# Anatomy of Microarchitecture-Level Reliability Assessment: Throughput and Accuracy

Athanasios Chatzidimitriou Dimitris Gizopoulos
Department of Informatics & Telecommunications
University of Athens
Athens, Greece
{achatz, dgizop}@di.uoa.gr

***Abstract*—The increasing density and complexity of modern microprocessors, which is driven by manufacturing technologies scaling, significantly affect their reliability. Reliability evaluation during the early design stages is a challenging process for microprocessor designers. Statistical fault-injection on microarchitecture simulators is commonly used, among other techniques, since it can deliver early and accurate reliability estimations for many important microprocessor hardware structures. However, full-system microarchitectural simulators have a relatively small simulation throughput. Thus, the number of injection experiments that can be performed during a fault injection campaign can be limited and therefore lead to smaller statistical significance of the reliability assessment.**

**Aiming to boost the throughput of microarchitecture-level fault injection, we present, in this paper, a multi-faceted microarchitecture-level toolset for reliability assessment of modern microprocessors. The framework is built around the Gem5 simulator and provides several modes of operation which employ acceleration features for all stages of a fault-injection based reliability assessment campaign. The tool throughput and the accuracy of the delivered reliability assessments can be traded off and allow architects to make informed decisions about the most suitable error protection mechanisms of any given microarchitecture and workload by studying the reports delivered by the toolset. We provide experimental results of the different modes of the toolset for both the x86 and ARM out-of-order models of Gem5. Our experimental results show that up to 8x acceleration of the fault injection campaigns can be achieved with less than 0.5 percentile points of accuracy loss.**

***Keywords—microprocessor reliability evaluation; statistical fault injection; microarchitecture simulators***

## I. INTRODUCTION

The rapid development of semiconductor technologies is continuously increasing the density and complexity of modern microprocessor chips in favor of performance and functionality. This extreme scaling however has a negative impact on the reliable operation of microprocessors, making them more vulnerable to cosmic radiation, latent manufacturing defects, device degradation and low voltage operation [2] [6] [8] [24]. As a result, transient faults tend to appear more frequently, which makes them a major threat for the reliable system operation. Luckily, not all faults can harm the system stability and not all faults have the same severity. Several metrics have been proposed to express aspects of the system vulnerability from different observation angles. Architectural Vulnerability Factor (AVF) [21] was proposed by Mukherjee et al. to express the vulnerability factor for transient faults; AVF of a microprocessor hardware structure is the probability that a bit flip in it affects the correct execution of a program. Similarly, IVF [25] and H-AVF [5] have been proposed to express the vulnerability of microprocessor structures to intermittent and permanent faults.

Sridharan and Kaeli later proposed finer levels of abstraction for the AVF, introducing the Hardware Vulnerability Factor (HVF) [30] and the Program Vulnerability Factor (PVF) [28], each to quantify the portion of AVF that can be attributed to the Hardware layer (including the microarchitecture) and the Software layer, respectively.

Vulnerability assessment is very important during the early design stages. Fault-tolerance mechanisms impose area, power and performance overheads which can, for example, vary in a range between 1% and 125% (in terms of extra memory capacity) for typical memory error detection and correction [20]. Thus, significant effort must be devoted to effectively measure the vulnerability of a system design as early as possible and make suitable design decisions for error protection.

Simulation tools that model a system being designed are widely used to assess the system vulnerability. The detail of their models depends on the requirements of the assessment. These requirements may vary on the level of estimation accuracy, time limitations, scope (e.g. full-system, processor, component etc.) and differences in the software workloads. Microarchitectural simulators are excellent candidates for early reliability assessment because they provide an ideal balance between sufficient modeling detail for most important components and a reasonable simulation throughput. Compared to simpler functional simulators, they offer microarchitectural details on their implementation while, compared to RTL models, they are several orders of magnitude faster and can combine full-system capabilities. In addition, microarchitecture simulators are very flexible and support easy modifications of the hardware structures configurations and thus offer easier exploration of design alternatives. Different approaches, like statistical fault injection [7] [9] [10] [12] [14] [15] [16] [26] [33], Architectural Correct Execution (ACE) analysis [4] [21] [22] and probabilistic approaches [1] [11] [31] can be employed on top of microarchitecture-level simulators for measuring the vulnerability of microprocessor components to soft errors. ACE analysis and probabilistic methods are quite fast methods but require significant modifications of the simulator and tend to overestimate the vulnerability by 3x to 7x compared to fault injection [12] [19] [32]. Fault injection on the other hand needs minimum changes in the simulator, delivers very accurate reliability reports, but requires large numbers of simulations to reach statistical significance [18].

In this work, we focus on the anatomy of microarchitecture-level reliability assessments using fault injection. We analyze the fault injection simulation lifetime, and explore several acceleration opportunities of fault injection for transient faults. We propose a variety of approaches for finer balancing of the assessment accuracy and the throughput of fault injection campaigns. We also present a microarchitecture-level fault injection framework which integrates the different acceleration options and offers fast fault-injection based reliability measurements. Section II describes the enhancement of a fault injection framework with the proposed acceleration features; Section III presents how higher-level workload attributes can be considered for result prediction and thus early termination of fault injection; Section IV presents our experimental results for x86 and ARM microprocessor configurations and Section V concludes the paper.

## II. Assessing System Vulnerability

### A. Background, Definitions and Concepts

The Architectural Vulnerability Factor (AVF) of a hardware component was introduced by Mukherjee et al. as the probability that a fault in a hardware component affects program execution [21]. AVF was presented alongside with Architectural Correct Execution (ACE) analysis, a method for fast AVF estimation. The set of bits that are required for correct operation are called ACE bits; the remaining bits are called un-ACE bits. ACE analysis includes the notion of time, and thus, each bit is characterized as ACE or un-ACE for a certain period of time. AVF can then be defined as the fraction of ACE bits in a hardware component.

Identifying ACE bits is a difficult process since it refers to a full-system aspect. Existing techniques of ACE analysis, tend to overestimate the AVF by up to 7x [12] [19] and even refined methods still report up to 3x overestimations [32]. To better address this issue, Sridharan and Kaeli have introduced the System Vulnerability Stack [30], where the AVF is separated in many fine-grained layers, which are however substitutes of two major ones: Hardware and Software layers. They also introduced the Hardware Vulnerability Factor (HVF) [30] and the Program Vulnerability Factor (PVF) [28], respectively. Separating AVF into hardware and software layers can accelerate the assessment process since each measurement can be performed with different tools. PVF for instance can be estimated using fast functional emulators without microarchitectural details, only to measure the software masking effects.

Our work proposes acceleration features and techniques for faster vulnerability estimation through fault injection. The fault injection framework we built provides both AVF estimation and hardware vulnerability estimation (not to be confused with HVF). Since we utilize concepts like the system vulnerability stack, it is important to clearly highlight the differences with the existing metrics and define the limits of each approach.

Statistical fault injection is a reliability estimation technique that can bypass the complexity of ACE analysis and directly access the program output that was produced with the injected faults. Therefore, it can offer complete (full-system) AVF estimation. Techniques that target to profile and simulate incomplete portions of a program, such as SimPoints [27] the capability of accessing the effect of faults on the program output. The simulated part of the program may include masking effects of both software and hardware layers. Comparing the system or the architectural state at the end of a portion will give an estimation that does not clearly belong to one layer of the system vulnerability stack. Stating that it corresponds to the full-system AVF may be inaccurate and misleading, since there is not enough evidence of the participation of incomplete program parts to the program's final output.

On the other hand, a clear separation of HVF and PVF can be achieved by stopping a simulation whenever a fault reaches an architectural visible resource [28] [29]. In [29], a *software resource* is defined as any independently-addressable architectural structure. This abstraction level raises several issues with the definition of PVF that are not clearly clarified, especially for memory structure estimations. Virtual memory complicates things significantly. Memory is indirectly accessed by the software, involving (in most cases) a hardware translation mechanism. A question that is hard to be answered is: Which portion of the virtual address space should be considered a software resource, the mapped one or the complete address space? For the mapped part, multiple addresses can correspond to the same hardware structure and many can be temporarily mapped to peripheral devices. From the HVF point of view, any valid cache line is a visible resource as long as the same line is not valid on a higher cache level. But an eviction caused by the microarchitecture can immediately change this condition and a fault that was characterized as software visible can turn to invisible, without software interference.

To address such issues, we use a slightly different variant of the Hardware/Software vulnerability separation than HVF and PVF. Since both determine jointly the AVF, defining one also defines the other. For the software side we use the concept of *instruction flow*, to which we will refer as *program flow*. Program flow is a subset of software resources and limits the software bounds only to those resources that are actually used. This applies to both registers and memory. The program flow consists of: (i) the decoded instruction and its operands, (ii) the data transactions in both registers and memory, (iii) the program instruction order, and (iv) the execution time of each instruction (to monitor performance deviations). Therefore, an architecturally mapped register that is not used by the program and does not participate in the program flow is not considered to reach the software layer, in spite of the fact that it is part of the architectural state.

*Software masking* can then be defined to express the probability that a fault which was involved on the instruction flow to be masked by the program. On the complementary hardware side, we define as *hardware vulnerability* the probability that a fault on a hardware structure reaches a visible point of the program flow. Therefore, unused hardware resources are characterized as masked on the hardware layer, even if part of them is mapped to architectural resources.

### B. GeFIN – A Gem5 fault injection framework

In this work, we employ *GeFIN*, a microarchitecture level fault injection framework first presented in [16] built on top of the Gem5 simulator [3]. The framework consists of a modified Gem5 version that allows fault injection along with

instrumentation for running and controlling simulation campaigns. It supports all types of fault-models (transient, permanent and intermittent) in single or multiple fault configurations of any combination. In this paper, we focus of the use of GeFIN for transient single-bit faults.

The GeFIN framework uses *faultmasks* to describe injection of faults. Each faultmask can include one or multiple faults for the simulation. It carries sufficient information to accurately target one or multiple component(s) at a given time or period. Each fault is described by: (i) thread id (cpu id), (ii) microarchitecture component, (iii) position within the component (bit granularity), (iv) fault model, (v) clock cycle of the occurrence, (vi) duration (for intermittent faults) and (vii) mask effect (bit flip, stuck-at 1, stuck at 0).

GeFIN uses configuration *presets*. Each preset consists of attributes, such as the ISA, memory configuration, CPU core (in-order, out-of-order etc.), multicore setup, system setup, disk images, kernels etc. along with GeFIN attributes and a list of supported components for injections. New presets can be easily added to cover different requirements.

A complete *Fault Injection Campaign* is initiated on GeFIN by selecting: (i) the configuration preset, (ii) the benchmark, (iii) the fault model, (iv) the number of faultmasks for the simulation and (v) the number of workers (spawned threads to work in parallel). Multiple campaigns can also be scheduled independently through an appropriate configuration file.

When a simulation ends, GeFIN stores all its outputs for later processing. The fault effects *classification* phase is an offline process initiated (manually or automatically) after the end of an injection campaign. Since each configuration preset may produce different outputs, each preset is accompanied with its own default parser script for the classification phase. Different versions of the parser scripts can be used additionally to report different fault effect classes.

The level of flexibility and ease of expansion that GeFIN introduces makes it a perfect candidate for our work.

### C. Fault simulation epochs

There are three possible important events during a fault injection run that define different epochs of the simulation. The events are:

- **Fault Injection event**. It corresponds to the time (cycle) of the bit-flip occurrence.
- **First access event**. The first access of the faulty entry, which can be a *read* or *write*. This event may never happen during a simulation.
- **Visible fault effect event**. The first visible effect of the fault on the program flow. This event can only occur after a read access of the fault.

These three events define five different epochs of a fault injection run (see Fig. 1):

**Pre-fault epoch.** The epoch starts with the simulation and ends at the time of the injection of the fault. The pre-fault epoch is out of interest for reliability estimation since it has no residing faults. The simulation is identical to the fault-free (golden) simulation.

**Idle epoch.** This epoch starts at fault injection time and ends either on the first access of the faulty entry or at the end of simulation; whichever comes first. A fault has been injected into the system and resides silently unused during Idle epoch.

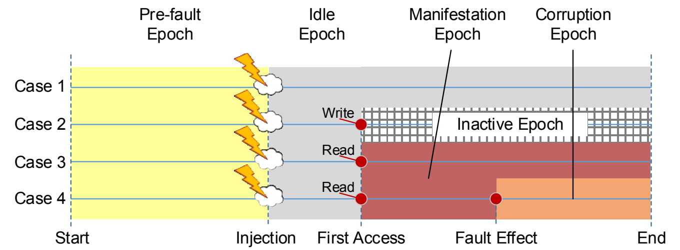


Fig. 1: Fault injection timeline illustrates the different epochs defined by the fault injection event, the first access of the faulty entry and the first program visible fault effect. In Case 1, the fault remains unused (no access); in Case 2, the fault gets overwritten before being read; in Case 3 the fault is read but its effect is software masked and in Case 4 the fault corrupts the program flow.

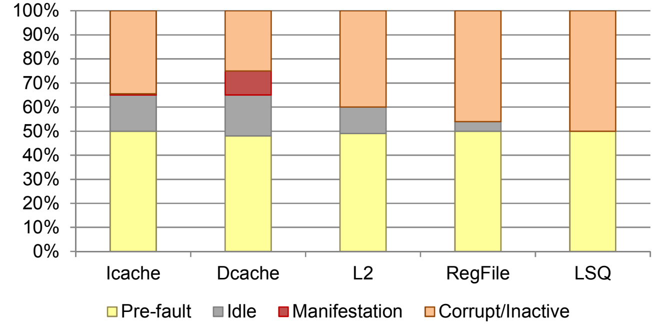


Fig. 2: Breakdown of the duration of the simulation epochs.

**Inactive epoch.** The fault gets overwritten and no longer exists on the simulation.

**Manifestation epoch.** The first access of the faulty entry marks the beginning of the Manifestation epoch. During this epoch, a fault has been accessed and can potentially harm the program flow.

**Corruption epoch.** When a fault reaches the program flow, the Corruption epoch starts. From this point onwards, the fault resides in the software layer and may cause corruption of the program output, may crash the system or may be masked at the software layer.

Idle and Manifestation are the most important epochs, since they contain the fault injection, fault propagation and fault effect events. Although the Manifestation epoch describes the complete lifetime of a fault before propagating to the program flow, Idle epoch is an important part of the simulation because it contains all the hardware access patterns related to if, when and where the fault will be used by the microarchitecture. The Idle and Manifestation epochs contain all the required information to characterize a fault effect for the hardware layer. However it must be pointed out that these epochs can also include parts of no interest. For instance, cases where a fault is initially propagated to the microarchitecture but then it gets hardware masked (e.g. due to a misspeculation and a pipeline flush). Such cases waste simulation time as the fault has no effect.

It is important to note that the described epochs are determined only by the first occurrence of the three important events, which however may not depend on each other. This means that the first visible effect of a fault may be due to the second or third access of the fault and not necessarily the first. We have modified GeFIN to identify these events and determine the corresponding epochs.

Fig. 2 presents the duration of these epochs during the entire simulation timeline. The Pre-fault epoch consumes an average 49% of total simulation time with the Corruption and the Inactive epoch following with an average of 38%. We can also see that the most critical epochs for the effect of a fault (the Idle and the Manifestation) only hold an average of 13% of simulation time.

The fact that only 13% of the baseline simulation time is enough to characterize the hardware vulnerability is the motivation of our work. Our efforts focus to reduce the simulation time only to the critical parts.

### *D. Acceleration features*

In this subsection we will describe the new acceleration features integrated into GeFIN. The baseline tool presented in [16] was enhanced to monitor and detect the events and successfully identify the epochs described in the previous section. By knowing which epoch the simulation is going through, the corresponding acceleration potential is automatically revealed. Each epoch has some special attributes to which the simulator can adapt and either reduce some of the monitoring overhead, or even apply more aggressive approaches, such as fast forwarding or fault effect prediction.

The features are separately described as *direct* features, the ones that lead to pure acceleration without any loss of accuracy in the vulnerability measurements; and *indirect* features, which can further reduce the simulation time significantly but can lead to some loss of accuracy in the vulnerability measurements. All these features can be independently enabled or disabled, giving the flexibility of different combinations, according to the estimation requirements of a particular campaign.

#### *1. Direct Features*

**Enhanced checkpointing and fast-forwarding.** The *Pre-fault* epoch does not contain any faults and thus, it is out of interest for the reliability assessment. Each GeFIN run is aware of the fault injection time and therefore can eliminate the *Pre-fault* epoch by starting simulation from the point of the fault injection. Checkpointing is a widely known method employed to skip insignificant parts of the simulation and fast-forward the execution of a program. However, checkpoints come at a cost. The loss of microarchitectural state cannot be ignored, especially when considering large trained components, such as cache memories. This is the reason why checkpoints are usually accompanied with warm-up intervals.

To address this accuracy loss, we have enhanced Gem5 checkpoint infrastructure to also include the state and data of all cache memories. This significantly reduces the loss of microarchitectural state and allows direct restoring of checkpoints, without the requirement of warm up. With the enhanced checkpoints, we can completely skip the Pre-fault epoch with a minimal deviation of the final vulnerability estimations, caused by the minimal pipeline detail loss (which is measured to less than 1 percentile point).

**Early stop on overwrite/invalid.** Injected faults that get overwritten before being read cannot harm the system state and they can be considered as masked prior to the completion. The same applies to faults injected in inactive (invalid) structure entries, which will be overwritten upon allocation. The epoch that follows the overwriting is the Inactive epoch, which can be safely discarded (simulation stops).

Skipping the Inactive epoch can save up to 50% of a campaign time without any impact on the final result. GeFIN was enhanced to recognize Inactive epochs and stop the simulation, marking the injected fault as masked.

#### *2. Indirect features*

**Early stop on program corruption.** This feature aims to skip the entire Corruption epoch. By monitoring the instruction commit flow, in terms of static instructions, operands, data transactions and time, we can detect changes on the expected program flow. Any kind of mismatch that is observed to the program flow marks the *fault effect* event. Stopping at that time, changes the observation point of the simulation result and limits the AVF estimation only to the *hardware* portion, as described in Section II.A. If this feature is enabled during an injection campaign the software masking portion of AVF is ignored. This is a useful feature when the hardware vulnerability report it delivers is combined with a software vulnerability estimation (using for example a functional simulator and software-level fault injections).

GeFIN was expanded with the capability to monitor and compare on-line the commit stage of the processor and stop on the first mismatch of the program flow. An initial fault-free execution is used to produce the correct instruction trace, which is then supplied to the campaign workers for on-line comparison.

**Early switch to functional emulation.** Unlike the previous features that target to skip portions of the simulation, early switch intends to fast-forward an epoch, in particular the Corruption epoch. Gem5 supports different operation modes. Among others, it includes simple functional CPU models which have no microarchitectural details and operate at a 20x higher throughput than the detailed models.

Following the *early stop on program corruption* feature, and considering that Gem5 supports switching to the functional model at any time, the missing *software vulnerability* part of the complete AVF estimation for that particular experiment can be measured using the same tool. By switching to emulation mode, we can move the observation point at the end of the program execution and report the complete AVF measurement. The source of potential accuracy losses is only the possibility that the fault would cause further problems after its first visible effect; switching to emulation will only capture this effect on the final output. However, our results (Section IV.B) show that these cases are rare and barely have any impact on the final fault effects classification.

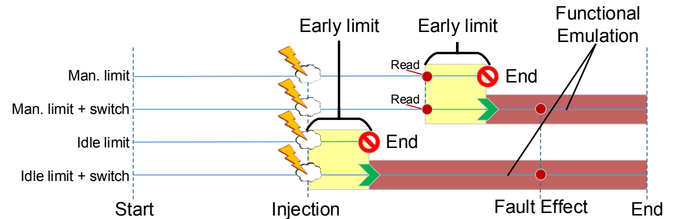


Fig. 3: Effect of Manifestation early limit and Idle early limit individually or combined with early switch feature.

**Early limit.** This feature is used to stop the simulation early after the first access of the faulty entry. It condenses the manifestation epoch to a user-defined amount of time, during

which the fault has the opportunity to manifest itself. This feature is intended to be used along with *fault effect prediction*. In Section III we present how we can effectively use prediction to get high acceleration with minor loss of accuracy, depending on behavior patterns of some hardware structures.

Early limit comes also in an alternative form aiming to condense the Idle epoch. This form of the feature is only recommended for use in strict campaign time constraints. When employed, it can lead to significant loss of accuracy since it adds a large number of unknown cases. However, it can be combined with early switch to indicate a switch instead of stop after the defined time.

Fig. 3 illustrates how early limit and early switch features apply to the simulation timeline.

## III. Workload Studies

Although skipping uninteresting epochs and stopping earlier can be effective for speeding up the vulnerability estimation, the simulation still doesn't completely focus on critical parts of the fault simulation timeline. As we already mentioned, the Idle and Manifestation epochs also include masked cases. In the workloads study of this section we investigate component behaviors in order to detect and stop simulations that have little or no chances to report any fault effects. At this point, it is important to mention that there is a fundamental difference between components in the pipeline (e.g. register file) and components out of it (e.g. cache memories). Allocated pipeline resources have a very short residency time and hence, very small Idle epochs, while on the other hand, memory entries (especially in lower levels) may even never be used again (long Idle or Manifestation epochs).

Using a very extensive simulation-based study, we collected a set of interesting observations that reveal opportunities for further speeding up the vulnerability estimation process. Our benchmarks base consists of 10 benchmarks (see Section IV.C), simulated for 5 hardware components in 2 different ISAs with 2,000 faults injected per component. This corresponds to 40,000 injections for each component (20,000 injections per ISA).

This section presents the fault effect patterns observed in our study for different hardware components and can be taken into account for predicting the effects of faults significantly earlier.

**Register file.** Out-of-order processors come in two flavors: The traditional Reorder Buffer & Architectural register file concept, where the ROB also holds the data for the pending dynamic instruction values, and the Physical register file & Active List (ROB term is also commonly used) scheme, where all data is stored to the physical register file and the ROB contains pointers to those resources. Both approaches have pros and cons and both can be found in modern processor chips. Gem5 uses the latter approach on its out-of-order core, with a Physical register file combined with a rename map to indicate the current up-to-date committed values for the architectural registers.

Allocated resources in the physical register file come at two different types: *architectural* and *dynamic*. In a typical case, the physical register file has allocated at a minimum the number of architectural registers (along with any micro-implementation static registers; e.g. Fault handling, ALU side utility etc.) at any time, and from that point on, it additionally allocates resources for the in-flight dynamic instructions. Each dynamic instruction usually allocates two source and one destination registers. Upon renaming, the real dependencies are resolved and the operands of static instructions may change to temporal resources. The order is restored upon commit (alt. retire) stage and the rename map is updated to the current architectural values. We will use the terms *dynamic registers* and *architectural registers* to express the nature of the allocated resource in the physical register file.

The fundamental difference between architectural and dynamic registers is their residency time. Dynamic registers remain active as long as their instruction lives inside the pipeline (usually a few clock cycles) while on the other hand, architecturally mapped registers are part of the program state and may be used millions of cycles later, or even not used at all. This implies that any fault in a dynamic register has a short window of opportunity to lead to a state corruption. Our experimental sample of 40,000 injections confirms that all of the faults that hit dynamic registers and led to a state corruption, did reach a visible point in the program flow in less than 500 clock cycles, indicating a safe point of time to stop the simulation without loss of accuracy.

The architectural registers on the other hand do not follow this behavior pattern. In fact, each ISA reports different behavior and thus, our analysis is separated on an ISA basis.

ARM ISA defines 31 architectural registers, 16 for the used mode and 15 for the FIQ, IRQ, SVC, Undefined and Abort modes. In addition, Gem5 implementation also includes 12 microarchitectural registers, totaling up to 43 entries in the rename map.

The hardware vulnerability of architectural registers is presented in Fig. 4 (top). The Stack Pointer of SVC, ABT and IRQ modes seems to be *highly* critical (100% vulnerability) while the Link Register in these modes is *not* critical and always gets masked (0% vulnerability). The diagram does not include the 12 micro-architectural registers (due to space limitations) that are also held in rename map, which also have a similar trend in severity: some are highly critical and others are not critical at all. It is important to mention that these behaviors highly rely on the system platform. A system that uses Fast Interrupts would have different vulnerability compared to our system that does not use the FIQ mode at all.

The lower diagram of Fig. 4 shows how long it takes before a fault in each of the registers becomes visible on the program flow (hardware vulnerability). This time is equal to the duration of the Manifestation epoch. All registers (except for the different mode ones) have a Manifestation epoch of less than 10,000 clock cycles for more than 90% of the cases. The remaining mode registers have a standard trend on their vulnerability (as shown on the upper diagram of Fig. 4). In summary, we can conclude that it could be enough to stop simulation 10,000 clock cycles after the first fault access and only lose less than 10% of the corrupted cases. Considering that the register file vulnerability is estimated approximately at around 5% (see next section), the inaccuracy of this under-estimation will be less than 0.5 percentile points.

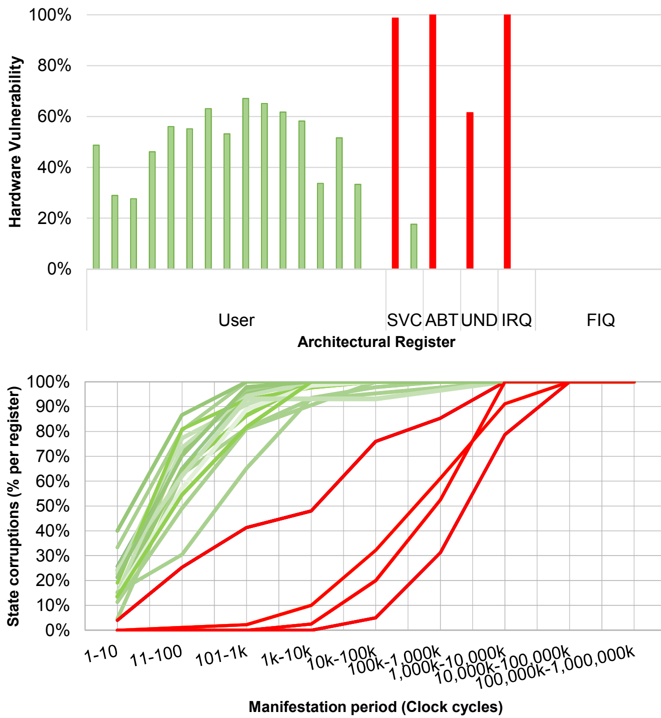


Fig. 4: ARM ISA. Red color is used for the registers with long Manifestation epochs and Green for the rest (top) Hardware vulnerability per architectural register (bottom) manifestation epoch length per architectural register

x86 ISA is a CISC ISA that supports more complex operations and direct memory operations. Fig. 5 shows the HW vulnerability of x86 registers and the Manifestation epochs duration of the x86 ISA. Unlike ARM, in x86 ISA there are no clear trends. The special purpose registers (RSP, RBP, RDI, RSI) have a relatively consistent behavior and tend to have a Manifestation epoch of 10,000 cycles for more than 90% of the cases. In contrast, for the same period, the percentage of the remaining general purpose registers is between 70% and 90%.

In practice, x86 is less "friendly" to this estimation approach, as it will lead to a larger number of unresolved corruptions for the same early-limit, compared to the ARM ISA.

**Load-Store queue.** The Load-Store Queues are pipeline components with short residency time. The role of LSQ is to support the Load-Store dependency resolution for the in-flight memory instructions. Store queue also holds data along with the referred address which is then forwarded to the memory hierarchy.

A similar timing analysis on the LSQ reveals that all the corrupt cases reached the program flow in less than 500 clock cycles. The Manifestation epoch for the corrupt cases is very short and thus, the possible estimation loss (if any) that will occur with early stopping will be virtually zero.

**Instruction & Data cache.** Instruction cache is exploited by the CPU front-end, the fetch stage. Requests that come from the instruction port either follow the program flow or, in cases of control instructions, are predicted by the branch prediction units. In both cases, the incoming memory block contains instructions with high probability to be used, due to locality and accurate branch prediction. This further implies that a faulty fetched cache block is very likely to be used by the core. The upper diagram of Fig. 6 presents the Manifestation epoch of the corrupt cases for the instruction cache, per benchmark. We can observe a clear trend, similar to the pipeline-structures, but with longer Manifestation epochs. All benchmarks follow similar trend lines, with 90% of corruption cases to appear in less than 100,000 clock cycles.

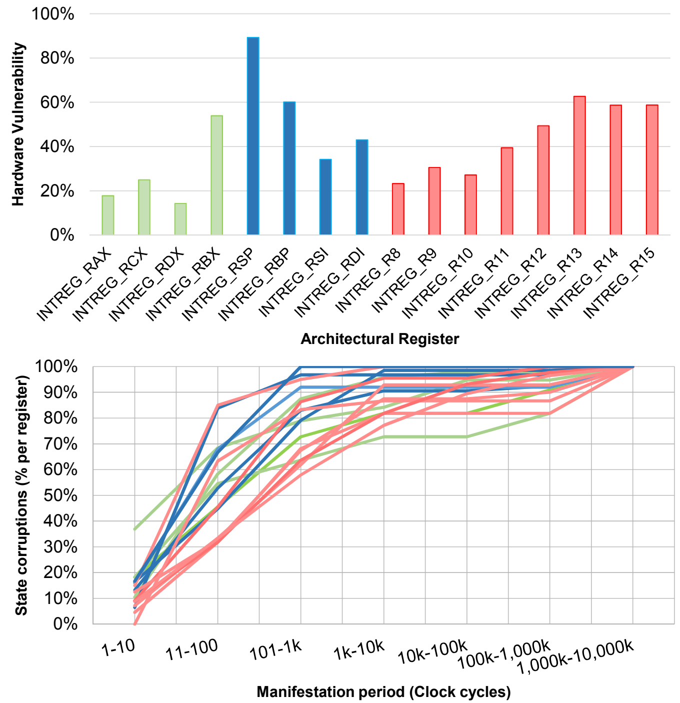


Fig. 5: x86 ISA. Blue is used for special purpose, Red for general purpose and Green is used for double purpose registers. (top) Hardware vulnerability (bottom) manifestation epoch length per architectural register.

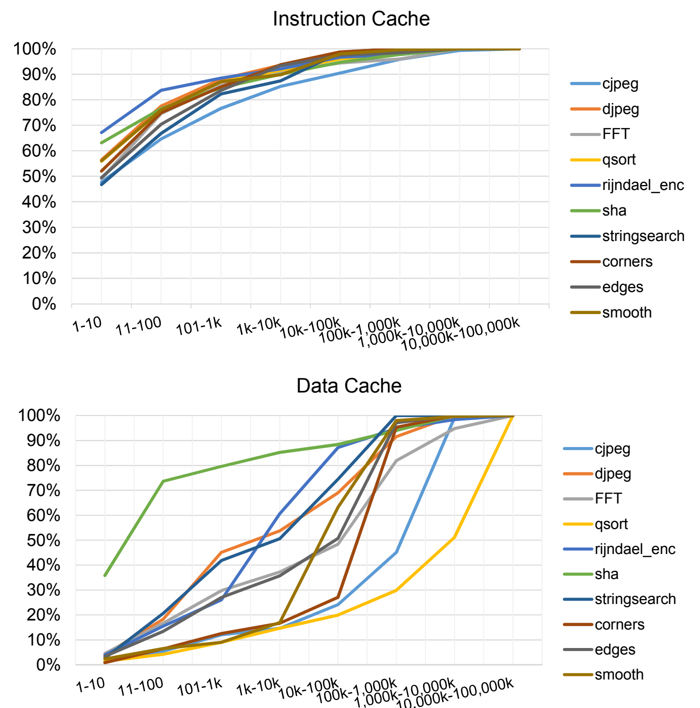


Fig. 6: Manifestation epochs duration for the L1 instruction and data cache, per benchmark.

Data cache requests on the other hand are guided by memory operations of the program flow, mispeculated load instructions, prefetching and unresolved load/store dependencies. Unfortunately, there is no standard ratio among these sources of data cache requests and the workload itself can have a severe impact on each one of them. We have experimented using functional cores against the detailed for cache memory vulnerability estimation and the results reinforced the argument that speculation can severely affect the measurements. Data cache accessing is highly unpredictable and this is also observed in the lower diagram of Fig. 6. Our analysis did not identify clear behavior patterns for the cache memories that would reveal any acceleration potential for the early stop features of our fault injection framework.

**L2 cache.** The unified second level of cache memory has similarities with L1 cache. It includes both data and instruction blocks and behaves accordingly, similar to the corresponding L1 cache. The data part is highly unpredictable similarly to the L1 data cache. However, the findings of the L1 instruction cache also apply for L2. Instructions in L2 follow the similar but more slackly trend lines than L1 instruction cache and are normalized above 80% on a Manifestation epoch of 100,000 clock cycles. Considering that the number of corruptions in the L2 that were caused because of faults in instruction blocks is very small (see the experimental results section), the loss of accuracy is marginal.

TABLE I summarizes the conclusions that can be extracted by our workload analysis. These conclusions are provided as input for the parser in order to be used for result prediction of early-stopped cases.

TABLE I: WORKLOAD ANALYSIS CONCLUSIONS

| Component | Comments |
|---|---|
| Register File | Faults in dynamic registers reach the program flow in less than 500 clock cycles. It is safe to consider a simulation as masked after 1000 clock cycles.<br>Architectural registers are significantly different. For the ARM ISA, if the fault hits a mode SP register, it can be considered as corruption. If the fault hits a mode RA register, it can be considered as masked.<br>90% of the corruptions in the general purpose registers appear in less than 10,000 clock cycles.<br>For the x86 ISA there are no safe guidelines. Special purpose registers tend to follow a pattern, with 90% of the corruptions to appear in the first 10,000 clock cycles. The remaining registers have smaller percentages that range between 70% and 90%, for the same Manifestation epoch. |
| LSQ | Program flow corruptions appear in less than 500 clock cycles. It is safe to consider a fault as masked after 1000 clock cycles in the Manifestation epoch. |
| Instruction cache | All benchmarks follow steady trend lines, with more than 90% of the corruptions to appear in less than 100,000 clock cycles. Stopping after 100,000 clock cycles and assuming that 10% of the corruptions was missed. |
| Data cache | The component is highly unpredictable and no assumptions can be drawn to make early predictions of the fault effect. |
| L2 | The unified L2 cache follows similar trends with the instruction cache for instruction cache lines. Data cache lines in the L2 are unpredictable similarly to the L1 data cache. |

## IV. Experimental Results

The enhanced GeFIN framework offers a variety of fault injection acceleration features that can be used individually or combined. In this section we present the results of our experiments for different combinations of these features and how the final vulnerability estimation as well as the campaign throughput is affected compared to the baseline fault injection without the new features.

### *A. Experimental Setup*

Our GeFIN framework can perform two types of vulnerability estimation: complete AVF estimation and Hardware vulnerability estimation (as described in Section II.A). Our experimental setup includes 4 different configurations for each mode, for a total of 8 sets of simulation campaigns. TABLE II summarizes the features that were enabled for each preset.

The choice of the configuration intends to show the tradeoff between accuracy and performance. Most of the acceleration features introduce some inaccuracy on the final outcome and thus, we expect the configurations with the most acceleration features enabled to be also the most inaccurate.

TABLE II: GeFIN Configurations

| Acronym | AVF measurement | Hardware vulnerability measurement |
|---|---|---|
| *Baseline* | ▪ Baseline fault inject | ▪ Early stop on program corruption |
| *Early* | ▪ Early stop on overwrite | ▪ Early stop on overwrite |
| *Early-Fwd* | ▪ Fast forwarding<br>▪ Early stop on overwrite | ▪ Fast forwarding<br>▪ Early stop on overwrite |
| *Early-Fwd-Sw* | ▪ Fast forwarding<br>▪ Early stop on overwrite<br>▪ Early switch after 100,000 cycles | ▪ Fast forwarding<br>▪ Early stop on overwrite<br>▪ Early stop after 100,000 cycles |

### *B. Fault effect classification*

Each fault injection run is classified depending on the effect that the fault had on the program execution. Since the observation point is different in the AVF measurements and the hardware vulnerability measurements, each mode has different classes to characterize a fault.

The AVF measurement campaigns have a finer grained fault effect classification which consists of 5 classes:

**Masked**: Complete program execution with no deviations from the fault-free simulation. The fault did not affect the system or the application in this class. The results of a masked simulation is identical to the fault-free simulation

**Silent Data Corruption (SDC)**: Complete program execution where the program output was different compared to the fault-free simulation, without any observable indications of this effect.

**Crash**: A simulation that did not reach the end of the program, as it was disturbed by a catastrophic event. The crash may refer to process crash (killed process) or system crash (kernel panic).

**Assert**: A simulation that was unexpectedly terminated due to a simulator failure. If the simulator crashes or reaches a high level condition that it is unable to handle, it raises an assertion to stop the simulation.

**Timeout**: Includes all cases where the simulation did not finish within a certain amount of time (that equals to 4x the fault-free execution time). Simulations are stopped to solve possible deadlock or livelock situations.

The Hardware vulnerability measurement campaigns, on the other hand, have only 2 classes of fault effect classification:

**Masked**: The fault did not reach the program flow until the end of simulation.

**Corrupt**: A mismatch was detected on the program flow compared to the fault-free trace. The mismatch could be on the instruction, operands, data transactions or program order. Performance deviations are ignored in this particular setup.

## C. Benchmarks

In our experiments we use a subset of MiBench benchmarks suite [13]. These are: *FFT, djpeg, stringsearch, smooth, edge, corners, sha, qsort, cjpeg, rijdnael.*

The suite is commonly used in reliability studies [12] [16] [17] [23] [34], since it contains benchmarks from different application domains that also have similar instruction mixes, with SPEC benchmark suite [13]. The validation of our toolset requires complete executions for comparison with AVF estimation. The MiBench suite is the perfect candidate due to the small execution times of the programs which permits a large number of fault injections.

## D. Reliability Measurements

Fig. 7 and Fig. 8 present the fault effects classifications for the AVF measurements and the hardware vulnerability measurements, respectively, for the different GeFIN configurations using the ARM and the x86 ISA. As expected, the Baseline and Early configurations have identical estimations as they only skip 100% masked Inactive epochs and don't suffer any accuracy loss. Early-Fwd configuration shows that the integration of the cache memories states to the checkpoint mechanism of Gem5 significantly improves the accuracy, however we can see that x86 ISA is more affected by the microarchitectural state loss caused by the fast forwarding. Early switching capability seems to fail in some cases for the LSQ. This is due to the fact that store instructions may be forwarded to the memory hierarchy after the instruction commits; the LSQ entry is marked as "ready to write back" and the store is forwarded to the memory hierarchy as long as the cache port is available. If the store is blocked due to other pending stores, the instruction will not stall the back end and will get committed. At that point, the LSQ entry may get detached by the committed instruction, which no longer exists in the pipeline. Switching to emulation mode with this situation raises assertions because the architectural state is not synchronized with the CPU core and fails to drain. This explains the large percentage of assert cases for the LSQ.

Hardware vulnerability measurements on the other hand introduce the fault effect prediction concept that is based on the analysis presented at Section III. The results show that the approach can be potentially used for components with steady behavior patterns, such as register file, LSQ or Instruction cache. In components with unpredictable behavior like the L1 data cache and the L2, results show a large loss of accuracy.

TABLE III summarizes the configuration fault effect measurement deviations in percentile units (vulnerability expressed as summary of the not-masked classes). The most interesting finding is that the switching to emulation mode after a program flow corruption or the 100,000 cycles limit seems to have small impact on the overall vulnerability estimation. This applies especially to the cache memories, where we could not identify trends on the behavior and get close estimations using prediction.

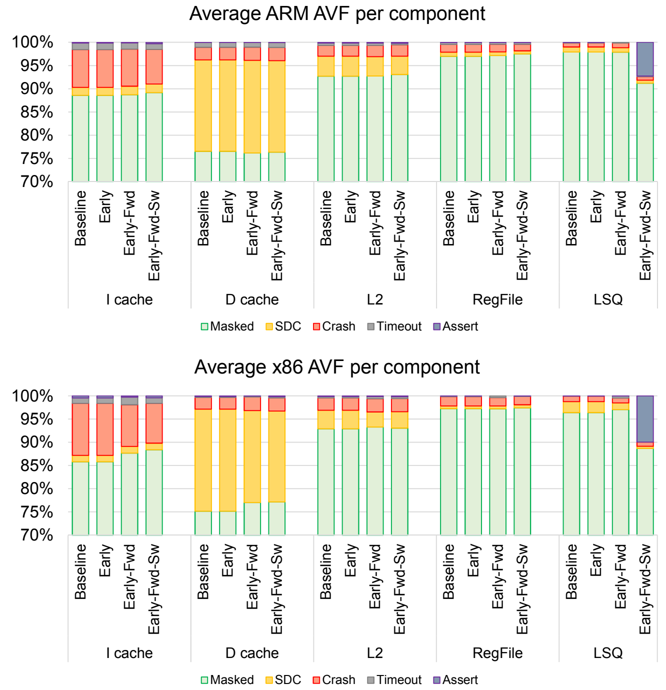


Fig. 7: Fault effects classification for the (top) ARM ISA and (bottom) x86 ISA for AVF measurements for the different modes of operation of the fault injector.

Fig. 8: Fault effects classification for the (top) ARM ISA and (bottom) x86 ISA for the hardware vulnerability measurements for the different modes of operation of the fault injector.

TABLE III: AVERAGE VULNERABILITY DEVIATIONS COMPARED TO THE BASELINE.

| *Comp* | GeFIN Configuration | **AVF deviation** | | **Hardware vulnerability deviation** | |
|---|---|---|---|---|---|
| | | **ARM** | **x86** | **ARM** | **X86** |
| *I$* | ***Early*** | 0 p.u. | 0 p.u. | 0 p.u. | 0 p.u. |
| | ***Early-Fwd*** | 0.16 p.u. | 1.86 p.u. | 0.57 p.u. | 3.58 p.u. |
| | ***Early-Fwd-Sw*** | 0.59 p.u. | 2.58 p.u. | 0.26 p.u. | 3.70 p.u. |
| *D$* | ***Early*** | 0 p.u. | 0 p.u | 0 p.u. | 0 p.u |
| | ***Early-Fwd*** | 0.36 p.u. | 1.86 p.u | 0.34 p.u. | 4.1 p.u |
| | ***Early-Fwd-Sw*** | 0.16 p.u. | 2.02 p.u | 20.8 p.u. | 24.9 p.u |
| *L2* | ***Early*** | 0 p.u. | 0 p.u | 0 p.u. | 0 p.u |
| | ***Early-Fwd*** | 0.03 p.u. | 0.37 p.u | 0.01 p.u. | 0.77 p.u |
| | ***Early-Fwd-Sw*** | 0.38 p.u. | 0.15 p.u | 4.29 p.u. | 5.93 p.u |
| *Reg* | ***Early*** | 0 p.u. | 0 p.u. | 0 p.u. | 0 p.u. |
| | ***Early-Fwd*** | 0.19 p.u. | 0.03 p.u. | 0 p.u. | 0.2 p.u. |
| | ***Early-Fwd-Sw*** | 0.51 p.u. | 0.15 p.u. | 0.19 p.u. | 0.1 p.u. |
| *LSQ* | ***Early*** | 0 p.u. | 0 p.u. | 0 p.u. | 0 p.u. |
| | ***Early-Fwd*** | 0.03 p.u. | 0.64 p.u. | 0.21 p.u. | 1.84 p.u. |
| | ***Early-Fwd-Sw*** | -- | -- | 0.19 p.u. | 1.84 p.u. |

### *E. Fault simulation throughput*

The major outcome of this paper is to successfully accelerate the fault injection based vulnerability estimation. Fig. 9 presents the average simulation time speedup of each configuration for the two types of vulnerability measurements (AVF measurements and hardware vulnerability measurements). Each feature of GeFIN delivers a speedup that depends on the duration of the epoch it targets. Enhanced fast forwarding for instance aims to skip the Pre-fault epochs. Pre-fault epoch occupies 49% of the total simulation time and therefore, the ideal performance benefits of skipping this epoch is up to 2x speedup. This is observable on the speedup graphs, where the modes that have fast forwarding enabled report more than 1.8x speedup (the expected speedup gain with 10 checkpoints that was used in our experiments).

We can observe the simulation performance gains of the newly presented features of GeFIN. Pipeline components benefit the most of the implemented techniques, while the AVF Early-Fwd-Sw mode offers the best balance between accuracy and acceleration, by achieving an average speedup of 4.4x and deviating from the baseline fault injection AVF measurement at less than 0.5 percentile points on average (apart from the LSQ).

On the hardware vulnerability estimation, we can see that Early-Fwd-Sw mode speeds up the simulation at an average of 6x for the register file and the LSQ, which are also the components with the greatest accuracy for the configuration. The L2 and L1 instruction caches obtain a 2.9x speedup for the Early-Fwd mode and at the same time report deviations of less than 0.4 percentile points on the classification.

An observation that is not visible on the presented graphs is that longer benchmarks tend to benefit more from the acceleration features of GeFIN, and this is due to the fact that the instrumentation part of the simulation remains the same.

## V. CONCLUSION AND FUTURE WORK

In this work, we propose several acceleration techniques for fault-injection based reliability estimation at the microarchitecture level. By analyzing the lifetime of a fault injection simulation, we are able to extract behavior patterns that can be used for stopping simulations earlier and predicting the effect of faults. We have implemented all of the described techniques on the GeFIN fault injection framework and our experimental results show that the proposed methods maintain high accuracy in the vulnerability measurements while offering significant simulation speed up for certain components.

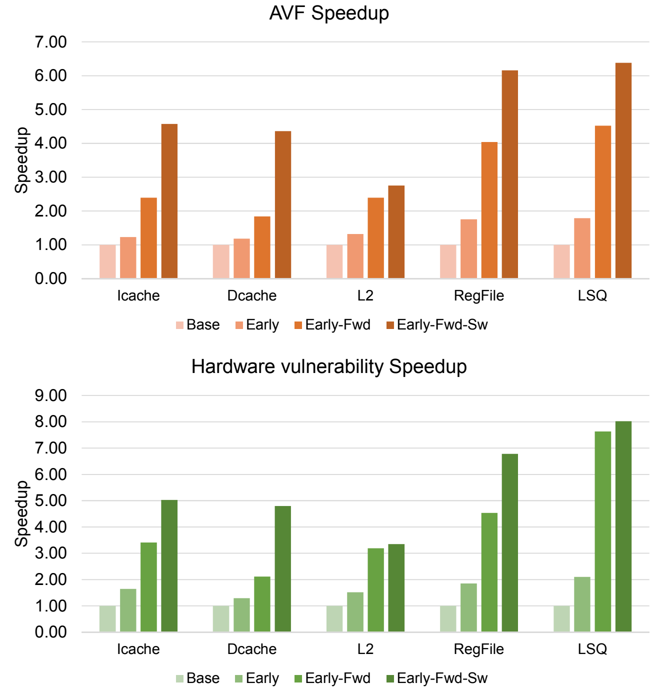


Fig. 9: Fault injeciton campaigns speedups compared to baseline per configuration. (top) for AVF measurements; (bottom) for Hardware vulnerability measurements.

Among the presented techniques, the combination of early stop on overwrite, early stop on program corruption, enhanced fast forwarding and early switching to emulation offers the most efficient tradeoff between accuracy loss and simulation speed up. We have also concluded that some of the techniques are not applicable for certain components, such as the data cache, and further improvements are required. The long Idle epochs that are detected in these components indicate that a profiling process is required to efficiently evade injection on unused resources.


## ACKNOWLEDGMENT

This paper has been supported by the 7th Framework Program of the European Union through the CLERECO Project, under Grant Agreement FP7-611404.


## REFERENCES


[1] G.H.Asadi, V.Sridharan, M.Tahoori, D.Kaeli, "Balancing performance and reliability in the memory hierarchy", ISPASS 2005.

[2] R.C.Baumann, "Soft errors in advanced computer systems", IEEE Design & Test of Comp., vol. 22, no. 3, pp. 258-266, May/June 2005.

[3] N.Binkert et al., "The Gem5 simulator", ACM SIGARCH Computer Arch. News, vol. 39, no. 2, May 2011.

[4] A.Biswas et al., "Computing architectural vulnerability factors for address-based structures", ISCA 2005.

[5] F.A.Bower, D.Hower, M.Yilmaz, D.Sorin, S.Osev, “Applying architectural vulnerability analysis to hard faults in the microprocessor”, SIGMETRICS 2006.

[6] Z.Chishti, A.R.Alameldeen, C.Wilkerson, W.Wu, S.-L.Lu, “Improving cache lifetime reliability at ultra-low voltages”, MICRO 2009.

[7] H.Cho, S. Mirkhani, C.-Y. Cher, J. Abraham, S. Mitra, “Quantitative evaluation of soft error injection techniques for robust system design”, DAC 2013.

[8] C.Constantinescu, “Trends and challenges in VLSI circuit reliability”, IEEE Micro, vol. 23, pp. 14-19, July 2003

[9] S.Feng, S. Gupta. A. Ansari, S. Mahlke, “Shoestring: propabilistic soft error reliability on the cheap”, ASPLOS 2010

[10] N.Foutris, M.Kaliorakis, S.Tselonis, D.Gizopoulos, “Versatile architecture-level fault injection framework for reliability evaluation”, IOLTS 2014.

[11] X.Fu, T.Li, J.Fortes, “Sim-SODA: A unified framework for architectural level software reliability analysis”, Workshop onModeling, Benchmarking and Simulation, 2006.

[12] N.George, C.Elks, B.Johnson, J.Lach, “Transient fault models and AVF estimation revisited”, DSN 2010.

[13] M.R.Guthaus et al., “MiBench: A free, commercially representative embedded benchmark suite”, IWWC 2001.

[14] S.K.S. Hari, S. V. Adve, H. Naemi, P. Ramachandran, “Relyzer: Exploting application-level fault equivalence to analyze applicationresiliency to transient faults”, ASPLOS 2012.

[15] S.K.S. Hari, R. Venkatagiri, S. V. Adve, H. Naemi, “GangES: Gang error simulation for hardware resilience evaluation”, ISCA 2014.

[16] M.Kaliorakis, S.Tselonis, A.Chatzidimitriou, N.Foutris, D.Gizopoulos, “Differential Fault Injection on Microarchitectural Simulators”, IISWC 2015

[17] D.S.Khudia, S.Mahlke, “Harnessing soft computations for low budget fault tolerance”, MICRO 2014.

[18] R.Leveugle, A.Calvez, P.Maistri, P.Vanhauwaert, “Statistical fault injection: Quantified error and confidence”, DATE 2009.

[19] X.Li, S.V.Adve, P.Bose, J.A.Rivers, “Architecture-level soft error analysis: Examining the limits of common assumptions”, DSN 2007.

[20] Y.Luo et al., “Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneousreliability memory”, DSN 2014.

[21] S.S.Mukherjee, C.T.Weaver, J.Emer, S.K.Reinhardt, T.Austin, “A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor, MICRO 2003.

[22] A.A.Nair, S.Eyerman, L.Eeckhout, L.K.John, “A first-order mechanistic model for architectural vulnerability factor”, ISCA 2012.

[23] A.A.Nair, L.K.John, L.Eeckhout, “AVF stressmark: Towards an automated methodology for bounding the worst-case vulnerability to soft errors”, MICRO 2010.

[24] S.Nassif, N.Mehta, Y.Cao, “A resilience roadmap”, DATE 2010.

[25] S.Pan, Y.Hu, X.Li “IVF: Characterizing the vulnerability of microprocessor structures to intermittent faults”, IEEE Transactions on VLSI Systems, vol. 20, no. 5, pp. 777-790, May 2012.

[26] K.Parasyris, G.Tziantzoulis, C.Antonopoulos, N.Bellas, “GemFI: A fault injection tool for studying the behavior of applications on unreliable substrates”, DSN 2014.

[27] T.Sherwood, E.Perelman, G.Hamerly, B.Calder, “Automatically characterizing large scale program behavior”, ASPLOS 2002.

[28] V.Sridharan, D.R.Kaeli, “Eliminating microarchitectural dependency from architectural vulnerability”, IEEE International Symposium on High Performance Computer Architecture (HPCA-15), 2009.

[29] V.Sridharan, D.R.Kaeli, “Quantifying software vulnerability”, Workshop on Radiation effects and fault tolerance in nanometer technologies (WREFT), 2008

[30] V.Sridharan, D.R.Kaeli, “Using hardware vulnerability factors to enhance AVF analysis”, ISCA 2010.

[31] J.Suh, M.Annavaram, M.Dubois, “MACAU: A Markov model for reliability evaluations of caches under single-bit and multi-bit upsets”, HPCA 2012.

[32] N.J.Wang, A.Mahesri, S.J.Patel, “Examining ACE analysis reliability estimates using fault injection”, ISCA 2007.

[33] G.Yalcin, O.S.Unsal, A.Cristal, M.Valero, “FIMSIM: A fault injection infrastructure for microarchitectural simulators”, ICCD 2011.

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