection," Proceeding International Conference on Dependable Systems and Networks. DSN 2000, New York, NY, USA, 2000, pp. 417-426, doi: 10.1109/ICDSN.2000.857571.

[43] R. Natella, D. Cotroneo, J. A. Duraes and H. S. Madeira, "On Fault Representativeness of Software Fault Injection," in IEEE Transactions on Software Engineering, vol. 39, no. 1, pp. 80-96, Jan. 2013, doi: 10.1109/TSE.2011.124.

[44] J. Wei, A. Thomas, G. Li, and K. Pattabiraman, "Quantifying the Accuracy of High-Level Fault Injection Techniques for Hardware Faults," 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Atlanta, GA, 2014, pp. 375-382, doi: 10.1109/DSN.2014.2.

[45] S. Han, K. G. Shin, and H. A. Rosenberg, "DOCTOR: an integrated software fault injection environment for distributed real-time systems," Proceedings of 1995 IEEE International Computer Performance and Dependability Symposium, Erlangen, Germany, 1995, pp. 204-213, doi: 10.1109/IPDS.1995.395831.

[46] T. K. Tsai, R. K. Iyer and D. Jewitt, "An approach towards benchmarking of fault-tolerant commercial systems," Proceedings of Annual Symposium on Fault Tolerant Computing, Sendai, Japan, 1996, pp. 314-323, doi: 10.1109/FTCS.1996.534616.

[47] L. Palazzi, G. Li, B. Fang, and K. Pattabiraman, "A Tale of Two Injectors: End-to-End Comparison of IR-Level and Assembly-Level Fault Injection," 2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE), Berlin, Germany, 2019, pp. 151-162, doi: 10.1109/ISSRE.2019.00024.

[48] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, “Understanding the propagation of hard errors to software and implications for resilient system design,” in ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Seattle, WA, USA, pp. 265–276, 2008, doi: 10.1145/1346281.1346315.

[49] N. Nakka, G. P. Saggese, Z. Kalbarczyk, and R. K. Iyer, “An Architectural Framework for Detecting Process Hangs/Crashes,” in Dependable Computing (EDCC), Springer Berlin Heidelberg, 2005, pp. 103–121, doi: 10.1007/11408901_8.

[50] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. K. Iyer, "Dynamic Derivation of Application-Specific Error Detectors and their Implementation in Hardware," 2006 Sixth European Dependable Computing Conference, Coimbra, 2006, pp. 97-108, doi: 10.1109/EDCC.2006.9.