2015 IEEE International Symposium on Workload Characterization

Differential Fault Injection on Microarchitectural Simulators

Manolis Kaliorakis, Sotiris Tselonis, Athanasios Chatzidimitriou, Nikos Foutris, Dimitris Gizopoulos

Department of Informatics & Telecommunications, University of Athens, Greece
{manoliskal, tseloniss, achatz, nfoutris, dgizop}@di.uoa.gr

***Abstract*—Fault injection on microarchitectural structures modeled in performance simulators is an effective method for the assessment of microprocessors reliability in early design stages. Compared to lower level fault injection approaches it is orders of magnitude faster and allows execution of large portions of workloads to study the effect of faults to the final program output. Moreover, for many important hardware components it delivers accurate reliability estimates compared to analytical methods which are fast but are known to significantly over-estimate a structure's vulnerability to faults.**

This paper investigates the effectiveness of microarchitectural fault injection for x86 and ARM microprocessors in a *differential* way: by developing and comparing two fault injection frameworks on top of the most popular performance simulators, MARSS and Gem5. The injectors, called MaFIN and GeFIN (for MARSS-based and Gem5-based Fault Injector, respectively), are designed for accurate reliability studies and deliver several contributions among which: (a) reliability studies for a wide set of fault models on major hardware structures (for different sizes and organizations), (b) study on the reliability sensitivity of microarchitecture structures for the same ISA (x86) implemented on two different simulators, (c) study on the reliability of workloads and microarchitectures for the two most popular ISAs (ARM vs. x86).

For the workloads of our experimental study we analyze the common trends observed in the CPU reliability assessments produced by the two injectors. Also, we explain the sources of difference when diverging reliability reports are provided by the tools. Both the common trends and the differences are attributed to fundamental implementations of the simulators and are supported by benchmarks runtime statistics. The insights of our analysis can guide the selection of the most appropriate tool for hardware reliability studies (and thus decision-making for protection mechanisms) on certain microarchitectures for the popular x86 and ARM ISAs.

Keywords-reliability evaluation, fault injection, micro-architectural simulators, microprocessors

I. INTRODUCTION

The reliable operation of modern and forthcoming computing systems can be affected by *transient faults* (soft errors), *intermittent faults*, and *permanent* (hard) *faults* [2] [9] [17]. Hardware faults can be caused by external factors such as radiation or are due to latent manufacturing defects, device degradation, or certain modes of operation such as very low voltage operation [2] [7] [9] [30]. Several metrics have been proposed for the assessment of reliability; for example the Architectural Vulnerability Factor (AVF) [28] quantifies the probability of a transient fault in a hardware component to produce a program-visible error accounting for both the hardware and the software masking effects. Similarly, vulnerability factors for intermittent faults [31] (IVF) and permanent faults [6] (H-AVF) have been defined.

Early assessment of the expected reliability of a computing system (or equivalently its resiliency to hardware faults) is an important task which steers design decisions related to the required mechanisms for the detection and diagnosis of hardware faults and the recovery of the system from their effects. Such fault tolerance mechanisms always impose area, power and performance overheads. Straightforward guard-banding of the system with inaccurate knowledge of the effect of hardware faults can easily make the costs of protection against hardware faults excessive. For example, typical memory error detection and correction techniques can have a cost (in terms of added memory capacity) which ranges from 1% to 125% depending on the detection and correction capabilities of each technique [24]. Clearly, the selection of the most appropriate protection techniques depends on the required reliability levels and studies of its inherent resiliency to hardware faults.

Tolerance mechanisms against any fault model must be decided as early as possible to avoid costly re-design cycles for late integration of such mechanisms. However, early decisions on the protection mechanisms are hard to make because during the early stages of a system design important parameters are unknown: hardware components sizes and architectures, workload. It is widely recognized that microarchitecture simulators, apart from their importance for performance studies, offer an opportunity for an effective combination of *early* and *accurate* reliability estimations:

- They are available in *early* design phases and important parameters of the major hardware structures can be easily configured.
- They are significantly *faster* than simulators at more detailed levels of abstraction (RTL, gate-level) and thus allow studies on large intervals of software execution.
- They *accurately* model important array-based microarchitecture components: storage arrays which occupy the majority of a chip's area and thus largely determine vulnerability to faults. For instance, on-chip caches, register files, buffers, queues.

A set of approaches that utilize performance simulators for reliability evaluation is based on probabilistic models and ACE-based (Architectural Correct Execution) analysis to determine the AVF of hardware structures [5] [13] [28] [29] [43] [51]. Other approaches also use performance measurements [10] [44] for online AVF estimation, while others try to separate the masking effects that hardware and

978-1-5090-0088-3/15 $31.00 © 2015 IEEE
DOI 10.1109/IISWC.2015.28

software can have on hardware faults [38] [40] [41]. All these approaches are very fast (a single or few runs of a benchmark are needed to feed the models) but require significant modifications of the simulator as they are based on tracking data flow through the microarchitecture. Despite their speed, the recognized drawback of these approaches is that they over-estimate the vulnerability of microprocessor structures [14] [23] [45]; for example, [14] reports a 7x AVF over-estimation and [45] reports that even a refined ACE-based analysis (which requires even more elaborate modifications of the microprocessor simulator model) leads to up to 3x over-estimation. This can lead to decisions for expensive but not justifiable protection mechanisms.

On the other hand, *fault injection* frameworks that utilize microarchitectural simulators much closer resemble the actual hardware and software behavior in the presence of faults [12] [14] [32] [48]. Fault injection approaches require much simpler modifications of the simulators but require larger simulation time for the fault injection experiments. However, when statistically significant numbers of fault injections are performed [20], fault injection delivers very accurate reports on the faulty behavior of hardware components.

In this paper, we investigate the limits of microarchitecture level fault injection for x86 and ARM ISAs conducting a *differential* analysis on two comprehensive fault injector tools supporting the same fault models and running the same workloads. Such a differential analysis can bring insights about the sensitivity of the vulnerability of hardware structures and workloads to the underlying microarchitecture as well as the ISA of the microprocessor. It can also identify common trends and diverging reliability reports in the two tools which can lead to informed design decisions for error protection.

Our differential fault injection framework can serve many different studies in this context. We inject hardware faults on actual microarchitecture structures (all storage arrays: caches, register files, buffers, queues – not only on architecturally visible points) to better assist design decisions for error protection of individual components.

Our microarchitecture-level fault injection tools, called MaFIN and GeFIN (for MARSS-based and Gem5-based Fault Injector, respectively), are built on the two most popular microarchitectural simulators (MARSS [33] and Gem5 [3]) and the two popular ISAs (x86 and ARM). Both injectors have been developed modularly using exactly the same principles and employ the check-pointing features of the simulators to ensure that faults affect only the execution of the benchmark being studied as well as to speed up the injection campaigns.

Both MaFIN and GeFIN consist of three modules: a *fault masks generator*, an *injection campaign controller* and a *parser* of the logged information. The tools allow studies on the full range of fault models: transient, intermittent and permanent, as well as studies with multiple faults injected in: (i) different bits of the same entry of a hardware structure, (ii) different entries of a structure, (iii) different hardware structures simultaneously, (iv) all combinations of the above.

Table I summarizes the state in microarchitectural fault injectors and puts the new contributions of this paper in this context. The two new microarchitectural fault injectors built for the needs of our differential study cover several important missing aspects of the research area.

We keep a balance in the content of the paper between: (a) the essential description of the microarchitecture-level fault injector tools (realization of the injection, modifications of the simulators, their features, the supported fault models, etc.) and (b) the presentation, analysis and explanation of the experimental results to identify the sources of common trends and diverging reliability reports between the tools. Sections II and III focus on the former while Section IV focuses on the latter. Section V discusses related work and Section VI concludes the paper.

TABLE I. STATE-OF-THE-ART AND CONTRIBUTIONS OF THIS PAPER IN FAULT INJECTION TECHNIQUES ON MICROARCHITECTURAL SIMULATORS

Aspect	*State-of-the-art*	*This work*
Injection framework that targets all major microarchitecture structures	**None**[1]	Both MaFIN and GeFIN: all major structures
Comparison between ISAs (x86 vs. ARM)	**None**	GeFIN (x86 vs. ARM ISA)
Comparison between Out-of-Order microarchitectures	**None**	MaFIN and GeFIN
Comparison between simulators for same ISA	**None**	MaFIN and GeFIN (for x86 ISA)
Full system fault injection	[32]: Gem5; [48]: M5; [21] [22]: GEMS	Both MaFIN and GeFIN are full system injectors
New microarchitectural structures added	**None**	MaFIN
Transient, intermittent, permanent fault models	[48] (not all hardware structures)	MaFIN and GeFIN: all fault models

II. MICROARCHITECTURAL FAULT INJECTION

The objective of this work is the study and the analysis of workloads reliability in the presence of hardware faults on top of two different (thus diverse), configurable, microarchitecture-level full-system fault injectors for x86 and ARM ISAs. By setting their configurations and running benchmarks of interest several reliability studies for hardware components can be performed. This comparative study can reveal important insights about microarchitectural fault injection, among which:

- What are the characteristics of a microarchitectural simulator that make it more suitable as a substrate for fault injection studies?
- How much sensitive is the vulnerability of hardware structures to the ISA as well as the microarchitecture (simulator model or hardware structures configurations) for a given workload?

[1] [14]: integer register file and ROB only; [48]: no injections supported in any cache level.

Major decisions towards the objective of this paper are the selection of the microarchitectural simulators and configurations. We discuss these decisions.

A. Simulators Selection

We have considered a number of publicly available full system simulators. A recent study [16] on the sources of modeling errors in full system simulators summarizes the publicly available tools and their advantages: Flexus [47], Gem5 [3], GEMS [27], MARSS [33], OVPsim [18], PTLsim [49], Simics [25]. Among these full-system simulators MARSS [33] and Gem5 [3] are: *cycle-accurate* (thus can allow per cycle granularity of fault injections at any modeled hardware component), *publicly available*, and *regularly maintained* today by their developers. By themselves, these properties can justify selecting MARSS and Gem5 for our reliability studies. Moreover, the two simulators best serve our purposes because:

They are widely adopted by the computer architecture community. Both simulators are recent and very popular. Their increased popularity is mainly due to their accurate support of important ISAs, their detailed and configurable model of the memory system [42] and check-pointing support.

Their combination supports differential reliability studies. The combination of MARSS and Gem5 supports the purposes of our work – reliability studies on different ISAs and reliability studies on the same ISA on different simulators. In particular:

- Both MARSS and Gem5 support the x86 ISA and thus facilitate comparison of microarchitectural fault injections in the hardware components of an x86 microprocessor.
- Gem5 supports several ISAs; ARM and x86 are among the best supported and thus a comparative study of them can be performed.
- Both MARSS and Gem5 have a fully configurable model (pipeline depths and widths, structures sizes and organizations, etc.)
- MARSS models both a high-performance OoO pipeline and a simple in-order (Atom-like) pipeline; a reliability assessment study between these two models can be implemented (this paper focuses on the OoO model to compare with the corresponding one of Gem5).

An important difference between MARSS and Gem5 is that they require *different development* efforts to support fault injection at the microarchitecture level. Gem5 already includes all key microarchitecture components which model hardware arrays on which faults of any duration and severity can be injected. MARSS, on the other hand, does not contain important arrays needed for fault injection: data/instruction arrays of caches at all levels. This dual effort delivers a framework for comparative studies on important hardware components such as caches on MARSS. Details about the modifications of the original versions of the simulators are described along with the integration of the fault injection features in Section III.

B. Simulators Configurations

The three different configurations of MARSS and Gem5 on which we performed our experimental study and analysis (Section IV) are summarized in Table II. MARSS simulates x86 ISA while the x86 and ARM ISAs of Gem5 have been used[2].

Both MaFIN and GeFIN injectors can be easily modified for other values of the parameters shown in Table II. Our main focus in setting the parameters was to keep the sizes and organizations of the hardware structures the same (or as close as possible) in the two simulators. Section III describes the modifications of the simulators and the additional components we added on them. For any parameters not shown below the default values of the simulators were used.

TABLE II. SIMULATORS CONFIGURATIONS

Parameter	Simulator/ISA		
	MARSS/x86	**Gem5/x86**	**Gem5/ARM**
Pipeline	OoO	OoO	OoO
Physical register file	256 int; 256 FP; 16 store; 24 branch	256 int; 128 FP	256 int; 128 FP
Issue Queue entries	32	32	32
Load/Store Queue entries	32 (unified)	16 (load)/ 16 (store)	16 (load)/ 16 (store)
ROB entries	64	40	40
Functional units	2 int ALUs; 2 FP ALUs; 4 AGUs	6 int ALUs; 2 complex int ALUs; 4 FP ALUs, 2 FP mul/div, 4 SIMD	2 int ALUs; 1 complex int ALUs; 2 FP & SIMD
L1 Instruction Cache	32KB, 64B line, 128 sets, 4-way,write back	32KB, 64B line, 128 sets, 4-way, write back	32KB, 64B line, 128 sets, 4-way,write back
L1 Data Cache	32KB, 64B line, 128 sets, 4-ways, write back	32KB, 64B line, 128 sets, 4-ways, write back	32KB, 64B line, 128 sets, 4-ways, write back
L2 Cache	1MB, 64B line, 1024 sets, 16-way, write back	1 MB, 64B line, 1024 sets, 16-way, write back	1 MB, 64B line, 1024 sets, 16-way, write back
Branch Predictor	Tournament predictor	Tournament predictor	Tournament predictor
Branch Target Buffer	direct branches BTB (4-way, 1K entries), indirect branches BTB (4-way, 512 entries)	conditional and unconditional branches BTB (direct-mapped, 2K entries)	conditional and unconditional branches BTB (direct-mapped, 2K entries)
RAS	16 entries	16 entries	16 entries

III. MAFIN AND GEFIN FEATURES AND IMPLEMENTATION

In this section we discuss in detail the fault injection functionalities of MaFIN and GeFIN tools as well as their implementation and main modules.

A. Fault Injectors Features

The designs of MaFIN and GeFIN injectors rely on the same principles and our intention was to extend the original

[2] All ISAs that support full system simulation on Gem5 can be also employed in future versions of the tool.

versions of the MARSS and Gem5 simulators in order to support the same injection capabilities and features. The following description applies to both MaFIN and GeFIN injectors.

Fault models

Both MaFIN and GeFIN model exactly the same fault types on microarchitectural array components: transient, intermittent and permanent faults as well as their combinations. These three types of fault models allow a wide analysis of the effect of different factors that affect reliability: fabrication defects, environmental conditions, early-life failures, device degradation and voltage scaling. Table III describes the three basic single bit fault models.

TABLE III. FAULT MODELS

Fault model	*Description*
transient	a storage element's bit value is flipped in a clock cycle of the program execution; the bit position and the clock cycle can be set arbitrarily (randomly or directed)[3]
intermittent	a storage element's bit value is set to '0' or to '1' starting at a clock cycle and for an arbitrary number of clock cycles; the bit position, the start time and the duration of the fault can be set arbitrarily (randomly or directed)
permanent	a storage element's bit value is permanently set to '0' or to '1'; the bit position can be set arbitrarily (randomly or directed)

Moreover, both MaFIN and GeFIN support fault injection experiments for *multiple faults* in many different combinations to match both the temporal and the spatial behavior of faults in hardware structures. Such combinations can include injection of (a) multiple faults of any type and any duration in a single structure, (b) multiple faults on different structures.

Obviously, the type, the multiplicity and the locations of the faults used in a certain injection campaign depend on the study that a user of MaFIN and GeFIN wishes to perform. In this differential study we used only the single bit flip model, but the tools also support permanent, intermittent and multibit fault studies.

Fault effect classification

MaFIN and GeFIN injectors classify the outcomes of each fault injection simulation based on the impact of the fault on the simulated system. The fault classification is fully configurable and a user of the injector can modify the classes of the fault effects by changing the parser of the injection logging information (see the next subsection for the operation of the parser and examples of classification options). In this paper, we present the fault effects classification using the following six classes. These represent typical classes (and corresponding terminology) used in the reliability literature.

Masked: fault injection runs in which the fault does not affect the execution of the application (which is executed to its end). The result of an injection with a masked fault is identical to that of a fault-free simulation (both the output of the application and any exceptions generated during execution).

Silent Data Corruption (SDC): fault injection runs for which the final output of the program that is written to an output file is corrupted (differs from the output of the fault-free execution) and no other indication of the fault has been recorded (an abnormal event such as an exception, etc.).

Detected Unrecoverable Error (DUE): includes cases in which the simulated process completes successfully, but with indications of errors. The baseline microprocessor models do not include any error detection or protection mechanisms and therefore, the only indication of an error is the raising of ISA exceptions. Typically, reliability reports in the literature divide DUEs in two sub-categories: false DUE (the output is correct despite the error indication) and true DUE (output is corrupted). In our experimental results section we don't show these two different DUE sub-classes but of course the parser of both fault injectors can be configured to calculate them separately.

Timeout: includes all of the cases that lead to either a Deadlock or a Livelock. A Deadlock describes the condition in which the program flow has been trapped (due to the injected fault) and can't commit any further instructions. A Livelock, on the other hand, describes a situation where the program flow has been redirected and continues the execution of instructions on random code areas (again due to the fault). In order to monitor these cases, a configurable execution timeout limit is used. In our experimental results, the limit is three times the fault-free execution time of each benchmark.

Crash: includes any case that results in an unrecoverable situation and stops the simulated program. Crashes involve all three levels of the simulation, including a *process crash*, where the simulated program was abnormally terminated, a *system crash*, where the simulated full-system was unable to recover (typical cases of *kernel panic*) as well as a *simulator crash*, where the simulator process itself was abnormally terminated.

Assert: includes all cases where the simulator reached, due to injected fault, a (high level) condition which was unable to handle and an assertion was raised stopping the simulation.

In our experimental results (Section IV) we use the term *vulnerability* to refer any abnormal behavior due to a fault, i.e. the sum of the non-masked classes.

B. Fault Injectors Implementation

The major objectives of the two injectors were:

1) *Accuracy* of delivered reliability reports. Towards this objective we run applications/benchmarks to completion unless a fault is guaranteed masked. This execution to the end ensures the final program effect of faults is captured.

[3] We don't consider transient faults in combinational logic in this work because microarchitecture simulators don't model such logic accurately; however, their effects would propagate to storage elements and thus can be also implicitly studied with our tools.

2) *Speed* of the fault injections campaigns (applies to transient faults). Apart from the straightforward employment of several workstations to run experiments on, we *optimize* the injectors so that an injection run is stopped immediately in cases where: (i) a fault is injected in an invalid/unused entry of a structure, (ii) a faulty entry is over-written before ever read. For all benchmarks and all components these optimizations lead to 30%-70% speedup of each individual run and thus very large savings of the injection campaign time.

3) *High configurability* of the injectors. See all details about the features of the injectors in the following paragraphs.

Both the MaFIN and GeFIN injectors are built on three main modules which form the backbone of any fault injection campaign run on them. Fig. 1 visualizes the flow of operation of the two injectors.

In the first step, the *Fault Mask Generator* module produces the *fault masks* that are used during the injection campaign. This is a one-step process for each combination of hardware structure and benchmark. The Fault Mask Generator can produce (by user defined parameters) a random set of fault masks for any type of fault (transient, intermittent, permanent) for the entire simulation time of the benchmark.

A fault mask contains information about: (i) the processor core where the fault is going to be injected (can be used in a multicore architecture), (ii) the microarchitecture structure on which the fault will be injected, (iii) the exact bit position of the injection, (iv) the exact simulation cycle or exact instruction on which injection happens (for transient or intermittent), (v) the type of fault, and finally (vi) the population of faults (single or multiple). All the generated *fault masks* are stored in a "masks repository" from which the Injection Campaign Controller picks fault masks to apply.

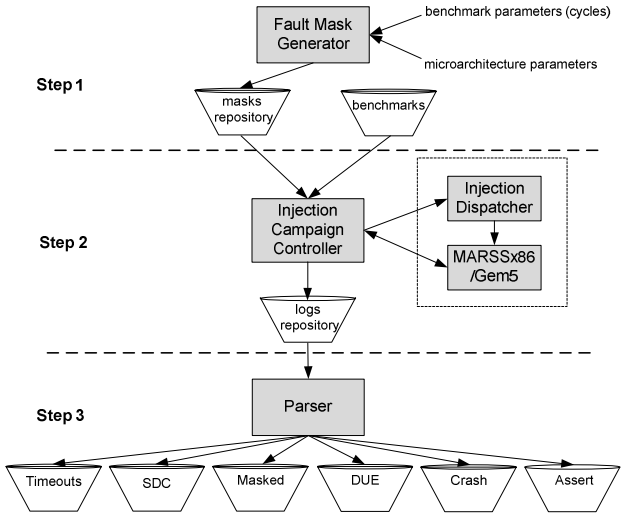

Figure 1. MaFIN and GeFIN injection frameworks

Provided the "mask repository", the actual fault injection campaign can begin. The *Injection Campaign Controller* reads the masks from the repository and sends injection requests to the *Injector Dispatcher* which is the module that directly communicates with the MARSS or Gem5 simulator, respectively. The interface between the Injection Campaign Controller and the individual Injection Dispatcher contains the transfer of user defined parameters concerning the injection to the microarchitectural simulators and the transfer of the results of the fault injection experiments from the microarchitectural simulator back to the Injection Campaign Controller. The last task of the Injection Campaign Controller is to store the results of the injection in a "logs repository" which contains all log files for further processing by the Parser.

The third and last step of the fault injection campaign is the processing of the injection results and the generation of the fault effects classification. The processing of the fault injection results is performed by the use of a *Parser.* The Parser is an easily reconfigurable script that classifies the faults into the six final categories described in previous sub-section: Masked, SDC, DUE, Timeout, Crash, and Assert. The classification results can be easily modified through small changes of the Parser code according to the user's needs as the input of Parser for an alternative classification is not changed and is already stored into the log files repository (no new fault injection campaign is required). For example, a more course-grain classification can be used just separating "Masked" from "Non-Masked" behavior. On the other hand, a more fine-grain classification may break down the DUE category in false-DUE and true-DUE (a usual separation in the reliability literature). Moreover, the user could move the results from the Simulator Crash subcategory to the Assert category to group together faulty behaviors attributed to simulator malfunctions due to the injected faults.

C. *Extensions to MARSS and Gem5*

Microarchitectural simulators are developed for performance measurements of the simulated model and their main objective is to save simulation time without modeling details that are not necessary for performance assessments. As a result, performance simulators may lack certain functionality necessary to perform accurate fault injection experiments. For example, the functional and the control logic components are not implemented in a way that resembles actual hardware structures. Therefore, injectors like MaFIN and GeFIN focus on reliability studies in hardware structures which are modeled as arrays in a performance simulator and thus the effect of faults on them can be accurately measured. The injection of transient, intermittent or permanent fault on a modeled storage bit of a microarchitectural simulator is largely equivalent to injecting it on the actual hardware.

Unfortunately, some simulators do not model data arrays of caches (and other structures such as queues, buffers); MARSS is such a simulator. It models the control information of cache memories (tags and control bits) but only keeps the actual data and instructions at the main memory model of the simulation. Without the

implementation of the actual arrays for the data and the instructions on caches, fault injection is not feasible. Our work addresses this issue by implementing the data array extension on MARSS to be able to compare the x86 reliability studies on both injectors (Gem5 already includes the caches data arrays). This modification of MARSS introduced an approximate ~40% throughput degradation which depends on the memory intensiveness of a program.

The development of MaFIN and GeFIN went through the following major tasks:

- Identification of existing structures; integration of the fault injector on these structures.
- Modification of structures that lack of accuracy to perform a fault injection study (missing bit arrays); integration of the fault injector on these structures.
- Enhancement of the x86 model of MARSS with new components (performance related) to fully resemble a modern design; integration of the fault injector on these new structures.

Table IV summarizes all enhancements made on MARSS and Gem5 for accurate measurements of the reliability of the hardware structures of x86 and ARM-based architectures.

TABLE IV. MAFIN AND GEFIN ENHANCEMENTS

Components	Simulator/ISA	
	MaFIN-x86	***GeFIN-x86 and GeFIN-ARM***
Existing	Load/Store Queue Issue Queue Integer Register File FP Register File Caches – Tag Data TLB – Valid, Tag Instr. TLB – Valid, Tag Branch Target Buffer – Uncond. indirect branches	Load/Store Queue Issue Queue Integer Register File FP Register File Caches – Tag Caches – Data Data TLB – Valid, Tag Instr. TLB – Valid, Tag Branch Target Buffer- Uncond./Cond. direct branches
Modified	L1D cache – Data arrays L1I cache – Instruction arrays L2 cache – Data arrays L1I cache – Valid bit L1D cache – Valid bit L2 cache – Valid bit Branch Target Buffer – Uncond./Cond. direct branches	Accurate reliability modeling of associative caches structure (replacement algorithm)
New	Prefetcher in L1D cache Prefetcher in L1I cache	

The two fault injectors can perform injections on *all* the components of Table IV. Both MaFIN and GeFIN already cover very important hardware structures consisting of large arrays of storage bits, which are also the most vulnerable components of modern processors. All these array structures are customizable rendering MaFIN and GeFIN very useful for several different reliability estimation studies.

IV. EXPERIMENTAL RESULTS

Although the number of studies that can be performed on MaFIN and GeFIN is very large[4] and can be the subject of future research, we present a comprehensive set of experimental results in this paper to identify and analyze consistent reliability trends between microarchitectures and workloads and also to explain reported divergences between the two tools.

In this section we detail the context of the experimental analysis, present the results and analyze/explain them. We first discuss the fault sampling method we used and the benchmarks employed in the study. We then provide the faulty behavior characterization results (using the classes described in Section III) and analyze them in detail to highlight common trends as well as to explain and root cause significant differences.

A. Fault Sampling

Any fault sampling approach can be applied in the injectors. In our experimental results we used statistical fault sampling as described in [20]. Given: (a) the number of bits of an array-based hardware structure, (b) the number of execution cycles of a benchmark, and (c) the required confidence and error margin of the sampling the formula of [20] delivers the number of required fault injection runs.

The major parameters in the fault sampling described in [20] are the confidence and the error margin. For a 99% confidence and a 3% error margin, for all the hardware structures and all benchmarks of our study the number of required fault injections is 1843. We round this number up by injecting 2000 faults in each structure/benchmark combination (this number of injections correspond to 2.88% error margin). The accuracy of any statistical fault injection campaign can be traded off with the time required to perform the campaign; this is the case also in our injectors. For example, if the error margin of the sampling is increased from 3% to 5% then the number of required injections per hardware structure is only 663 instead of 1843 which leads to significantly smaller (by approximately 3 times) campaign execution time.

B. Benchmarks

We utilize MaFIN and GeFIN frameworks to classify the behavior of 10 benchmarks in the presence of transient faults. All benchmarks are simulated to their completion (unless "safe" early-stop is decided at run time – see previous section) to guarantee full accuracy of the reported classification.

The 10 benchmarks we use are from the MiBench suite [15] (*djpeg*, *search*, *smooth*, *edge*, *corner*, *sha*, *fft*, *qsort*, *cjpeg*, *caes*). MiBench benchmarks suite consists of programs from different application domains, and are very similar in their instruction mixes and instruction throughput with SPEC benchmarks [15]. Their shorter execution times compared to SPEC (standard benchmarks for performance

[4] Studies per hardware structure; per benchmark; per simulator; for different sizes and organizations of the hardware structures; for different input data sets of the benchmarks, etc.

studies) make them very suitable for fault injection and reliability studies and for this reason they have been extensively used in such a context [14], [50], [52], [53]. Although we report results in this paper for the MiBench benchmarks, both MaFIN and GeFIN can be used in fault injection campaigns using SPEC2006 benchmarks (or any other benchmark). Both simulators support check-pointing and thus targeted runs on SimPoint samples [39] of the SPEC2006 benchmarks can be executed.

C. *Reliability Characterization Results Analysis*

In this subsection, we report the results of an extensive characterization study on hardware components on MaFIN and GeFIN. In particular, the results show reliability characterization of the following hardware structures (faults can be injected in all structures listed in Table IV; we selected to report on the following ones because of their importance in the CPU and their large sizes compared to other components):

- Integer physical register file (Fig. 2)
- L1D cache (Fig. 3)
- L1I cache (Fig. 4)
- L2 cache (Fig. 5)
- Load/Store Queue (Fig. 6)

For each component the sizes and configurations are the ones shown on Table II. As discussed previously, we inject 2000 transient faults randomly in each structure and for each benchmark. In total 300,000 fault injections have been performed (5 components x 10 benchmarks x 3 tools x 2000 injections = 300,000 injections).

Roughly, the complete fault injection campaigns reported in the paper took approximately 1 month: 2000 injections performed in 5 different hardware structures, for 10 different benchmarks, in 3 different setups – MaFIN-x86, GeFIN-x86 and GeFIN-ARM; we employed 10 different workstations providing about 100 threads that ran injections in parallel.

Each graph shows for a particular component the faulty behavior classification (using the classes in Section III) for each of the 10 benchmarks and on the average. For each benchmark the graphs show three stacked bars (each bar corresponds to a fault injection campaign): one for the execution on the MaFIN-x86 injector (M-x86 bar), one on the GeFIN-x86 configuration (G-x86) and one on the GeFIN-ARM configuration (G-ARM). For the average case, the same three bars are shown at the rightmost end of each diagram.

The average vulnerability[5] reports at the rightmost bars of each diagram reveal the following:

- The largest average case vulnerability differences are observed between the two x86-based configurations (MaFIN-x86 and GeFIN-x86): 7.20 percentile points in the L1D cache, 3.61 percentile points in the L1I cache, and 1.36 percentile points in the L2 cache.
- On the contrary, the vulnerability differences between the two ISAs (x86 and ARM) on GeFIN are much smaller in all components. In the L1D and L2 cache the differences between GeFIN-x86 and GeFIN-ARM configurations are only 0.55 and 0.13 percentile points respectively, while in the L1I cache the average difference is 2.03 percentile points (x86 being more vulnerable than ARM).

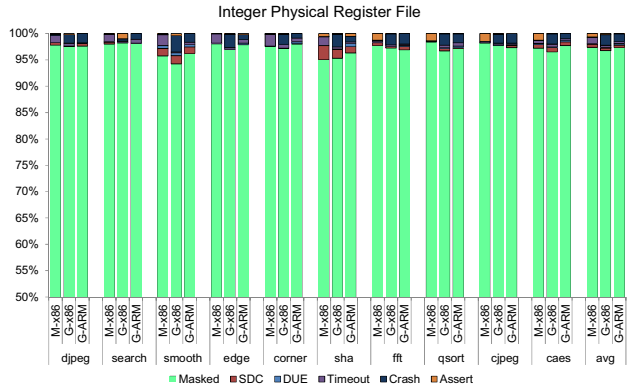

Figure 2. Faulty behavior classification for the integer physical register file.

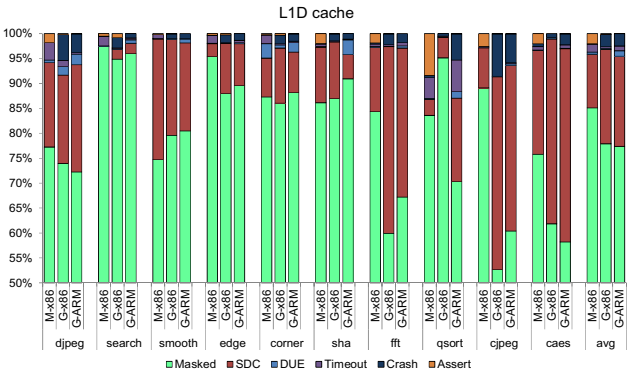

Figure 3. Faulty behavior classification for L1D cache (data arrays).

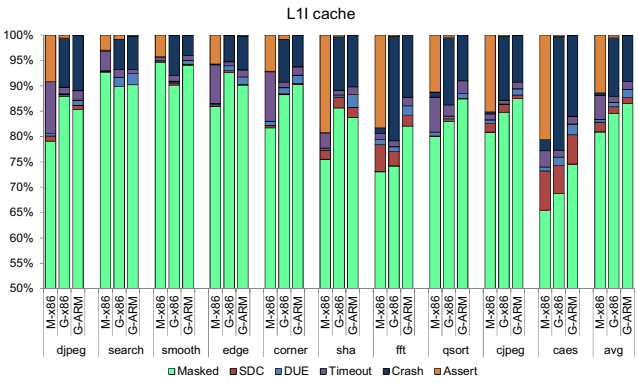

Figure 4. Faulty behavior classification for L1I cache (instruction arrays).

[5] As mentioned earlier we use the term *vulnerability* to refer to the sum of all non-masked behaviors. AVF can be also used in our context: the probability that a transient fault in a structure's bit leads to any erroneous behavior (i.e. not masked).

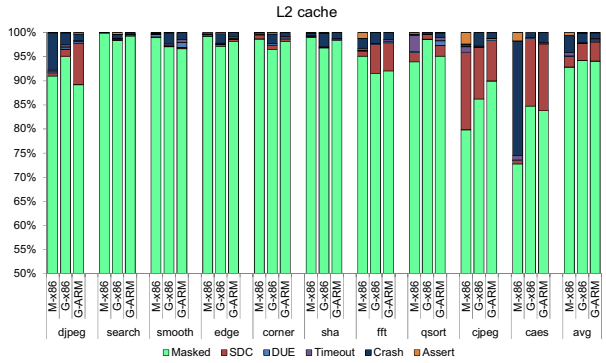

Figure 5. Faulty behavior classification for L2 cache (data arrays).

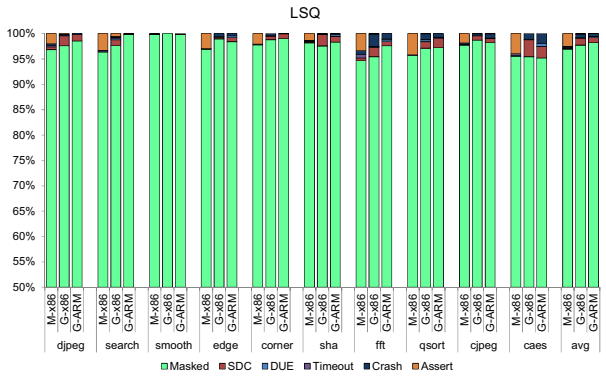

Figure 6. Faulty behavior classification for Load/Store Queue (data field).

We analyze in more detail the results of the classification shown in the diagrams both on a per-component basis to identify consistent trends and on a per-benchmark basis to interpret diverging behaviors. We discuss potential (microarchitecture or ISA related) reasons that explain the differences between the two tools providing execution statistics for the benchmarks of our study.

Integer Register File and LSQ

The Integer Register File (Fig. 2) and the LSQ (Fig. 6) are the least vulnerable components in all cases (benchmark, ISA and microarchitecture configuration). The *vulnerability* (sum of all non-masked classes) of the Register File and the LSQ for each individual benchmark and on the average across all benchmarks is almost always less than 3% for all three configurations. This is a consistent behavior that is also compatible to previous literature reports. The two components hold data of relatively short lifetime which explains the small vulnerability to transient faults.

- **Remark 1** – There is a consistent small difference of ~1 percentile point between the MaFIN and GeFIN report for the LSQ vulnerability (LSQ in MaFIN is always slightly more vulnerable than the GeFIN's LSQ). The reason for this slight difference is that MARSS implements a unified queue for loads and stores while Gem5 implements different queues and only the store queue holds data. Therefore, our injections on GeFIN's LSQ affect only stores while in MaFIN both queues are affected by faults.
- **Remark 2** – Both the Integer Register File and the LSQ have mixed faulty behaviors in the non-masked classes. Faults in both components in most cases can lead to any of the five non-masked faulty behaviors (SDC, DUE, Timeout, Crash, and Assert). The exact numbers in each class of course depend on the benchmark.

First-Level Caches (L1D, L1I)

The first-level cache memories (L1D cache in Fig. 3 and L1I cache in Fig. 4) are the most vulnerable components in all cases (benchmark, ISA and microarchitecture configuration).

The L1D cache vulnerability varies significantly among benchmarks and between ISAs and microarchitectures. Its vulnerability can be as low as 2.5% (*search* benchmark in the MaFIN-x86 setup) and as high 47.3% (*cjpeg* benchmark in the GeFIN-x86 setup). On average across benchmarks the L1D cache vulnerability is less than 15% in MaFIN-x86 while in both ISAs of GeFIN (GeFIN-x86 and GeFIN-ARM) it is more than 22%. The general trend in most (but not all) individual benchmarks) is that MaFIN reports a less vulnerable L1D cache than GeFIN.

- **Remark 3** – The significant ~7 percentile point difference between MaFIN and GeFIN vulnerability reports on the L1D cache can be attributed to two main differences between the two microarchitectural simulators:

  The MARSS CPU model uses more aggressive approaches than Gem5 (and other simulators) for loads issue. Load instructions are issued as soon as possible and before aliasing with earlier stores is determined. For this reason, the number of executed loads in MaFIN is significantly larger than in GeFIN although (for each benchmark) the number of committed loads is very close to each other. This significant difference leads to extra masking of the faults in L1D on MaFIN and along with the previous point consistently explains the L1D cache vulnerability differences between the two tools. For example, in *fft*, *cjpeg*, *caes* (the benchmarks with largest difference in L1D between MaFIN-x86 and GeFIN-x86) MaFIN issues 2.6x, 4.7x, 2.0x more loads than GeFIN; this confirms the general trend.

  MARSS employs the QEMU hypervisor for system functions as well as for unimplemented instructions. When QEMU is invoked, the cache of the microarchitecture is not accessed (memory accesses go to the main memory) – for this reason faults in the L1D cache are masked and do not affect the operation when QEMU runs (this is not the case in the L1I cache; see below). Gem5 on the other hand handles the complete

system operation internally and does not employ a hypervisor so this type of masking does not happen.

However, in *qsort* and *smooth* the expected higher masking in MaFIN is not observed. For *qsort* GeFIN-x86 has smaller L1D read hit rate than in MaFIN-x86 (by 0.64x), while GeFIN-x86 has higher L1D write hit rate than MaFIN-x86 (1.91x in *qsort* and 1.57x in *smooth*); this means that in MaFIN-x86 for these benchmarks faults in L1D are less likely to be over-written and thus MaFIN-x86 is more vulnerable than GeFIN-x86.

- **Remark 4** – The prevailing faulty behavior in the L1D cache is the SDC class (intuitively expected) which leads to corrupted benchmark final output. In all benchmarks and the average case the SDC class is from 3x to 5x larger than the sum of all four other non-masked classes.
- **Remark 5** – The most remarkable differences between the different ISAs (GeFIN-x86 and GeFIN-ARM) for the L1D cache are observed in *fft*, *qsort* and *cjpeg*. The ARM model has 2x more store instructions than that of x86 in the *fft* benchmark, while in *cjpeg* the L1D write misses of the ARM model are 6x more than x86 model, which naturally leads to more vulnerability for the x86 model for these two benchmarks. The GeFIN-x86 model in *qsort* follows a completely different memory access pattern which reports significantly more L1D replacements (4x) than GeFIN-ARM. This indicates that the ARM model is more vulnerable for *qsort* than the x86 model.

The L1I cache vulnerability, on the other hand, is less variable across benchmarks than the L1D cache but still it can be as low as 5.3% (*smooth* benchmark in the MaFIN-x86 setup) and as high 34.5% (*caes* benchmark in the MaFIN-x86 setup). On average across benchmarks, L1I cache vulnerability is around 19% in MaFIN-x86 while in both ISAs of GeFIN (GeFIN-x86 and GeFIN-ARM) it is more than 14%. Here, the general trend in most (but not all) individual benchmarks is that MaFIN reports a more vulnerable L1I cache than GeFIN (the opposite trend to L1D reports).

- **Remark 6** – Unlike L1D, the QEMU hypervisor *does not affect* the behavior of L1I cache. QEMU may be invoked during decode stage only, which is *after* fetching (and accessing of L1I). This means that any faults residing in the L1I cache can be propagated without disturbance by the hypervisor.

  On the other hand, MARSS and Gem5 have differences in the implementation of their front-end that can lead to different prediction accuracy. Both simulators implement a Tournament predictor, consisting of a local and a global predictor. A meta-predictor takes the final decision based on the accuracy of the local and global ones. The most noticeable difference between MARSS and Gem5 is that the final prediction is bound to the branch address in the case of MARSS and to the global branch history in the case of Gem5. Branch address is not taken into account at all on the decision of Gem5 global predictor as well. This prediction scheme difference leads to different memory access patterns and L1I cache state; this can explain the small differences in the masked category between MaFIN-x86 and GeFIN-x86. Unfortunately, there is no consistent trend for all benchmarks. For instance, in the *edge*, *corner* and *sha* benchmarks MaFIN-x86 has by 0.83x, 0.82x, 0.68x less mispredictions than GeFIN-x86 which implies that GeFIN-x86 brings more L1I blocks from lower levels, increasing the probability to overwrite faults.
- **Remark 7** – The *fft*, *qsort*, *caes* are the benchmarks with difference more than 5 percentile points between GeFIN-x86 and GeFIN-ARM. For these benchmarks the replacements of L1I blocks in ARM model are 4.2x, 2.0x, and 7.2x more than in the x86; this can explain a more vulnerable x86 behavior than the ARM model.
- **Remark 8** – Fig. 4 shows that SDCs in the L1I cache are much less likely to be observed than in the L1D cache. The prevailing non-masked behavior in L1I cache in the MaFIN injector is the Assert class, while in the GeFIN injector the Crash class prevails. This difference is because MARSS simulator includes a significantly larger number of assertion checking points in its code which are raised during faulty executions of the benchmarks and stop the simulation abnormally. On the other hand, assertion checking in Gem5 is compact and less frequent and for this reason injected faults eventually lead to crashes.[6]

Second-Level Cache (L2)

The L2 cache memory vulnerability (Fig. 5) is in all cases (benchmark, ISA and microarchitecture configuration) a few percentile points higher than the Register File and LSQ and significantly lower than both first-level caches. On average, it ranges between 6% and 7% for the three ISA and microarchitecture combinations.

The difference in the L2 cache vulnerability between MaFIN and GeFIN is only about 1 percentile point which shows a consistent behavior between the two tools.

- **Remark 9** – Since L2 is unified the vulnerability reports show a balance between SDCs and other abnormal classes (Crashes etc.).
- **Remark 10** – Vulnerability differences larger than 5 percentile points are observed in *cjpeg* and *caes* benchmarks between MaFIN-x86 and GeFIN-x86 for the L2 cache. In *cjpeg* GeFIN-x86 has 1.2x more L2 write misses than MaFIN-x86, while in *caes* GeFIN-x86 has 1.54x more write hits than MaFIN-x86 increasing the probability that a fault is overwritten.

[6] We discuss this point here for the L1I cache vulnerability but it is also observed for the L1D cache in the non-SDC classes which include significantly more Assertions in MaFIN than Crashes (the case in GeFIN).

- **Remark 11** – Concerning the ISA differences between GeFIN-x86 and GeFIN-ARM, *djpeg* is the only benchmark with difference larger than 5 percentile points. In this case, the x86 model has 0.5x less L2 read hits and 6.8x more L2 write misses than the ARM model, making this benchmark less vulnerable for the x86 architecture.

V. Related Work

Previous work on microarchitecture-level fault injection includes papers that focus on the tools themselves as a stand-alone method for reliability assessment. A microarchitecture-level injection tool built on M5 simulator [4] for Alpha ISA only is briefly described in [48]. The injector was built on a simple in-order microarchitecture and reliability studies of complex out-of-order x86 or ARM microprocessors are not supported. An injection tool based on Gem5 and the Alpha ISA is described in [32]; the tool only injects transient faults in architectural registers. Very preliminary results of a MARSS-based (MaFIN-like) microarchitecture level injection are provided in [12]. Also, [11] [14] use PTLsim for fault injections on very few hardware structures. Unlike previous approaches our differential setup covers both state-of-the-art simulators (MARSS and Gem5), both x86 and ARM architectures, and provides fault injections capabilities of any fault type in all actual hardware structures of complex out-of-order microarchitectures.

Other approaches combine performance simulators with lower-level simulators to improve reliability assessments accuracy. The approach in [22] presents a combination of GEMS and Simics simulators with Cadence NC-Verilog gate-level simulator. For logic components it delivers a more accurate estimation at the expense of long simulation times. In [26] a fault injection method at the RTL and gate-level is described for the control blocks of an Alpha microprocessor.

Microarchitectural simulators have been also used for injections only at architectural visible points (the architectural registers) to measure the effectiveness of error protection techniques. In [35] the ASIM functional simulator is used and faults are only injected at the registers.

Other approaches are even lower level and work at the RTL, on FPGA realizations of a microarchitecture or on hardware emulators. The experimental study of [37] injects faults in a DLX processor FPGA realization and an ASIC realization of an Alpha processor. The framework described in [1] uses an FPGA-based system for the reliability characterization of a full system stack. In [34], an FPGA-based reliability analysis framework is described. In [45] and [46] an RTL model of an Alpha processor is developed and used for fault injection experiments. In [19] faults are injected in an RTL model of picoJava-II processor. In [36] a hardware emulation platform is used for injections at the latches of a microarchitecture. An interesting recent study [8] quantitatively evaluates the impact of flip-flop soft errors using several injection approaches at different levels of abstraction and discussed the sources of inaccuracies when higher levels of abstraction are employed in fault injection setups.

VI. Conclusions

We have presented a detailed vulnerability analysis of the hardware structures of out-of-order x86 and ARM models using a differential microarchitecture-level framework which employs two microarchitecture-level fault injectors (MaFIN and GeFIN – built on MARSS and Gem5). The fully parameterized tools support high-throughput, comprehensive injection campaigns for single and multiple transient, intermittent and permanent faults on one or more of the major hardware structures of the microarchitecture. The injectors can be used for differential studies on the reliability of hardware components running any workload, and support early design decisions for fault protection mechanisms.

We presented detailed characterization results for five important hardware structures employing ten benchmarks from the MiBench suite which is extensively used in reliability studies. We discussed the common trends in the reliability report and we explained diverging behaviors by provided insights about the internal implementations of the simulators.

In the average case, the reported differences between the two x86 injectors (MaFIN-x86 and GeFIN-x86) are larger than between the two ISAs implemented in Gem5 (GeFIN-x86 and GeFIN-ARM). The main sources of the differences are the different front-end structures of the simulators, the more aggressive memory requests approach that MARSS follows compared to Gem5 and also the use of the QEMU hypervisor in MARSS: the largest differences due to these reasons are observed in the cache memories. For particular benchmarks cases, significant differences are observed both between the two tools and also between the two ISAs and we have provided potential explanations for these differences based on the fundamental implementation differences of the simulators as well as runtime statistics of the benchmarks.

Acknowledgment

This work is supported by the 7th Framework Program of the European Union through the CLERECO Project, under Grant Agreement 611404, and also by EU and Greek national funds under the Thales/HOLISTIC project and the DIaSTEMA project.

References

[1] R.Balasubramanian, K.Sankaralingam, "Understanding the impact of gate-level physical reliability effects on whole program execution", HPCA 2014.

[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] N.L.Binkert et al., "The M5 simulator: modeling networked systems, IEEE Micro, vol. 26, no. 4, pp. 52-60, July/August 2006.

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

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

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

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

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

[10] L.Duan, B.Li, L.Peng, “Versatile prediction and fast estimation of architectural vulnerability factor from processor performance metrics”, HPCA 2009.

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

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

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

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

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

[16] A.Gutierrez et al., “Sources of error in full-system simulation”, ISPASS 2014.

[17] L.Huang, Q.Xu, “AgeSim: A simulation framework for evaluating the lifetime reliability of processor-based SoCs”, DATE 2010.

[18] Imperas. *OVPsim*, http://ovpworld.org [Online].

[19] S.Kim, K.Somani, “Soft error sensitivity characterization for microprocessor dependability enhancement strategy”, DSN 2002.

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

[21] M.-L.Li et al., “Understanding the propagation of hard errors to software and implications for resilient system design”, ASPLOS 2008.

[22] M.-L.Li, P.Ramachandran, U.R.Karpuzcu, S.K.S.Hari, S.V.Adve, “Accurate microarchitecture-level fault modeling for studying hardware faults”, HPCA 2009.

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

[24] Y.Luo et al., “Characterizing application memory error vulnerability to optimize datacenter cost via heterogeneous-reliability memory”, DSN 2014.

[25] P.S.Magnusson et al., “Simics: a full system simulation platform”, IEEE Computer, vol. 35, no. 2, pp. 50-58, February 2002.

[26] M.Maniatakos, N.Karimi, C.Tirumurti, A.Jas, Y.Makris, “Instruction-level impact analysis of low-level faults in a modern microprocessor controller”, IEEE Transactions on Computers, vol. 60, no. 9, pp. 1260-1273, Sept. 2011.

[27] M.K.Martin et al., “Multifacet's general execution-driven multiprocessor simulator (GEMS) toolset”, ACM SIGARCH Computer Arch. News, vol. 33, no. 4, November 2005.

[28] 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.

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

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

[31] 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.

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

[33] A.Patel, F.Afram, S.Chen, K.Ghose, “MARSS: A full system simulator for multicore x86 CPUs”, DAC 2011.

[34] A.Pellegrini, K.Constantinides, D.Zhang, S.Sudhakar, V.Bertacco, T.Austin, “CrashTest: A fast high-fidelity FPGA-based resiliency analysis framework”, ICCD 2008.

[35] P.Racunas, K.Constantinides, S.Manne, S.S.Mukherjee, “Perturbation-based fault screening”, HPCA 2007.

[36] P.Ramachandrant, P.Kudvatt, J.Kellingtont, J.Schumannt, P.Sanda, “Statistical fault injection”, DSN 2008.

[37] G.Saggese, N.J.Wang, Z.Kalbarczyk, S.J.Patel, R.Iyer, "An experimental study of soft errors in microprocessors" IEEE Micro, vol. 25, no. 6, pp. 30-39, Nov-Dec 2005.

[38] A.Savino et al., “Statistical reliability estimation of microprocessor-based systems”, IEEE Transactions on Computers, 2012.

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

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

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

[42] J.Stevens et al., “An integrated simulation infrastructure for the entire memory hierarchy: cache, DRAM, non-volatile memory, and disk”, Intel Technology Journal, vol. 17, no. 1, 2013.

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

[44] K.R.Walcott, G.Humphreys, S.Gurumurthi, “Dynamic prediction of architectural vulnerability from microarchitectural state”, ISCA 2007.

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

[46] N.J.Wang, J.Quek, T.M.Rafacz, S.J.Patel, “Characterizing the effects of transient faults on a high-performance processor pipeline”, DSN 2004.

[47] T.F.Wenisch et al., “SimFlex: Statistical sampling of computer system simulation”, IEEE Micro, vol. 26, no. 4, pp. 18-31, 2006.

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

[49] M.T.Yourst, “PTLsim: A cycle accurate full system x86-64 microarchitectural simulator”, ISPASS 2007.

[50] 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.

[51] G.-H.Asadi, V.Sridharan, M.Tahoori, D.Kaeli, “Balancing performance and reliability in the memory hierarchy”, ISPASS 2005.

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

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