Silent Data Corruptions: Microarchitectural Perspectives

George Papadimitriou, *Member, IEEE,* and Dimitris Gizopoulos, *Fellow, IEEE*

Abstract—Today more than ever before, academia, manufacturers, and hyperscalers acknowledge the major challenge of silent data corruptions (SDCs) and aim on solutions to minimize its impact by avoiding, detecting, and mitigating SDCs. Recent studies on large scale datacenters conducted by Meta and Google report an unexpected rate of silent data corruption incidents that are attributed to modern microprocessor generations. Despite the acknowledged severity of the phenomenon, particularly at the datacenter scale, there is no in-depth analysis of the microarchitectural locations in a complex microprocessor that are more likely to generate an SDC at the program outputs. In this paper, we present a detailed analysis of the faulty behavior of many critical microarchitectural structures of a modern out-of-order microprocessor generating silent data corruptions. Our analysis unveils several observations, including: (i) the magnitude of silent data corruptions attributed to different hardware structures, (ii) the instruction-related parameters that are more likely to result in a silent data corruption, (iii) the extent to which the operating system affects the silent data corruption occurrences, and (iv) the byte positions of a word which are more likely to result in silent data corruptions. Collectively, such findings can assist decisions for hardware and software schemes for the reduction of the likelihood of silent data corruptions generation.

Index Terms—Silent data corruptions, faults, errors, microarchitecture, microprocessor, fault injection

1 Introduction

The extreme scaling of semiconductor devices and the complexity of integrated circuits lead to less reliable microprocessors operation due to the continuously increased error rates from transient faults [1]–[4]. Neutron and alpha particle impacts, process variances, marginalities, environmental factors, and low-voltage operation are the most common causes of such faults [5]–[7]. Such scenarios threaten the system functionality and the output correctness. Faults, such as transient or permanent faults, timing errors, design bugs, and manufacturing defects, occur inside a microprocessor chip and may cause silent errors in the program's output – known as Silent Data Corruption (SDC) – without any indication of the output degradation in system events or error logs. The data handled by the impacted devices may be silently corrupted by errors brought on by transitory defects. However, a single, hardware-induced fault can cascade into a massive problem in modern large-scale infrastructures. Recent studies on large datacenters conducted by Meta and Google [8], [9] indicate that ephemeral computational errors in CPUs, such as timing errors, design bugs, and manufacturing defects, generate SDCs at a much higher rate than the soft error-induced SDCs, while prior studies attribute silent data corruptions mainly to soft errors due to radiation and particle-strikes [1]–[4].

Although all studies in the literature stress that silent data corruptions are real and severe phenomena, especially in datacenter applications running at scale, interestingly, there is no report on the criticality of hardware (microarchitectural) structures in reaching the software layer and eventually generate silent data corruptions. Today more than ever, academia, manufactures, and hyperscalers acknowledge the major challenge of silent data corruptions, and try to propose solutions to measure and mitigate it. We anticipate that only holistic, cross-layer solutions can overcome this problem, based on the latest reports from large-scale datacenter infrastructures. However, it would be extremely difficult to deliver effective solutions to mitigate the problem without understanding:

1. the exact way that faults in all major hardware structures propagate to the software and eventually result in silent data corruptions, and
2. the scope of this challenging problem.

Understanding the faults behavior and cross-layer propagation statistics is an important step towards understanding the problem of silent data corruptions.

To reduce the effects of on-chip memory faults, error correcting codes (ECC) are deployed to detect and correct faults when they occur [10]. However, ECC methods require additional storage overheads and complexity, and cannot detect or correct all hardware-induced faults [11]. The most frequently used ECC methods can detect a small number of faults and correct even smaller number of them. For example, the most common ECC method (i.e., single error correction, double error detection – SECDED) can detect up to 2 flipped bits and correct 1 flipped bit per 64 bits and requires storing an additional 12.5% of ECC information [10] [11]. Diverse events can change the data stored in on-chip memory structures, or even permanently damage their bit-cells, and if left uncorrected, they can threaten the program integrity. This phenomenon is amplified in newer fabrication technologies, in which multiple-bit faults are more likely to occur in on-chip memory structures [2], [12]. Therefore, even if ECC can be beneficial for reducing the failure rates in some on-chip memory structures, the

• *G. Papadimitriou and D. Gizopoulos are with the Department of Informatics and Telecommunications, National and Kapodistrian University of Athens, Panepistimiopolis, Ilissia, Greece, GR-15784.*
E-mail: {georgepap | dgizop}@di.uoa.gr.

entire microprocessor does not have similar protection in all functional and memory blocks. It has been shown that even using common ECC methods, silent data corruptions are unavoidable, especially in large-scale datacenter infrastructures [8], [9], [13]. Therefore, it is essential to understand what kind of faults and in which hardware structures are more likely to propagate from the hardware all the way to the software and silently affect the program's execution. Through such an analysis, we can build effective solutions towards reliable modern systems, including new ECC mechanisms, or new software-based redundancy solutions that tolerate silent data corruptions.

In this paper, we contribute with an in-depth microarchitectural analysis for the causes of SDCs: the distribution of hardware corruptions that affect the software layers and eventually result in corrupted program outputs. We focus on hardware affected by transient faults, i.e., flipped hardware bits, an assumption that models any failure with transient behavior. It covers a broad variety of physical mechanisms: cosmic radiation, alpha particles from packaging, intentional low-voltage operation, circuit variability, manufacturing defects, and ephemeral errors due to escaped design bugs [5]–[7]. Using our simulation infrastructure in a similar approach than the one described in this paper, the same microarchitecture-level evaluation can be performed for other types of hardware faults including permanent faults (due to aging/wear-out), as well as hardware bugs.

For a complex RISC 64-bit out-of-order microprocessor (modeling Arm's Cortex-A72), we study the propagation of faults from the locations (hardware structures) they originate at all the way through the microarchitecture and the software levels to reach program output. For the first time, we deliver the following information at the microarchitectural detail: (i) the magnitude of silent data corruptions that are attributed individually to 11 major hardware structures of modern CPUs, (ii) the instruction-related parameters that are more likely to result in a silent data corruption (Section 2.3), (iii) the extent to which the operating system affects the silent data corruption occurrences, and (iv) the byte positions of a data or instruction word which are more likely to result in silent data corruptions. Our entire analysis is performed from a microarchitectural perspective (because the faults of the hardware are the causes that generate the SDCs) and provides insights across the layers of abstraction.

1. We extensively describe the critical problem of SDCs and summarize the limitations of current well-established methods that aim to harden the applications to tolerate silent data corruptions.
2. We augment the description of this problem by examining the cross-layer propagation of a hardware-induced fault and explain the sources of the problem from the microarchitectural point of view. We show which on-chip structures are more likely to result in silent data corruptions and explain the reasons.
3. We explore the impact of user vs. kernel instructions that result in silent data corruptions, separately for each hardware structure. Our analysis reveals that although the kernel instructions account for significantly lower number compared to the total executed ones, for some components they show extremely high probability to get affected by a hardware-induced fault.
4. We study the likelihood of different byte positions to generate silent data corruptions and present the different probabilities that each byte position inside an 8-byte word has, for the largest hardware structures, in terms of the total bits they can store. These structures are the data field of L1 instruction and data caches, the L2 cache, and the Physical Register File.

2 Understanding Silent Data Corruptions & Cross-Layer Fault Propagation

2.1 Silent Data Corruptions (SDCs)

Large-scale infrastructure services may be negatively affected by silent data corruption (SDC). Silent data corruption is a widespread problem that is now attributed more than in the past to microprocessor chips in addition to off-chip (main) memory, storage, and networking that have been for long considered main contributors to the problem [8], [9]. SDCs are not traceable at the hardware level since error reporting systems in microprocessors cannot keep record of such corruptions, and this is the reason that they are called silent. The data corruptions, however, spread throughout the system stack and show up as issues at the application level, or worse, in large-scale datacenters the issue may be distributed through several server locations. These errors can lead to data loss and may take months to correct them, because the detection of a silent data loss occurs very late [8], [9]. A silent data corruption happens when a microprocessor chip that is unintentionally affected (i.e., by soft errors, manufacturing defects, escaped design bugs, etc.) corrupts the data it processes. For instance, a CPU with a hardware fault or bug might compute data incorrectly (e.g., $2 \times 2 = 5$), or load/store an incorrect value, which is used for a subsequent computation. Unless the software systematically checks for such kinds of corruptions, there might not be any evidence of these computational problems.

Unlike other failures which are very visible, such as an application crash, SDCs cannot be observed and in most of the cases can get totally undetected. Software-level redundancy methods (or software-based fault-tolerant methods) can be implemented in applications for dealing with SDCs and preventing software-level failures [14], [15]. These methods are based on the redundancy concept and usually provide duplication or triplication of the application's resources. In such a way, the fault-tolerant application can eliminate a potential data corruption, since redundant copies can increase the probabilities for detecting and isolating a potentially malfunctioning hardware resource. Although software-based fault-tolerant methods can significantly reduce the potential SDCs and guarantee the correct execution, they have four major limitations:

1. The cost of redundancy in terms of performance and power efficiency is excessive and imposes large performance degradation and increased power consumption. This performance reduction and increased power consumption has direct impact on the end users. The stronger the redundancy approach is (e.g., triplication of the application's resources), the higher the performance degradation and the power consumption.
2. Since these methods aim to tolerate SDCs by providing computational and/or resources redundancy, the

application's code footprint is significantly increased. Increasing the code (and data) size to implement redundancy, the execution patterns of the fault-tolerant program are changed compared to the initial, unprotected version of a program. As a result, although the SDCs may be reduced, the probability of the occurrence of any other potential fault effect (i.e., a crash) may be increased. It has been recently shown that software redundancy methods may increase the vulnerability of the hardened application to crashes [16].

3. Software redundancy methods are primarily applied only to the application and not to the entire software stack, i.e., libraries and operating system (unless these are open source). Even if there have been some previous studies that aim to tolerate well-known open-source libraries (e.g., PyTorch [17]) and Linux kernel (e.g., FT-Linux [18]) for hardware-induced faults, the fault coverage locations and the performance degradation remain unsolved and severe issues. For example, FT-Linux [18] is a Linux-based operating system that transparently replicates race-free, multithreaded POSIX applications on different hardware partitions of a single machine. The slowdown of this method is measured to be up to 40% additional to the replication (which is attributed to $2\times$ more resources usage). Moreover, from both the development effort and performance degradation point of view, it would be impractical, or even unfeasible, to add redundancy to the full-software stack (i.e., applications, libraries, operating system).
4. Hardening the full software stack could be considered as a simplified solution for protecting the software-side from the occurrences of hardware-induced corruptions that result in SDCs. However, a recent study has shown that there can be a relative high number of hardware-induced faults that certainly result in SDCs in the output of the program, but these faults are not visible to the software or architecture layer [16] (see more details in subsection 2.2 and subsection 4.1). Therefore, even with a strong software-based protection through the full software stack, a significant portion of hardware faults that eventually lead to SDCs may remain undetected, not only from SECDED ECC at the hardware side (especially if the corruption is more than a single bit), but also from the software protection, and slip into the output without any indication.

For all these reasons, we believe that it is of paramount importance to deeply understand how the faults propagate from the hardware through the software and eventually to the output and which cross-layer aspects (microarchitectural components machine instructions, kernel operations, data bit locations) are more likely to participate heavily in the generation of SDCs. This is the first and most important step towards building truly resilient and holistic solutions to overcome the most challenging problem of SDCs.

2.2 Cross-Layer Fault Propagation

As a first step, it is important to model and understand the way that hardware faults propagate through the hardware layer, manifest to the software layer, and eventually affects the output of a program as SDCs. The abstraction layers of the computing stack include (among others) the microarchitecture, the architecture (or the ISA), and the software, which consists of the operating system, the libraries, and the application itself. As hardware-induced faults we mean any corruption that can occur at a microarchitectural structure. Consequently, a microarchitecture-level corruption may, or may not, be architecturally visible (i.e., the fault becomes visible to the ISA, and thus, to the application). If the corruption gets architecturally visible (i.e., it touches the software), it may or may not become visible to the output of the program. This is the abstract way of the fault propagation across different abstraction layers. However, this propagation path consists of several aspects, that need to be considered and completely understood:

- The propagation of a hardware-induced fault (i.e., a fault that corrupts the microarchitectural layer) strongly depends on (i) the microarchitecture, and (ii) the executed program. A fault occurrence is architecturally visible, if and only if, it does not get masked at the microarchitecture layer (i.e., hardware masking) due to a microarchitectural operation (which is, of course, agnostic to the ISA layer). Hardware masking can occur in three different cases: (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 (read), (iii) the fault affects a mis-speculated instruction (i.e., the fault is discarded due to a pipeline flush). In all three cases the hardware fault is Benign because it exists, but it never appears at the software layer (see Figure 1 the "Benign" case in the top). In any other case, a fault in a hardware structure or functional block will eventually be architecturally visible (i.e., cases "Masked" and "SDC" in Figure 1 represent two different cases of architecturally visible faults; the SDC case means that the fault propagates to the output while the Masked case means that the fault is software masked and output is not affected). Consequently, given a certain workload, different microarchitectures can primarily affect the population (i.e., the absolute number) of the Benign faults. The reason is that, for a given workload, different microarchitectures can only affect the hardware masking [19]. 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). For a given workload, M1 will lead to more Benign faults due to larger number of mispredictions, and thus, M1 and M2 have different hardware masking levels.
- On the other hand, an architecturally visible fault may

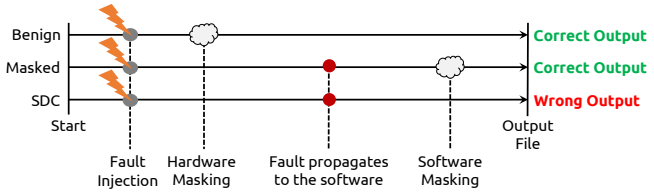

Fig. 1. Fault propagation from the injection of fault at any microarchitectural structure all the way to the final output of the program.

affect the program's output, if and only if, it does not get masked by the software (i.e., case "SDC" in Figure 1 represents an architecturally visible fault which does not get masked at the software layer and eventually affects the output of the program). Software/Logical masking 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, for example, 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). As another example, assume that an architecturally visible corruption in a register's content, is not used by another program's operation (the register content is dead). In such a case, even if the corruption is architecturally visible, the program's execution and the output will be correct due to software masking effect. On the other hand, in any other case, an architecturally visible corruption will affect either the program's execution (e.g., application crash) or the output (i.e., the case "SDC" in Figure 1).

- However, as we briefly described in section 2.1, there is also another important portion of hardware faults, which although initially considered Benign, they will certainly corrupt the output (i.e., SDC), without being architecturally visible. This portion of faults was initially defined in [16] 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). Such kind of faults determine 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. Those faults (in a cache level only) are impossible to be identified by any protection method that aims to detect SDCs, because 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). We explore this scenario in detail in subsection 4.1.

2.3 Classification of Corruptions at Software Layer

Since this study focuses on the propagation of hardware-induced corruptions all the way to the output, it is necessary to explore the anatomy of fault propagation by distinguishing the final effects of a fault (i.e., the effect to the output) to the fault effects at the time when they propagate from the hardware to the software. Thus, we need to rely on a solid observation point, that can provide this information between the hardware and the software interface. From the microarchitecture point of view, if the fault occurrence passes the commit stage of an out-of-order (OoO) microprocessor, the fault is considered architecturally visible (i.e., it passes to the software). Hence, the observation point is the point where speculated instructions get committed. The detailed methodology, we follow during simulations to capture this, is shown in Figure 2.

Each instruction (for any ISA) is associated to the following parameters upon its commit (i.e., retirement): (i) the cycle it commits, (ii) the Program Counter (PC), (iii) the Opcode, (iv) the register operands and/or an immediate field, and (v) the register contents. The instruction format can be shown at the bottom of Figure 2. Any corruption at the microarchitecture level can be considered architecturally visible, if that corruption affects any of these instruction-related parameters. For example, the third committed instruction at the bottom of Figure 2 is corrupted in its register contents and shown with red color. To this end, we keep track of the running trace of each committed instruction, and if any of the instruction-related parameters is different from the fault-free executions, we record it as a corruption (see next section for the experimental infrastructure we employ for the implementation of the method). This means that the fault, in any microarchitectural structure, becomes architecturally visible. Upon the corruption in the commit stage, we classify the kind of corruption (similarly to [20]) to one of the following groups:

(a) **Execution Time Error:** The instruction, the operand fields, the immediate value, and the contents of register are correct, but the instruction was committed in wrong cycle compared to the fault-free execution.
(b) **Instruction Flow Change:** A different instruction is executed compared to the original flow due to an incorrect instruction fetching (PC corruption).
(c) **Instruction Replacement:** A different instruction is executed compared to the original program flow due to a corrupted Opcode.
(d) **Operand Forced Switch:** One or more instruction operand fields are corrupted, or they are unknown to the ISA. This includes register operands or immediate values of the instruction format.
(e) **Data Corruption:** The correct resource is used, but the content (i.e., the data) of the resource (register or memory word) is corrupted.

The above five groups provide any manifestation of a hardware fault at the software layer. Any corruption at the microarchitecture-level (in any hardware structure) may affect one and only one instruction-related parameter, and thus, it will be categorized to one of these five distinct groups. However, depending on the corruption, the committed instruction may be different in more than one instruction-related parameter. Assume that an executed instruction has committed in the correct cycle, with a correct PC, but due to the fault occurrence, the committed instruction is different from the fault-free case. Apparently, the destination register contents will also be wrong, but this is due to the execution of wrong instruction. In that case, this

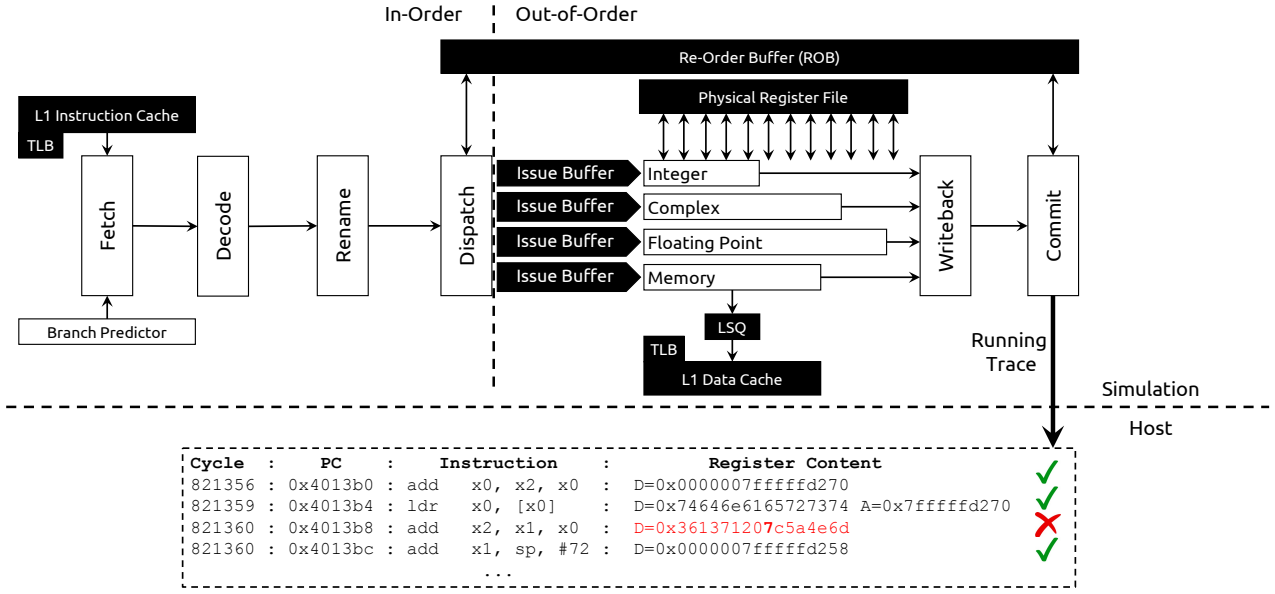

Fig. 2. Simulated microarchitecture model. The committed instructions are compared to the fault-free ones to recognize if there is a mismatch during the commit stage. If so, the corrupted instruction is marked and is categorized depending on the instruction-related parameter.

corruption in the commit stage will be categorized as an Instruction Replacement instead of Data Corruption, since the primary reason of corruption is the wrong opcode (i.e., the hardware-induced fault affected the opcode). Therefore, any corruption of the architectural state can be categorized to one and only one group and uniquely determines the kind of corruption.

3 Experimental Setup

3.1 Infrastructure

Such a study should ideally be based on a real system or a low-level simulator (e.g., RTL). Although low-level simulators may provide accurate fault effects, their simulation throughput is extremely low (3 to 5 orders of magnitude [21]) and cannot model long-running workloads with OS activity. Therefore, we are based on microarchitecture-level simulation using gem5 [22]. Gem5 allows deterministic end-to-end execution of large workloads on top of an operating system, which is impossible at lower levels. But even if an RTL model of a microprocessor was available and full system fault injection on it was possible (our study can be applied at this abstraction layer), it would only marginally augment our analysis [21] with the vulnerability of the combinational logic, which has very low raw failure rates compared to storage elements on which we focus [23]. For this reason, we present our results harnessing the high throughput of microarchitectural modeling in performance simulators. To this end, we employ GeFIN [20], the state-of-the-art microarchitecture-level fault injection framework built on top of the gem5.

GeFIN consists of a modified gem5 version that allows fault injection along with instrumentation for running and controlling simulation campaigns on full-system setup [12]. In this study, we use GeFIN to inject transient faults (i.e., the fault model used in this study). GeFIN injects random faults in different bits of the same entry of a hardware structure and in different entries of a structure, for the desired hardware structure, following the uniform distribution as defined in [24]. The execution running trace of GeFIN (see Figure 2) collects information from the entire pipeline, stores several instruction-related parameters, and can distinguish between architectural and physical maps, among other features. These features are necessary for categorizing the fault occurrences into groups (as discussed in Section 2.3). Of course, in a real microprocessor, there are no data stored in commit stage. Since we are based on a simulation environment (on which we can inject faults), we can store and have access to any simulated resource. GeFIN is generally well-instrumented and provides features that can be found only in such a simulated environment.

3.2 Hardware Structures and Benchmarks

For this study, we employ a CPU from the 64-bit Armv8 ISA, modeling an out-of-order microarchitecture, which is very similar to the Arm Cortex-A72. For some experiments, we also present results for another ISA, the Armv7, modeling a Cortex-A15-like microprocessor in order to show that our observations still exist in other ISAs. For a comprehensive analysis, our evaluations target 11 important hardware structures: L1 data and instruction caches (tags and data fields), L2 cache (data field), the Physical Register File, the Load Queue (LQ), the Store Queue (SQ), the Reorder Buffer (ROB), and the Instruction and Data TLBs (Translation Lookaside Buffers). The most important hardware parameters for the considered microprocessor model are shown in Table 1. These hardware structures occupy most of the chip's area and are, therefore, the largest contributors to the vulnerability of the entire chip. The only SRAM structures, which have not been targeted in this study and are available in the gem5, are the structures of prefetchers, branch predictor units, and branch target buffers. The reason is that faults in these structures cannot result in any kind of corruption of the architectural state, let alone produce SDCs. Further, the floating-point physical register file is not considered because our benchmarks do not contain any FP operations. We employ a diverse set of 10 workloads from the MiBench

TABLE 1
Basic Simulated Hardware Architectures Parameters for the Considered Microprocessor Models.

ISA	Armv7	Armv8
Pipeline	Out-of-Order / 15 stages	
L1 Data Cache	32 KB (2-way)	
L1 Instruction Cache	32 KB (2-way)	48 KB (3-way)
L2 Cache	1 MB (8-way)	2 MB (16-way)
Physical Register File	128 registers	192 registers
Instruction TLB	32 entries (fully associative)	
Data TLB	32 entries (fully associative)	
Issue Queue	32 entries x 32 bit	64 entries x 64 bit
Load / Store Queue	16 entries x 32 bit	16 entries x 64 bit
Reorder Buffer	40 entries	128 entries
Fetch/Execute/WB width	3 / 6 / 8	

suite [25] using the largest possible input datasets for all benchmarks. The execution times of the benchmarks range from 100 million cycles to 1.4 billion cycles. Unlike some 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 [19], [26]–[28], in this study, we consider the end-to-end execution of benchmarks with significantly higher number of cycles (up to 1.4 billion).

The MiBench suite is very commonly used in reliability studies [29]–[35] as it combines realistic benchmarks with reasonable execution (thus simulation) time and facilitates complete end-to-end executions for the thousands of fault injections required in such an analysis. For each of the eleven on-chip hardware structures, 2,000 single-bit faults (thus 2,000 fault injection runs) were randomly generated following the uniform distribution as defined in [24] resulting in 220,000 faults for all ten benchmarks (22,000 for each benchmark; in other words 22,000 simulation runs for each benchmark) and the eleven different hardware components. We follow the widely adopted formulation of [24] for the statistical fault sampling calculations with 2.88% error margin and 99% confidence level.

3.3 Final Execution Fault Effect Identification

Any fault injection campaign, regardless of the abstraction layer, assumes that the origin of faults may influence the output of the program. Typically, the final fault effects classification is performed based on the following effects:

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 from the fault-free simulation, without any observable indication.

Crash: A simulation that neither reached the end of the workload nor finished within a certain amount of time, because it was disturbed by a catastrophic event. As a result, no program output was produced. The crash may refer to a process crash, a system crash (kernel panic), deadlock or livelock situations.

However, since this study focus on the SDC effects, we limited our observations into this fault effect class.

3.4 Evaluation Flow

For the needs of this study, our experiments need to be performed in two distinct phases to evaluate the propagation of faults. The first phase considers those faults that eventually reach the software layer, whereas the second phase evaluates the effect of the architecturally visible faults to the output of the program. The former can be evaluated through the HVF (Hardware Vulnerability Factor [36]) assessment, while the latter through the AVF (Architectural Vulnerability Factor [37]) assessment. The HVF assessment and the AVF assessment, providing the microarchitecture-dependent vulnerability and the cross-layer (full stack) vulnerability, respectively. The microarchitecture-dependent evaluation (i.e., HVF) targets on the effects of hardware faults until they "touch" the software layer (i.e., the fault becomes architecturally visible). For the HVF analysis, we consider as Benign faults, those faults that eventually get masked by a microarchitectural operation (as described in Section 2.2), and thus, the fault occurrence never reaches 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) group presented in Section 2.3, depending on the type of corruption.

From the AVF point of view, a Corruption (i.e., an architecturally visible fault) may or may not affect the program's execution, and thus, the output. 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 (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 and subsequently affects the program's operation, it is classified either as an SDC or as a Crash. However, for this study, we consider only those faults that silently affect the execution of the program, and in the following sections we provide the correlation of architecturally visible faults to the SDC fault effect.

4 Cross-Layer Analysis of SDCs

4.1 Correlation of SDCs to the Hardware Structures

In this subsection, we present the correlation of silent data corruptions to each hardware structure considered in this study. Before we discuss the distribution of the architecturally-visible faults (see Section 4.2), it is essential to evaluate the percentage of faults in each hardware structure that produces SDCs. In such a way, we can gain a clearer view on the susceptibility of each individual structure to faults that eventually lead to SDCs. Figure 3 shows the susceptibility of each structure to faults that eventually lead to silent data corruptions for faults that did not get masked at the hardware level (i.e., non-Benign faults), for both Armv8 and Armv7 ISAs to present a comprehensive comparison between different architectures. For completeness, Figure 4 shows the remaining percentage, which is covered by Masked and Crash fault effect for Armv8. The percentages at the top of the bars show the sum of Masked and Crash percentage for each structure.

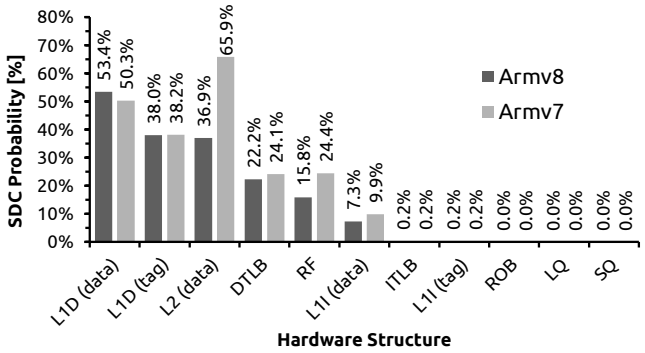

Fig. 3. The percentage of hardware corruptions that reach the software layer (i.e., non-Benign) that result in SDC for both Armv7 and Armv8.

In Figure 3, the first and major observation is that the Reorder Buffer (ROB), the Load Queue (LQ) and the Store Queue (SQ) have zero probability to contain hardware faults that can generate silent data corruptions. The reason is that any fault occurrence in these hardware structures cannot be architecturally visible, primarily due to dependence graph checks failures before the commit stage. Hardware structures, such as ROB, LQ, and SQ, which are located deep in the microprocessor's pipeline, account for the correct instruction ordering when instructions are ready to commit, and their queue entries consist of the Program Counter and the architectural or physical register specifiers. Any corruption in those parameters can result in dependence graph check failures before the commit stage, and thus, to a visible crash. Assume, for example, that a fault corrupts the physical register specifier of a certain ROB entry. The correct register specifier was, for example, `r9`, whereas the corrupted register specifier is `r13`. If the physical register `r13` is an available specifier, currently located in the free list, a dependence failure occurs before commit. Therefore, the related instruction will not commit, and the microprocessor results in a crash. It is highly unlikely due to the operation of these structures and their role in the microarchitecture to result in an SDC due to a fault in them.

On the other hand, the data, and the tag field of L1 data cache, and the L2 cache are the most susceptible on-chip hardware structures to faults that eventually result in silent data corruptions. The probability of these hardware structures to result in a silent data corruption due to a fault is 53.4%, 38.0%, and 36.9%, respectively, as shown in Figure 3, for Armv8. Interestingly, the same observations hold true for the different ISA (i.e., the Armv7), in which we can see that follows the same trends.

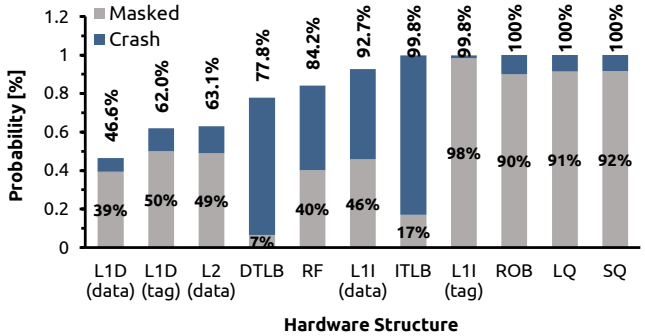

Fig. 4. The percentage of hardware corruptions that reach the software layer (i.e., non-Benign) that result in Crash and Masked for Armv8.

L1 data cache stores data values that are usually used for result or address calculations. Therefore, if a fault occurs in the data field of L1 data cache, it is very likely to corrupt a computation (i.e., the corrupted entry was read) or directly the output (i.e., the corrupted entry is stored without being read). Moreover, L2 cache also stores data values that are usually used for result or address calculations, as well as instructions and address translations in addition to data values. Therefore, the probability that faults in this structure can generate an SDC is, naturally, less than the corresponding probability of the L1 data cache. The reason is that it is more likely for a corrupted instruction or an address translation to result in a Crash rather than in a silent data corruption. However, as we can see in Figure 3, the L2 cache provides higher SDC vulnerability for Armv7, in contrast to Armv8. The reason is that Armv7 L2 cache is half the size of Armv8 ISA, so the probability is getting higher. This cannot affect the initial observations of SDC probabilities.

At this point, it is important to discuss in more detail the case in which a fault directly affects the output (i.e., the corrupted entry is stored without being read), because it is the most interesting one in terms of reliability evaluation. This can happen for both L1 and L2 caches (see Section 2.2). Figure 5 presents the probability of hardware-induced faults and for each program to result in silent data corruption, directly to the output, without be visible to the software layer. It is clear in this graph, that the number of this kind of faults is different for different programs. Moreover, such faults are initially categorized as *Benign Faults* during the HVF analysis, since they do not affect the architectural state [19]. However, it is necessary to correlate the results of HVF analysis to the output file in order to unveil such corruptions. Interestingly, our analysis unveils a strong correlation of the number of this kind of faults to the number of Benign Faults and the output size of each program. Specifically, when the program's output is relatively small (e.g., *bitcount* has total output size less than 1 KB), the probability of such a fault to affect the program's output is zero (as it is clearly shown in Figure 5 for every hardware structure). On the other hand, the probability of such a fault to affect the program's output is getting higher as the output size gets high (e.g., *blowfish* has output data greater than 3 MB). The reason is that, for programs with relatively small output size, a cache level stores only a small part of the total output at every given moment, so it is extremely unlikely for a fault to hit those (few) words that are stored in cache just before they are written to the output file. In contrast to program's with large output size, for which it is more likely for a fault to hit the

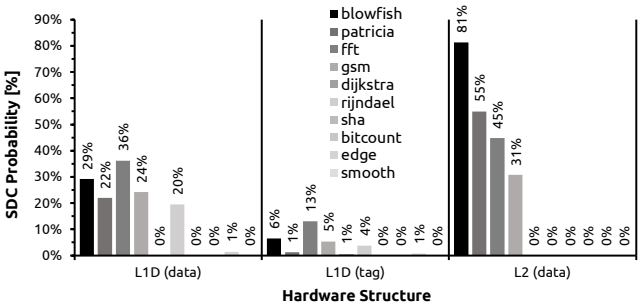

Fig. 5. The probability of hardware-induced faults that result in silent data corruptions which do not pass through the program trace.

part of the output data reside in a cache level before they are written to the output.

The data TLB and the Physical Register File (RF) are also susceptible to faults that result in silent data corruptions, as shown in Figure 3. Their SDC probabilities are account for 22.2% and 15.8%, respectively, of the total vulnerability of each structure. The reason that these structures are also susceptible to faults that lead to silent data corruptions, but less than L1 data cache and L2 cache, is because these structures are also attributed to interplay with data values, or memory addresses that belong to data in memory. However, the reason that the Physical Register File and the data TLB are less susceptible to faults that result in silent data corruptions than L1 data cache and L2 cache, is because the former handles also memory addresses apart from data values, and the latter has high probability to result in Masked or Crash.

Assume for example that a fault affects the physical address part of an address translation that is stored in the data TLB. Since this physical address aims to a data frame in memory (not an instruction frame), it is likely due to the wrong physical address to fetch a different word because, due to the fault, the physical mapping is wrong. However, it is also likely for the corrupted physical address to point to an unmapped memory location, or an unprivileged one. Therefore, due to the nature of this structure and its role in the microarchitecture, the silent data corruption probability occurs (and it is attributed to 22.2%), but at the same time the probabilities for masking or crash are reasonably higher.

On the other hand, a fault in the instruction TLB or the tag field of L1 instruction cache has nearly zero probability to result in an SDC (the probability is 0.2%, as shown in Figure 3). The reason is that both structures account for memory addresses of instruction-related memory frames, and thus it is more likely for these structures to generate a crash due to a fault, rather than an SDC.

Interestingly, the data field of the L1 instruction cache shows a non-negligible susceptibility to silent data corruptions. L1 instruction cache stores instructions, which primarily consist of an opcode, register operands, and immediate values. A corruption in the opcode field will most likely result in a crash, and in very rare cases in an SDC. Specifically, the most common scenario that a corruption in the opcode could result in an SDC, is if the new opcode is ISA-defined and the new executed instruction correlates somehow its operation to the fault-free case. For example, if the fault-free instruction is an add instruction and due to a fault becomes an or instruction, it is likely to result in a silent data corruption. However, this case is extremely rare. On the other hand, if the corruption occurs in a register operand or the immediate field of an instruction, the probability of an SDC is higher compared to a corruption in the opcode. The reason is that it is more likely for a corruption in the operands or the immediate field to let the execution normally proceed, but since the value which is used is wrong, the execution could result in a silent data corruption. Of course, this is not the most likely case, however, as we discuss in the next section, it is likely to result in an SDC.

4.2 Exploring Architecturally-Visible Faults

As we discussed previously in subsection 2.3, any hardware fault that affects the software (i.e., the fault is architecturally visible), it may corrupt one and only one instruction-related parameter: (i) the committed cycle, (ii) the Program Counter (PC), (iii) the opcode, (iv) register operands and/or an immediate field, and (v) the register contents. In this section, we present the anatomy of the hardware-induced faults, by exploring the propagation of faults from the hardware layer to the software layer. Therefore, our experimental analysis consists of a detailed HVF analysis, which can indicate which faults affect the software layer.

During the HVF analysis, we categorized the architecturally visible faults into the five distinct groups, as described in subsection 2.3, i.e., (a) Execution Time Error (i.e., wrong committed cycle), (b) Instruction Flow Change (i.e., program counter change), (c) Instruction replacement (i.e., opcode corruption), (d) Operand Forced Switch (i.e., register operand or immediate field corruption), and (e) Data Corruption (i.e., register content corruption). Once we categorize the faults into these five distinct groups, we continue running complete AVF experiments, to obtain their final fault effect. In such a way, we can correlate each hardware-induced fault in any on-chip hardware structure, to its eventual final effect. Of course, since our study focuses only on silent data corruptions, we consider only the probability of these faults to result in a silent data corruption, and we study these probabilities, by breaking down the architecturally visible faults into categories.

Figure 6 shows the silent data corruption probabilities of all architecturally visible faults, for every hardware structure, separately for each of the five groups of fault manifestations at the software layer. It is clearly shown by these graphs that faults that belong to the Data Corruption group (i.e., faults that corrupt the register content) provide the higher SDC rate among any other group. Note that faults which directly corrupt the output without being software visible are considered as Data Corruptions as well (the reason is that this group of faults hit a part of the program's output which is exposed in any cache level, but only the cache arrays that store data, as explained in section 2.2, so the corruption can be only applied to data values.). Of course, it can be intuitively seemed reasonable that faults in the Data Corruption group provide the highest probability to result in silent data corruptions, since it is very likely for a fault that corrupts a data value to result in a silent data corruption. Apparently, most (if not all) of the software-based fault-tolerant methods exist in the literature, are primarily based on this intuition. Software-based fault-tolerant methods are primarily based on the redundant operations of the application code and a comparison between the actual and the redundant code to ensure that they both match. In case of a difference, the detection of a potential data corruption occurs, and the necessary actions are employed depending on the method. However, a major insight of this study is that, as we can see in Figure 6, all SDCs are not due to value corruptions (i.e., Data Corruption group), and thus, it is very difficult for some (if not all) software-based fault-tolerant techniques to detect such kinds of corruptions.

Assume for example the faults that belong to the Instruction Flow Change group. As we can see in the corresponding graph (the top-middle graph of Figure 6), nearly 10% of the faults in the tag field of the L1 instruction cache may result in silent data corruptions. Hardware-induced faults

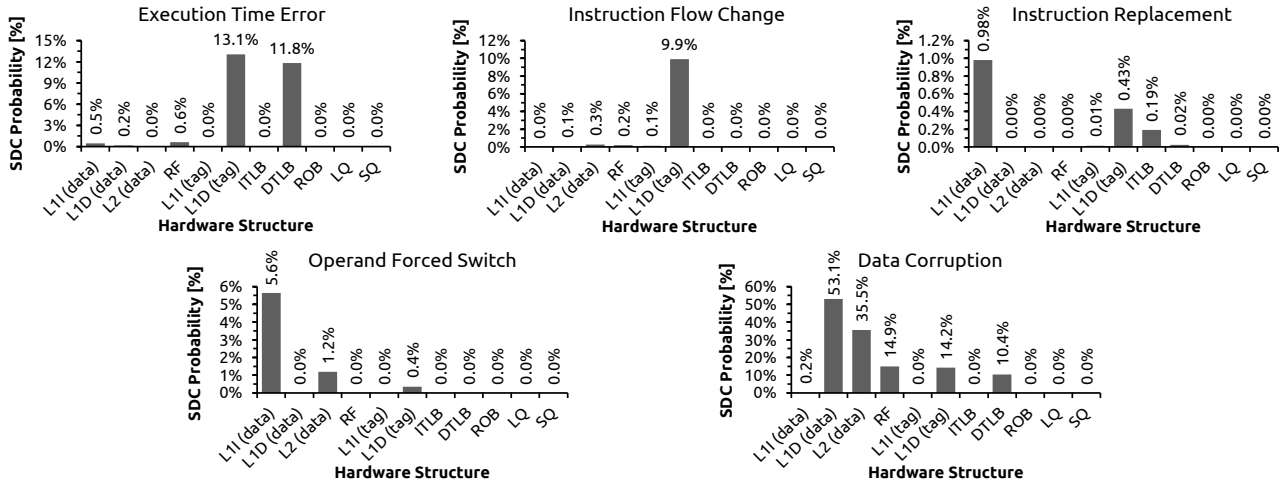

Fig. 6. SDC probabilities of non-Benign faults (i.e., faults visible to the software), for all five corruption groups. Each graph shows only those faults that correspond to one group, for every hardware structure (x-axis) (notice the different scale of the y-axis across the graphs).

in the tag field of the L1 instruction cache are likely to affect the instruction flow of the executed program, since any corruption in it will affect the tag comparisons of the next-executed instructions. In such a scenario, it is likely for the execution to result in a false tag match, and a different instruction will be executed. Of course, the instruction will be correct, with the correct register operands, but it is not the one that it was supposed to be executed at this cycle. Such a case is very difficult to be detected by a software-based redundancy technique, since there is a part of the code that may be not executed at all (and thus neither the redundant code that depends on it). Another example is the faults that belong to the Execution Time Error. Faults in this group affect the execution flow of the program, i.e., the instruction will be committed in the correct order, but in the wrong cycle. As we can see in the corresponding graph (the top-left graph of Figure 6), the address-based memory structures, which are strongly related to the data manipulation, may result in silent data corruptions. Specifically, the tag field of the L1 data cache and the data TLB provide the highest probability to result in silent data corruptions (i.e., 13.1% and 11.8%, respectively), compared to the data field of the L1 instruction and data caches, and the physical register file (0.5%, 0.2%, and 0.6%, respectively).

Note that as we discussed in subsection 4.1, Reorder Buffer (ROB), the Load Queue (LQ) and the Store Queue (SQ) show zero probability to silent data corruptions, primarily due to dependence graph checks failures before the commit stage. Therefore, the probability of faults that result in SDCs for these structures for any group is zero.

4.3 Impact of Multi-bit Faults on SDCs

It has been observed through physical experiments of accelerated beam testing [2] that on-chip storage arrays can suffer multiple-bit flips in adjacent areas. Therefore, multi-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., Data Corruption) and the opcode of that instruction at the same time (i.e., Instruction Replacement). Therefore, the final estimation results of SDC probabilities in the presence of multi-bit faults could be, eventually, higher than the single-bit faults, because the probability of a corruption is increased. This is in line with previous studies that consider multiple-bit faults [2], [12], [19].

5 Instruction-Level Analysis of SDCs

Another major limitation of software-based fault tolerance methods, which focus on protecting an application against silent data corruptions, is that these methods are primarily applied only to the application code and not to the entire software stack, i.e., application, libraries, and operating system. This means that from the cross-layer resiliency point of view, hardware-induced faults that may corrupt the kernel or libraries instructions, are practically unprotected and will potentially result in silent data corruptions (or any other fault effect depending on the corruption). However, in a full-system setup (i.e., in real-life servers' execution), system calls and library calls occur hundreds or thousands of times during the normal application's execution, and thus, kernel and libraries code are both executed along with the application's code. To this end, it is important to study the frequency of corruptions during the execution of kernel and libraries code and assess the impact of them on the final estimated silent data corruptions of executed programs.

Since this work aims to study the SDCs from the microarchitectural perspective, we distinguish and present our instruction-level analysis between user and kernel instructions. The reason is that from the microarchitecture-level point of view, it is not possible to know which instruction is related to the program and which instruction is related to a library, since both are considered in user space. Thus, our instruction-related study focuses on the differences between user and kernel instructions.

5.1 User vs. Kernel Instructions Breakdown

As a first quantitative evaluation, we present the breakdown of user and kernel instructions for all programs used in this study. As shown in Figure 7, the percentage of kernel instructions of each program is different and account for the minority of the total executed instructions. Specifically,

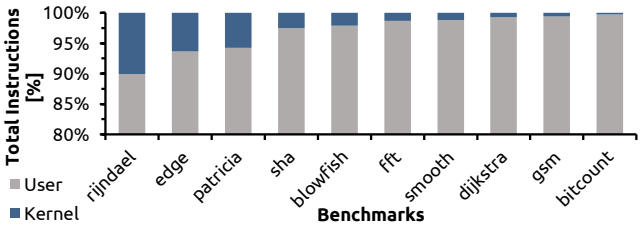

Fig. 7. Breakdown of user vs. kernel instructions for all programs considered in this study.

the benchmark that provides the most kernel instructions during the entire program's execution is *rijndael*. For *rijndael*, 10.1% of the total executed instructions are related to kernel instructions. Edge and *patricia* benchmarks constitute nearly half of kernel instructions compared to *rijndael*, which account for 6.3% and 5.8%, respectively. Moreover, as we can see in Figure 7, the remaining benchmarks have less than 3% of kernel instructions. Specifically, *bitcount* is the benchmark with the least kernel instructions, which are account for only 0.2% of the total executed instructions of this benchmark. Overall, the total kernel instructions executed for any program constitutes at most the 10% of the total executed instructions, while the user instructions constitute the remaining 90% (or more) of the total executed instructions.

According to this breakdown, we can naively assume that the impact of kernel instructions on the SDCs of each program could be extremely low compared to the impact of user instructions on the silent data corruptions of each program. However, as we discuss in the next subsections, the impact of kernel instructions on silent data corruptions is not negligible and account for nearly half of the silent data corruptions in some cases.

5.2 Impact of User vs. Kernel Instructions on SDCs

In this subsection, we present the impact of user vs. kernel instructions that result in silent data corruptions, separately for each hardware structure. Figure 8 shows the breakdown of user versus kernel instructions for the L1 data cache (data field) structure, since it is the most vulnerable structure to SDCs among the other structures. It is clearly shown in this graph that each benchmark provides different percentage of SDC probability between the executed user and kernel instructions. More specifically, in Figure 9 we can see the breakdown of user vs. kernel instructions for every hardware structure that provide SDCs, consolidated for all benchmarks used in this study. The percentages at the top of each graph show the total probability of silent data corruptions for each hardware structure. As we

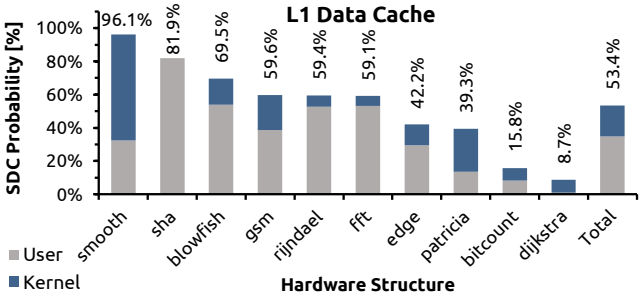

Fig. 8. Impact of user vs. kernel instructions on the silent data corruptions for the L1 data cache (data field).

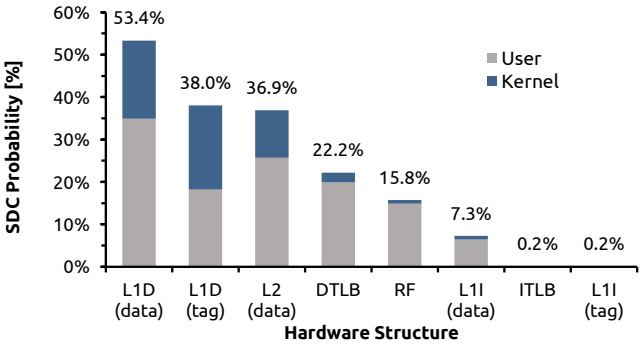

Fig. 9. Impact of user vs. kernel instructions on the silent data corruptions for each hardware structure.

can see in this graph, hardware structures that primarily account for data manipulation (i.e., L1 data cache and L2 cache), not only provide the highest probability of silent data corruptions, but also, the kernel instructions contribute significantly more to silent data corruptions compared to any other hardware structure. Specifically, for the data field of L1 data cache, we can see that 18.4% out of 53.4% of the total silent data corruptions belong to kernel instructions. For the tag field of the L1 data cache, 19.8% out of the 38% of the total silent data corruptions belong to kernel instructions. For the L2 cache, 11.2% out of the 36.9% of the total silent data corruptions belong to kernel instructions.

On the other hand, the impact of kernel instructions on silent data corruptions for the other hardware structures is lower compared to user instructions, but not negligible. These hardware structures (i.e., the instructions and data TLBs, the Physical Register File, and the tag and data fields of L1 instruction cache) are primarily addressed-based structures and account for instructions. Therefore, they fundamentally provide lower SDC probability than the structures that primarily account for data manipulation. Specifically, for the data TLB, 2.3% out of the 22.2% of the total SDCs belong to kernel instructions. For the Physical Register File (RF), 0.9% out of the 15.8% of the total SDCs belong to kernel instructions. For the data field of L1 instruction cache, 0.8% out of the 7.3% of the total silent data corruptions belong to kernel instructions. For the instruction TLB, there is no kernel instruction (i.e., 0%) that provides SDCs, while for the tag filed of the L1 instruction cache more than half of the total instructions that result in SDC belong to kernel instructions (i.e., 0.14% out of 0.2% belong to kernel instructions).

5.3 Kernel Instructions Contribution to the SDCs

Apart from the impact of user vs. kernel instructions on the silent data corruptions for each structure, it is also

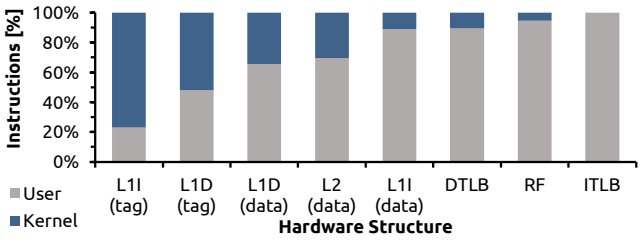

Fig. 10. The contribution of the kernel instructions to the total silent data corruption rate of each hardware structure.

essential to study the relative contribution of the kernel instructions to the total silent data corruption rate of each hardware structure. To this end, in this subsection, and more specifically in Figure 8, we analyze the contribution of kernel instruction to the total SDC rate. Specifically, in Figure 10 we can see that a hardware-induced fault in the tag field of the L1 instruction and data caches can severely affect the kernel instructions compared to user instructions. Specifically, for the L1 instruction cache (the tag field) the contribution of affected kernel instructions that result in silent data corruptions is nearly 77% of the total instructions that result in silent data corruptions, while for the L1 data cache (the tag field) the corresponding contribution is more than 50%. Moreover, for the data field of L1 data cache and for L2 cache, the contribution of kernel instructions to the silent data corruptions compared to the total instructions that result in silent data corruption is also non-negligible and account for more than 30%. It is clear from this graph that in these four hardware structures, there is significantly higher probability for the kernel instructions to be affected by a hardware-induced fault compared to user instructions. This insight is exceptionally interesting, if we consider that the total kernel executed instructions for all programs that are used in this study, account for less than 10% of the total executed instructions (both user and kernel instructions).

6 Byte-Level Analysis of SDCs

Although the analysis of the impact of user vs. kernel instructions on the silent data corruptions is conceptually essential for understanding the main sources of faults propagation, it is also important to study the impact of byte position on the SDCs. To this end, in this section we present the different probabilities that each byte position inside an 8-byte word has, for the largest size hardware structures, in terms of the total bytes they can store. These structures are the data field of L1 instruction and data caches, the L2 cache, and the Physical Register File.

Figure 11 presents the probability of different byte positions in an 8-byte word of the four different hardware structures to result in silent data corruptions under the occurrence of a single-bit flip. The x-axis shows the byte positions, i.e., B0 label in the x-axis represents the least significant byte of the word, whereas B7 label in the x-axis represents the most significant byte of the word. Note that exceptionally for the L1 instruction cache, the word size is 4 bytes and not 8 bytes as it is in all other structures. The reason is that in Armv8 ISA, which is the architecture we study in this paper, the instruction length is 4 bytes.

As we can see in Figure 11 for the L1 instruction cache (i.e., the left-most graph of Figure 11), the probability of any byte position is nearly the same for every byte. However, this is not the case for the other three structures, which are primarily account for data and address values. Specifically, we can see that in L1 data cache, the probability of a byte to result in an SDC under the occurrence of a single-bit flip becomes lower as we move towards the most significant bytes. The same observation holds also for the L2 cache, in which, however, there are some exceptions regarding the B6 and B2, which present a relatively larger probability to result in silent data corruptions (i.e., nearly 15%). On the other hand, the Physical Register File shows an extremely unbalanced behavior (the rightmost graph of Figure 9). As we can see in the rightmost graph of Figure 11, B2 and B3 provide the highest probability to result in silent data corruption under the occurrence of a single-bit flip, while this probability is getting lower as we move from B4 towards the most significant byte.

7 Related Work

Several works have been presented in the literature regarding silent data corruptions caused by transient faults [38]–[41]. Although these studies can be conceptually useful, the distribution of hardware-induced faults that eventually affect the software, and thus, may silently corrupt the output, is completely different to what the software-level studies consider [16]. More specifically, Fang *et al.* in [38] presented a full-software stack study in respect to SDCs. Guan *et al.* in [39] present an empirical study on three sorting algorithms to determine the SDC vulnerability of individual portions of an algorithm. Li *et al.* in [40] propose a visualization tool, named SpotSDC, to facilitate the analysis of a program's resilience to silent data corruptions. Xu *et al.* in [41] propose CriticalFault, which is a vulnerability-driven framework that instructs the fault injection locations towards likely non-derated faults. However, these studies focus only on the software or ISA layer, and they primarily evaluate the effects of applications to the SDCs, without considering the microarchitecture. In this study, we consider the way that faults, whose origin is a hardware (microarchitectural) structure, propagate from the hardware to the software and eventually result in silent data corruptions. To our knowledge, this is the first study that shows the anatomy of faults at the microarchitecture-level all the way to the software-level.

Hari et al. in [42] present low-cost program-level detectors for reducing SDCs. This study focuses on identifying and understanding the program portions, which may cause SDCs, and reducing them in a cost-effective manner. This study is also software-oriented and assumes that the origin of the faults are only the architectural registers (i.e., microarchitecture agnostic study). Duan *et al.* in [43] propose the

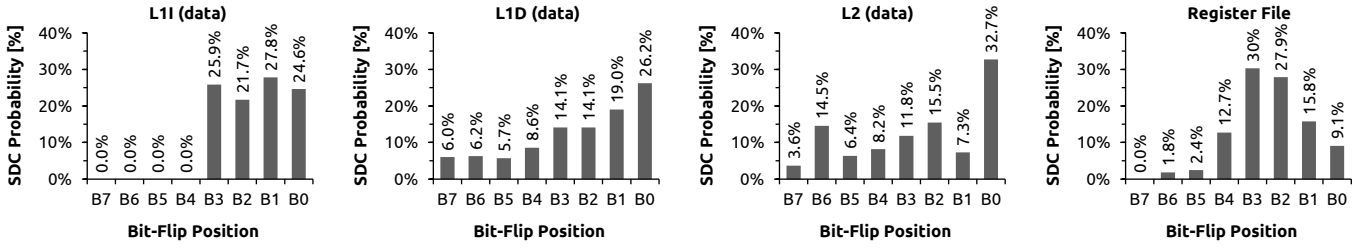

Fig. 11. The probability of bit positions in a 64-bit word to result in a silent data corruption under the occurrence of a single-bit flip.

use of boosted regression trees as a predictive model for AVF using ACE analysis, which cannot distinguish among SDCs and other fault effects, and thus, no essential outcomes can occur for any correlation. There are also similar studies that focus on the microarchitecture-level fault occurrences, and thus, consider the hardware masking, however, to the best of our knowledge, none of them focus on exploring the propagation of hardware-induced faults and investigate the correlation of several microarchitectural structures to the silent data corruption distribution. For example, Li et al. in [27] and [44] study the propagation of permanent faults to the software considering functional blocks, the register file, the register alias table, and the reorder buffer. In [44] the authors introduce the SWAT-Sim, which is a fault injection tool to study the system-level manifestations of permanent faults. Sangchoolie *et al.* in [45] present a recent ISA-level fault injection study, in which they present approaches that improve the controllability and efficiency of ISA-level fault injection techniques. [45] is completely in line with this study, as it concludes that address- and instruction-related injections are more likely to result in a crash, while injections in data variables are more likely to result in an SDC. Although all these studies provide meaningful results and insights, none of them explicitly studies the anatomy of the silent data corruptions in a cross-layer manner, as this study does. Moreover, a recent study [16] has demonstrated the diverging reliability evaluation results and the significant error margin that software-level studies can introduce. To this end, this study considers hardware-induced faults and presents, for the first time, the anatomy of these faults with respect to silent data corruptions.

8 Conclusion

In this paper we presented, for the first time, an in-depth analysis of the behavior and the cross-layer propagation of hardware faults. Our work focuses on eleven major microarchitectural structures of an out-of-order microprocessor that result in SDCs. Our analysis revealed several important insights, including: (i) the portion of SDCs that can be attributed exclusively to eleven different hardware structures, (ii) the instruction-related parameters that are more likely to result in a silent data corruption, (iii) the extent to which the operating system affects the silent data corruption rate, and (iv) the byte positions of a word stored in a hardware structure which are more likely to result in SDCs. All these insights that this study provides, advocate that chip manufacturers can take several steps to improve their next-generation microprocessors. They need to: (i) enhance the error detection and correction capabilities; (ii) introduce sophisticated redundancy and fault tolerance techniques to mitigate the impact of silent data corruptions; (iii) implement error logging and monitoring mechanisms to allow for better visibility into the occurrence and patterns of data corruptions; and (iv) refine their testing and validation process to include specific scenarios that emulate data corruptions and hardware faults. Further, our insights of single event upsets (transient faults) can be similarly extended to multi-bit upsets, and to design bugs, permanent and intermittent faults and explore the differences and similarities that all these kinds of corruptions provide to the SDCs.

Acknowledgments

This project has received funding from the European Union's Horizon Europe research and innovation programme under grant agreement No. 101070238 (NEUROPULS), No. 101097224 (REBECCA), and No. 101093062 (Vitamin-V). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them. It has also received funding by research gifts from Meta and Intel.

References

[1] C. Constantinescu, I. Parulkar, R. Harper, and S. Michalak, "Silent data corruption — myth or reality?" in *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, 2008, pp. 108–109.

[2] E. Ibe, H. Taniguchi, Y. Yahagi, K.-i. Shimbo, and T. Toba, "Impact of scaling on neutron-induced soft error in srams from a 250 nm to a 22 nm design rule," *IEEE Transactions on Electron Devices*, vol. 57, no. 7, pp. 1527–1538, 2010.

[3] S. Wen, R. Wong, M. Romain, and N. Tam, "Thermal neutron soft error rate for srams in the 90nm–45nm technology range," in *IEEE International Reliability Physics Symposium*, 2010, pp. 1036–1039.

[4] R. Lucas, "Top ten exascale research challenges," in *DOE ASCAC Subcommittee Report*, 2014.

[5] A. Chatzidimitriou, G. Papadimitriou, D. Gizopoulos, S. Ganapathy, and J. Kalamatianos, "Assessing the effects of low voltage in branch prediction units," in *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2019.

[6] G. Papadimitriou, M. Kaliorakis, A. Chatzidimitriou, D. Gizopoulos, P. Lawthers, and S. Das, "Harnessing voltage margins for energy efficiency in multicore cpus," in *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO-50 '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 503–516.

[7] G. Papadimitriou, A. Chatzidimitriou, and D. Gizopoulos, "Adaptive voltage/frequency scaling and core allocation for balanced energy and performance on multicore cpus," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019, pp. 133–146.

[8] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores That Don't Count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 9–16.

[9] H. D. Dixit, S. Pendharkar, M. Beadon, C. Mason, T. Chakravarthy, B. Muthiah, and S. Sankar, "Silent Data Corruptions at Scale," 2021. [Online]. Available: https://arxiv.org/abs/2102.11245

[10] R. W. Hamming, "Error detecting and error correcting codes," *The Bell System Technical Journal*, vol. 29, no. 2, pp. 147–160, 1950.

[11] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, "Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 467–478.

[12] A. Chatzidimitriou, G. Papadimitriou, C. Gavanas, G. Katsoridas, and D. Gizopoulos, "Multi-bit upsets vulnerability analysis of modern microprocessors," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*, 2019, pp. 119–130. [Online]. Available: https://doi.org/10.1109/IISWC47752.2019.9042036

[13] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field," in *IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 415–426.

[14] D. Kuvaiskii and C. Fetzer, "Δ-encoding: Practical encoded processing," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 13–24.

[15] M. Didehban and A. Shrivastava, "nzdc: A compiler technique for near zero silent data corruption," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, 2016, pp. 1–6.

[16] G. Papadimitriou and D. Gizopoulos, "Demystifying the system vulnerability stack: Transient fault effects across the layers," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 902–915.

[17] "Pytorch elastic documentation," 2022. [Online]. Available: https://pytorch.org/elastic/0.1.0rc2/overview.html
[18] G. Losa, A. Barbalace, Y. Wen, H.-R. Chuang, B. Ravindran, and M. Sadini, "Transparent fault-tolerance using intra-machine full-software-stack replication on commodity multicore hardware," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, 2017, pp. 1521–1531.
[19] G. Papadimitriou and D. Gizopoulos, "Avgi: Microarchitecture-driven, fast and accurate vulnerability assessment," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 935–948. [Online]. Available: https://doi.org/10.1109/HPCA56546.2023.10071105
[20] ——, "Anatomy of on-chip memory hardware fault effects across the layers," *IEEE Transactions on Emerging Topics in Computing*, pp. 1–12, 2022. [Online]. Available: https://doi.org/10.1109/TETC.2022.3205808
[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 *Proceedings of the 50th Annual Design Automation Conference*, ser. DAC '13. New York, NY, USA: Association for Computing Machinery, 2013. [Online]. Available: https://doi.org/10.1145/2463209.2488859
[22] 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," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, p. 1–7, aug 2011.
[23] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. Kim, "Robust system design with built-in soft-error resilience," *Computer*, vol. 38, no. 2, pp. 43–52, 2005.
[24] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Design, Automation & Test in Europe Conference & Exhibition*, 2009, pp. 502–506.
[25] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Fourth Annual IEEE International Workshop on Workload Characterization. WWC-4 (Cat. No.01EX538)*, 2001, pp. 3–14.
[26] K. R. Walcott, G. Humphreys, and S. Gurumurthi, "Dynamic prediction of architectural vulnerability from microarchitectural state," in *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ser. ISCA '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 516–527. [Online]. Available: https://doi.org/10.1145/1250662.1250726
[27] M.-L. Li, P. Ramachandran, U. R. Karpuzcu, S. K. S. Hari, and S. V. Adve, "Accurate microarchitecture-level fault modeling for studying hardware faults," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 105–116.
[28] X. Li, S. V. Adve, P. Bose, and J. A. Rivers, "Online estimation of architectural vulnerability factor for soft errors," in *2008 International Symposium on Computer Architecture*, 2008, pp. 341–352.
[29] N. J. George, C. R. Elks, B. W. Johnson, and J. Lach, "Transient fault models and avf estimation revisited," in *IEEE/IFIP International Conference on Dependable Systems & Networks*, 2010, pp. 477–486.
[30] G. Papadimitriou and D. Gizopoulos, "Characterizing soft error vulnerability of cpus across compiler optimizations and microarchitectures," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021, pp. 113–124. [Online]. Available: https://doi.org/10.1109/IISWC53511.2021.00021
[31] A. A. Nair, L. K. John, and L. Eeckhout, "Avf stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors," in *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, 2010, pp. 125–136.
[32] Z. Zhao, D. Lee, A. Gerstlauer, and L. K. John, "Host-compiled reliability modeling for fast estimation of architectural vulnerabilities," in *IEEE Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2015.
[33] G. Papadimitriou, D. Gizopoulos, A. Chatzidimitriou, T. Kolan, A. Koyfman, R. Morad, and V. Sokhin, "Unveiling difficult bugs in address translation caching arrays for effective post-silicon validation," in *2016 IEEE 34th International Conference on Computer Design (ICCD)*, 2016, pp. 544–551. [Online]. Available: https://doi.org/10.1109/ICCD.2016.7753339
[34] 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," *IEEE Transactions on Computers*, vol. 71, no. 10, pp. 2358–2369, 2022. [Online]. Available: https://doi.org/10.1109/TC.2021.3128501
[35] I. Tsiokanos, G. Papadimitriou, D. Gizopoulos, and G. Karakonstantis, "Boosting microprocessor efficiency: Circuit- and workload-aware assessment of timing errors," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*, 2021, pp. 125–137. [Online]. Available: https://doi.org/10.1109/IISWC53511.2021.00022
[36] 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*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 461–472. [Online]. Available: https://doi.org/10.1145/1815961.1816023
[37] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings. 36th Annual IEEE/ACM International Symposium on Microarchitecture, 2003. MICRO-36.*, 2003, pp. 29–40.
[38] B. Fang, P. Wu, Q. Guan, N. DeBardeleben, L. Monroe, S. Blanchard, Z. Chen, K. Pattabiraman, and M. Ripeanu, "Sdc is in the eye of the beholder: A survey and preliminary study," in *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, 2016, pp. 72–76.
[39] Q. Guan, N. DeBardeleben, S. Blanchard, and S. Fu, "Empirical studies of the soft error susceptibility ofsorting algorithms to statistical fault injection," in *Proceedings of the 5th Workshop on Fault Tolerance for HPC at EXtreme Scale*, ser. FTXS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 35–40. [Online]. Available: https://doi.org/10.1145/2751504.2751512
[40] Z. Li, H. Menon, D. Maljovec, Y. Livnat, S. Liu, K. Mohror, P.-T. Bremer, and V. Pascucci, "Spotsdc: Revealing the silent data corruption propagation in high-performance computing systems," *IEEE Transactions on Visualization and Computer Graphics*, vol. 27, no. 10, pp. 3938–3952, 2021.
[41] X. Xu and M.-L. Li, "Understanding soft error propagation using efficient vulnerability-driven fault injection," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 2012, pp. 1–12.
[42] S. K. S. Hari, S. V. Adve, and H. Naeimi, "Low-cost program-level detectors for reducing silent data corruptions," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, 2012, pp. 1–12.
[43] L. Duan, B. Li, and L. Peng, "Versatile prediction and fast estimation of architectural vulnerability factor from processor performance metrics," in *2009 IEEE 15th International Symposium on High Performance Computer Architecture*, 2009, pp. 129–140.
[44] 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," *SIGARCH Comput. Archit. News*, vol. 36, no. 1, p. 265–276, mar 2008. [Online]. Available: https://doi.org/10.1145/1353534.1346315
[45] B. Sangchoolie, R. Johansson, and J. Karlsson, "Light-weight techniques for improving the controllability and efficiency of isa-level fault injection tools," in *2017 IEEE 22nd Pacific Rim International Symposium on Dependable Computing (PRDC)*, 2017, pp. 68–77.

George Papadimitriou is a Postdoctoral Researcher in the Department of Informatics & Telecommunications at National & Kapodistrian University of Athens in Greece. He received his PhD in Computer Science from the same University in 2019. His research focuses on dependability and energy-efficient computer architectures, microprocessor reliability, functional correctness of hardware designs and design validation of microprocessors, in which he has published more than 40 papers in international conferences and journals. He is an IEEE member.

Dimitris Gizopoulos is Professor at the Department of Informatics & Telecommunications of the National & Kapodistrian University of Athens in Greece where he leads the Computer Architecture Laboratory. The group's research focuses on the dependability, the energy-efficiency and the performance of computer architectures. Gizopoulos has published more than 180 papers in top-tier conferences and journals, has served and is currently serving as Associate Editor for several IEEE and ACM Transactions and Magazines and as member of Program, Organizing and Steering Committees of IEEE and ACM conferences. He served as the General Chair of the 53rd and the 54th editions of the IEEE/ACM International Symposium on Microarchitecture (MICRO). Gizopoulos is an IEEE Fellow, a Golden Core member of the IEEE Computer Society and a Senior ACM member.