Versatile Architecture-Level Fault Injection Framework for Reliability Evaluation: A First Report

Nikos Foutris Manolis Kaliorakis Sotiris Tselonis Dimitris Gizopoulos
Computer Architecture Laboratory, University of Athens, Greece
{nfoutris, manoliskal, tseloniss, dgizop}@di.uoa.gr

***Abstract*—Forthcoming technologies hold the promise of a significant increase in integration density, performance and functionality. However, a dramatic change in microprocessor's reliability is also expected. Developing mechanisms for early and accurate reliability estimation will save significant design effort, resources and consequently will positively impact product's time-to-market (TTM). In this paper, we propose a versatile architecture-level fault injection framework, built on top of a state-of-the-art x86 microprocessor simulator, for thorough and fast characterization of a wide range of hardware components with respect to various fault models.**

Keywords—reliability evaluation, architectural fault injection

I. Introduction

Semiconductor technology evolution has continuously provided more transistors for roughly constant power and cost per chip. Computer architects have exploited this growing transistor budget to develop sophisticated techniques that boost performance. However, technology scaling trends lead to increasingly unreliable microprocessor products [36]. Thus, measuring microprocessor reliability and providing means to guarantee correct operation is a critical challenge for the forthcoming technologies.

Accurate identification of the vulnerabilities of a microprocessor product, early in design time, assists designers to carefully plan for reliability enhancements with low cost and high power efficiency. On the contrary, inaccurate reliability estimation often results on over-designed microprocessors and negatively impacts time-to-market (TTM) and product costs. To put this in perspective, Table 1 shows the additional amount of logic required to protect SRAM arrays, such as cache memories, from single and multi-bit hardware faults; different protection techniques impose significantly different overheads. Thus, computer architects require tools for fast and accurate assessment of a component's reliability, so that they can make high level architectural trade-offs early in the design process without resorting to worst-case and guard-banding approaches.

Techniques	Detect (Protect)	Area Overhead
Parity	1/64 bits (none)	1.6%
SEC-DED	2/64 bits (1/64 bits)	12.5%
DEC-TED	3/64 bits (2/64 bits)	23.4%

Table 1: Area overhead of SRAM detection and correction techniques (X/Y means that a technique can detect and protect X bits for every Y bits) [21].

Modeling and estimating the reliability of microprocessor components can be achieved either through analytical methods or through fault injection experiments. Table 2 presents a qualitative comparison between four wide-spread techniques for early reliability estimation. The comparison is done, in terms of simulation time (i.e. the time needed to characterize a microprocessor component), fault model accuracy (i.e. how representative is the fault model) and reliability estimation accuracy (i.e. the error margin of the final estimation). Fault injection experiments are either based on an RT-level or an architecture-level simulator. While RTL fault injections more accurately capture lower level fault model, the excessively long simulation time of these schemes prevents detailed evaluation of components with statistically safe numbers of injection runs. On the contrary, architecture-level fault injections are very fast and allow the execution of complex workloads for long simulation intervals. Fault injection is an experimental reliability evaluation approach that provides sufficient accuracy and is applicable early in the design time where RTL netlist is not available. A recent study [6] compares the results of fault injection in the flip-flop level (on FPGAs) and the architecture-level, showing significant difference between the two.

Analytical methods, such as ACE (Architectural Correct Execution) analysis and probabilistic models, utilize a high-level performance model, which is available early in the design cycle, coupled with low level information about processor's reliability to provide early reliability estimation. However, analytical approaches provide a conservative lower bound regarding the reliability of a microprocessor [19] [42].

	RTL injection [42] [23]	**Arch. injection** [9] [41] [30]	**ACE analysis** [26] [27]	**Probabilistic models** [20] [38] [39]
Simulation Time	High	Medium	Low	None
Fault Model Accuracy	High	Medium	None	None
Estimation Accuracy	High	High	Medium	Medium

Table 2: Early reliability estimation methods: A qualitative comparison.

In this paper, we propose a comprehensive architecture-level fault injection framework for early reliability estimation of x86 microprocessors. The proposed framework:

- Is built on top of an x86 microprocessor model, simulated with MARSSx86 full-system simulator.
- Provides realistic reliability estimation for storage elements, since the microprocessor model is enhanced with the data arrays of cache hierarchy (not realized in the original model).
- Models transient, intermittent, and permanent faults as well as multi-bit faults of these models.
- Exploits the full system capabilities of a cycle-accurate simulator (MARSSx86), in conjunction with a functional machine emulator (QEMU environment), to completely execute a workload and to model fault propagation till the

higher level of a system stack, such as the operating system- and the application-level.

II. Architecture-level Framework

In this section, the proposed architecture-level fault injection framework for early reliability estimation is presented.

A. Simulator

Our infrastructure runs on top of MARSSx86 architectural simulator [31] which is probably the most comprehensive publicly available x86 simulation framework today. The x86 functional model of MARSSx86 is more accurate than other publicly available simulators and its memory system better models real systems [37].

MARSSx86 is widely used for performance measurements [4] [24] [37]. MARSSx86 utilizes PTLsim [44] to simulate the internal details of an x86 microprocessor model. PTLsim has been used for reliability measurements [8] [9] [11], as well as silicon validation [10]. MARSSx86 is a full system, cycle-accurate simulator capable of simulating a multicore processor with a detailed implementation of the front-end and the back-end pipeline stages of a modern x86-64 architecture. In addition, MARSSx86 simulates the cache hierarchy, which we extend with the data arrays (to allow realistic fault injections at all different cache levels L1, L2, L3), and implements several cache coherency protocols. To provide full system capabilities MARSSx86 is coupled with QEMU emulator. We selected MARSSx86 as the kernel of our framework due to the following reasons: (a) it accurately simulates an x86-64 microprocessor model; and (b) its full system simulation operation provides us with the capability to trace the propagation of a low-level hardware fault, till its manifestation on the operating system- or on application-level output.

B. Fault Models

Transient, intermittent and permanent faults are the main fault types exploited for reliability evaluation.

Transient faults (or soft error) [2] [3] [6] [12] [18] [19] [20] [26] [27] [30] [32] [33] [34] [35] [38]: Neutron radiation from cosmic rays, alpha particles from packaging materials, environmental and design variations can flip temporally the contents of a storage element, such as memory cells and flip-flops, causing an error. Moreover, while moving deeper into nanometer scale integration levels and near threshold voltage operation, the sensitivity of storage elements to such phenomena will increase enormously.

In our fault injection framework, *transient faults are modeled by flipping the value of a randomly selected bit in a randomly selected clock cycle during simulation*.

Intermittent faults [7] [41]: Wear-out behavior, process variation, voltage and temperature fluctuations can cause burst of frequent faults, called intermittent faults. Intermittent faults occur at irregular intervals, on the same location and last for a short period of time.

In our fault injection framework, *intermittent faults are modeled by setting the state of storage elements to* 1 *or* 0*, in a randomly selected cycle, for a random period*.

Permanent faults [1] [9] [11] [15] [23]: Electro-migration, gate oxides, time dependent dielectric breakdown, thermal cycling and negative bias temperature instability are some representative sources of permanent faults during system operation. In general, permanent faults tend to occur early in the processor's lifetime due to manufacturing defects that escape manufacturing testing or late in its lifetime due to wear-out effects.

In our fault injection framework, *a storage element that suffers from a permanent fault can be set persistently to one (stuck-at-1) or to zero (stuck-at-0) for the entire simulation time*.

Multi-bit faults [17] [32] [35] [38] [39]: Multi-bit transient, intermittent and permanent faults can occur in storage cells.

Multi-bit transient faults are classified into the following main categories: (a) *Spatial*: Occur when a single particle strike flips the state of multiple bits. Recent studies [13] [16] [22] show that spatial multi-bit faults are usually compact, i.e. faults are confined to a contiguous rectangle; and (b) *Temporal*: Result from multiple single-event upset (single, independent particle strikes) distributed over time [39].

Multi-bit permanent and intermittent faults are expected to increase in future microprocessors, due to the extreme scaling and the operation in reduced voltage levels for power reduction purposes [1] [5] [28] [44]. More specifically, the single bit failure probability (pfail) of SRAM cells is expected to fall in the range between 10^{-6} and 10^{-4} [5] [28] [44] which given a binomial probability distribution results in very high probabilities of multi-bit permanent faults in SRAM arrays.

In our fault injection framework, *multi-bit transient, intermittent and permanent faults are modeled based on the single-bit fault model with the difference that more than one bit is affected*.

C. Fault-Injection Framework

Our versatile architecture-level fault injection framework built on top of MARSSx86 architectural simulator is outlined at a high-level in Figure 1.

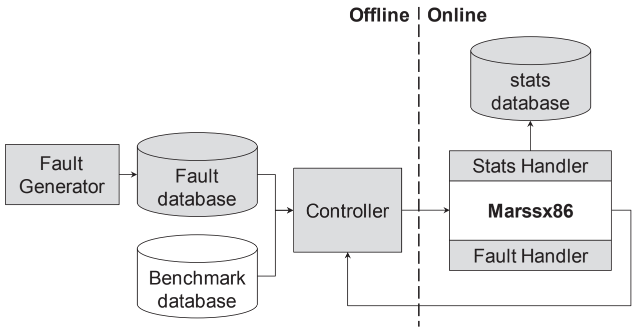

Figure 1: High-level block diagram of the architecture-level fault injection framework.

A guiding principle that we adopted throughout the development of our tool was to minimize the overhead induced by the simulator due to the fault injection infrastructure. Therefore, the proposed framework is separated into two main parts:

- *Offline Part*: The overall simulation time remains unaffected, since the procedures comprising this part are not in the critical path of the framework (i.e. do not affect simulation throughput). In particular, the offline part consists of the following processes: (a) population of the fault mask database from the *fault generation tool* (based on the fault models presented on section II-B); and (b) the *simulation controller*, which controls the fault injection experiments. The fault controller configures MARSSx86 simulator based on the user-defined parameters, launches the fault injection run, and collects the experimental results necessary to characterize the injected fault.

- *Online Part*: This part constitutes the kernel of the infrastructure. MARSSx86 simulator is extended with two new modules. The *statistics handler* and the *fault handler*. The former measures a variety of statistics relevant to the fault characterization process, while the latter controls the actual injection of a fault into the simulator based on the input parameter set defined by the simulation controller.

Figure 2 shows details of two key elements of the architecture-level fault injection infrastructure, the *fault mask database* and the *statistics database*. The fault mask generation tool produces fault masks with the following attributes: (1) *processor_id*: the targeted processor, (2) *module_id*: the targeted microarchitectural array in a processor, (3) *module_dimension*: the internal structure of the microarchitectural array (i.e., $Mod_{DIM} = M_{ROWS} \times N_{COLUMNS} \times L_{SIZE}$, where M_{ROWS} and $N_{COLUMNS}$ are equal to the number of rows and columns of an array and L_{SIZE} is the size of an entry. For example, $Mod_{DIM} = 1024 \times 4 \times 32$, means that the structure has 1024 rows, 4 columns per row and 32 bits per column), (4) *fault_type*: permanent, intermittent or transient fault, (5) *fault_dimension*: the internal structure of the fault (i.e., $F_{DIM} = M_{ROWS} \times N_{COLUMNS}$. For instance, $F_{DIM} = 2 \times 2$, means that a 4-bit fault is injected into an array, and is deployed as a rectangle), (6) *frequency*: how often a fault is activated (in our simulation environment an intermittent or a transient fault is activated only once during the simulation interval); and (7) *duration*: how many clock cycles the fault is active (refers to intermittent faults). Therefore, the fault handler parses the attributes of the fault mask and accordingly adjusts the simulated microprocessor model.

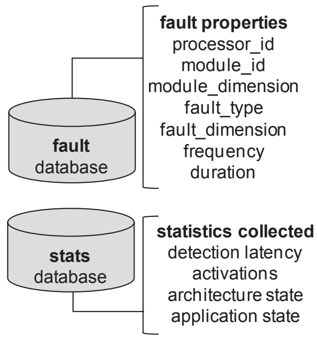

Figure 2: The properties of the key elements, fault mask database and statistics database, of the fault injection framework.

The statistics database is updated from the statistics handler module and is comprised of the following fields: (1) *detection_latency*: the time interval elapsed between the activation of a fault (clock cycle that the fault is excited for the first time) and its detection (clock cycle the fault is observed at an architecturally visible output), (2) *activations*: the access frequency of a faulty entry, (3) *architecture_state*: consists of the values of the program counter, and the architectural registers; and (4) *application_state*: the output of the application either on the stdout or on a hard disk file.

D. Fault Effect Characterization

The fault classification categories are shown in Figure 3. Depending on the system level that a fault is detected, we have the following fault classes:

- *Architecture-level*: A fault is detected at the architecture-level when a mismatch in the program counter, and/or the architectural registers is detected. The architecture-level classes are the following:
  - *SDC*: a silent data corruption in the architecture state of the microprocessor model (i.e. a mismatch with the fault-free case).
  - *Benign*: The architecture state of the fault-injected run equals to the fault-free execution.
- *Application-level*: A corruption in the application-level state reveals that the program output has been modified (either the stdout or a hard disk file) due to the fault. A fault detected at the application-level can be classified into the following categories:
  - *SDC*: a silent data corruption in the application output[1].
  - *DUE* (*detected unrecoverable error*): An unexpected exception, assertion, deadlock or interrupt occurred. Simulator crashes, either during simulation or emulation phase, are also clustered into this category. Finally, it should be noted that, false and true DUE are also included into this category.
  - *Benign*: The workload executed completely without manifesting any mismatch at its output compared with the fault-free execution.
  - *Hangs*: The application does not terminate within a reasonable time interval (in our framework the limit is set to 3x the fault-free execution time).

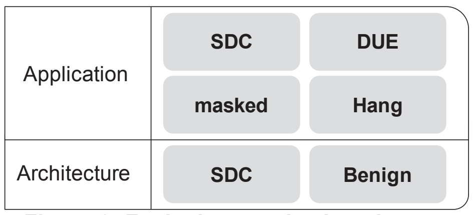

Figure 3: Fault characterization classes.

E. Simulation Timeline

In this sub-section we present the timeline of a single fault injection simulation run (Figure 4). We exploit the checkpoint capabilities of MARSSx86 simulator to avoid the initialization phase of a workload, bound the indeterminism induced due to the interaction with the OS and reach to a workload's point of interest. When the workload reaches this point, we switch to simulation mode and warm-up the structures of the

[1] Architecture-level SDCs may also result in application-level SDCs.

microprocessor model for a user-defined amount of cycles (warm-up period is configured during the initialization phase from the simulation controller). After committing a user-defined number of *k* x86 instructions (including user and kernel instructions), the statistics collected so far are reset, the fault handler is called and the fault injected simulation period is launched (similarly to the warm-up period, the simulation period is a user-defined parameter defining the *n* x86 instructions that will be simulated).

At the simulation end, the architecture states (of the golden and the fault-injected runs) are compared. In case of a mismatch an architecture-level SDC is detected, otherwise the fault is benign (equal architecture states). For cases where the architecture state is faulty, the thread context (i.e., program counter, architectural registers, and virtual memory) is transferred to the QEMU virtualization environment and executed until completion (this is feasible due to the high throughput of the QEMU emulator). At workload's completion, we compare the golden application state (which was generated offline) with the fault-injected state to detect whether the faulty propagates and corrupts a user-visible output (i.e., operating system-, or application-level). If does so, then is classified as an application-level SDC fault. If an exception, a deadlock, an assertion, an interrupt, or a simulator crash occur during the workload execution, then an application-level DUE fault is detected. If, the execution exceeds the maximum execution interval (3x the fault-free one) then a hang, is detected. Finally, if the application finishes normally and the output of the faulty execution is clear, the fault is classified as masked. The aforementioned process is iterated for every injected fault.

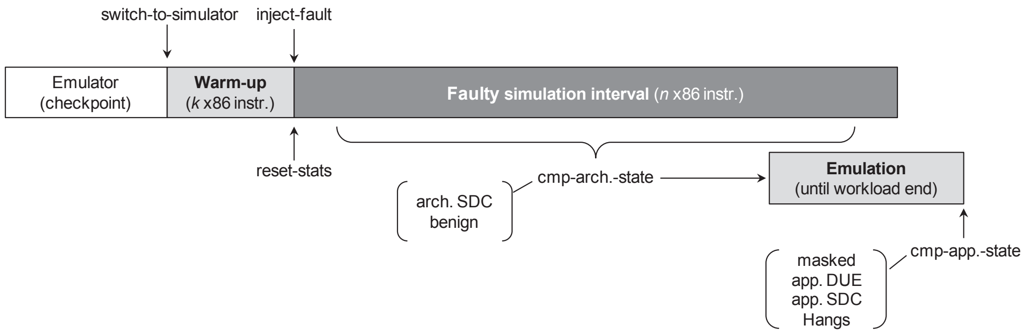

Figure 4: Fault injection simulation timeline.

III. Experimental Results

We evaluate the time implications induced from the "online part" (Section II-C) of our fault injection framework. The "offline part" is a one-time process and does not affect simulation throughput.

Table 4 reports the very small overhead induced on the unmodified model of MARSSx86 simulator (original model) from our fault-injection framework (enhanced model), measured in millions of x86 instructions committed per second (MIPS). The contents of Table 4 are the average values of three independent runs. The enhanced model has the following functionalities enabled: (a) the cache memories are enhanced with the data arrays, (b) the fault injection handler is integrated (for demonstration purposes, single, quintuple and tenfold transient and permanents faults are injected into L1 data cache array. The permanent fault injection constitutes the worst-case scenario in terms of simulation overhead, since it is active and affecting functionality of the simulator throughout the entire simulation interval); and (c) the statistics handler measures statistics relevant to the detection latency and the fault activations. The fault detection comparison happens at the end of the simulation interval (to detect an architecture state mismatch) and after the completion of the application run (to detect application level errors).

Data presented on Table 4 are produced from the end-to-end execution of a memory-intensive application, a bubblesort algorithm (sorting 10,000 integer), when 0, 1, 5, and 10 transient faults are injected into the L1 data cache (Table 3 shows the configuration of the simulated x86 model). The host machine of the experiments was an Intel i7-3970X CPU clocked at 3.5 GHz using 32 GBytes of RAM and running Ubuntu 12.04.04 LTS operating system.

Parameter	Setting
Pipeline depth	24 (max branches in-flight)
Fetch/Issue/Commit	4/4/4 instructions per cycle
RAS	16 entries
BTB	4KB (4-way set associative, 1K entries)
Combined Predictor	16KB (64K entries, 2 bits per entry, 16 bits BHR) meta predictor table: 64K entries
Issue Queue	16 entries (one per cluster)
Reorder Buffer	128 entries
Functional Units	4 clusters (ALUs: 4 INT, 4 FPU)
L1 instruction cache	64KB (64B line, 512 sets, 2-ways, 2 cycles latency)
L1 data cache	64KB (64B line, 512 sets, 2-ways, 2 cycles latency)
L2 cache	2MB inclusive (64B line, 8-ways, 5 cycles latency)
Main memory	Infinite size (50 nsec latency)

Table 3: Enhanced x86 microprocessor configuration.

The enhanced model of MARSSx86 simulator has a lower simulation throughput than the original model but the overhead on the simulation time from the integration of the fault injection framework is up to 5.8% when tenfold permanent faults are injected (3.3% when 10 transient faults are injected). In particular, 1.5% (out of 5.8%) is due to modeling the data array functionality on the simulator, while the rest 4.3% is due to the fault injection-related source code. Another key insight of these first results is that simulation throughput does not increase significantly as the number of injected faults is increased. On contrast, it remains almost stable. In particular, on singe and tenfold permanent faults injections the simulation throughput is 0.0467 and 0.0456, respectively, while on the transient fault injections the simulation throughput is equal to 0.0470 with single faults injected and 0.0468 with then faults injected.

		Original Model (MIPS)	Enhanced Model (MIPS)	Slowdown
Simulated x86 Instructions per second	**Transient Fault Injections**			
	No fault	0.0484	0.0477	**1.5%**
	1-fault	-	0.0470	**2.9%**
	5-faults	-	0.0469	**3.1%**
	10-faults	-	0.0468	**3.3%**
	Permanent Fault Injections			
	1-fault	0.0484	0.0467	**3.5%**
	5-faults	-	0.0461	**4.7%**
	10-faults	-	0.0456	**5.8%**

Table 4: Comparison of the original model of MARSSx86 simulator with the enhanced model (i.e. MARSSx86 with the proposed fault injection framework integrated) in terms of x86 simulated instructions per second on the host machine ("No fault" experiment measures the overhead induced only from the integration of the data arrays into the cache memory of MARSSx86 simulator).

Table 5 presents some first results for the reliability evaluation of the L1 data cache memory. Single-bit transient faults (randomly selected from the fault mask database) are injected in a randomly selected entry at randomly selected clock cycle (Section II-E). To evaluate cache memories, we develop four memory-intensive applications: (a) *vectorADD*$_1$: adds two arrays consisting of 10,000 integer elements and prints the sum in hard-disk file at the end of the calculations, (b) *vectorADD*$_2$: adds two arrays consisting of 10,000 integer elements and prints the sum in hard-disk file after each individual addition, (c) *mMul*: a matrix multiplication algorithm (100x100 integer arrays) that stores the product of operation in a hard-disk file at the end; and (d) *bubblesort*: a sorting algorithm (for 1,000 integer numbers). Overall, 1,265 fault injection experiments were performed (125 running *vectorADD*$_1$, 664 with *vectorADD*$_2$, 336 executing *mMul*, and 140 with the *bubblesort* algorithm). These numbers of injection give only a first report on the reliability studies that can be performed on our framework. Accurate reliability estimates for the caches and other structures can be only derived from significantly more extended campaigns with larger numbers of injected faults.

The variation in the rates of fault categories across the four applications is mainly related to their different data use profiles. Similar behavior is also encountered in analytical methods, such as [12], where L1 data cache AVF approximately varies from 8% to 42% depending on the benchmark. Compared to analytical methods our fault injection experiments show lower cache SDC vulnerability in three of the four applications. This is justified from the fact that analytical methods to calculate the AVF provide an over-estimation regarding component's vulnerability, which results in over-designed microprocessors and negatively impacts time-to-market (TTM) and product costs.

Fault Category	*VectorADD*$_1$	*VectorADD*$_2$	*mMul*	*BubbleSort*
App. SDC	2.4%	0.6%	47.5%	7.8%
App. DUE	0.8%	0.2%	0.6%	0.0%
Hang	17.4%	0.3%	0.3%	0.7%
Masked	79.4%	98.9%	51.6%	91.5%

Table 5: First results for the reliability evaluation of a L1 data cache memory with single transient fault injection.

IV. Related Work

Reliability evaluation has been carried out at various abstraction levels. Mainly is classified into simulation-based techniques and analytical methods.

Simulation-based methods: Many simulation based fault injection schemes in various abstraction levels have been proposed in the literature. Wang *et al*. [43] explored the effect of transient faults on IVM microprocessor model though RT-level fault injections, while on [42] examined the correlation between ACE analysis and RT-level fault injection experiments. Li *et al*. [18] proposed a hybrid simulation infrastructure to reliability evaluate various microprocessor components. Maniatakos et al. [23] developed a fault injection infrastructure on IVM microprocessor model. In [17] proposed a hardware-software framework to characterize a system under the presence of permanent faults. The authors of [41] proposed a fault injection framework on a microarchitectural simulator to perform dependability analysis. In [30] a fault injection tool based on the cycle accurate full system simulator Gem5 is proposed. Authors of [14] propose a technique to reduce the fault simulation time through grouping error simulations that produce same intermediate execution state. In [25], a statistical method to estimate the outcome of a system in presence of soft errors is proposed. In this paper, we propose an architecture-level fault injection framework for early reliability evaluation on the storage structures of a modern x86-64 architecture enhanced with data arrays in the cache hierarchy. Several research approaches proposed a reliability prediction framework build at the circuit- or gate-level [29] [40]. Even though their high accuracy, the low simulation throughput prevents the detailed evaluation of the propagation of faults into the higher level of the system stack. Finally, FPGA-based fault injection environments offer high throughput simulations, though the limited observability and controllability gives less flexibility [32] [33] .

Analytical Methods: Mukherjee et al. [26] introduced ACE analysis. Biswas et al. [3] extended the original ACE analysis framework to enable address-based processor structures. Fu et al. [12] proposed a unified framework for estimating microprocessor reliability in the presence of soft errors at the architecture level. Sridharan et al. [35] introduced hardware vulnerability factor for hardware vulnerability estimation to bound the inaccuracy of AVF measurements. Savino et al. [34] proposed an analytical way to estimate the reliability of a microprocessor-based system. Finally, Suh et al. [38] proposed a Markov model for reliability evaluation of cache under the presence of single and multi-bit upsets.

V. CONCLUSIONS

Early estimation of microprocessor reliability, to support the employment of efficient methods that guarantee correct operation is critical for forthcoming technologies. We have presented a first report on a flexible architecture-level fault injection framework for early reliability evaluation. The developed infrastructure was built on top of MARSSx86 full system simulator and supports characterization in the presence of any number of hardware faults of different types (transient, intermittent, permanent) in microprocessor components.

ACKNOWLEGMENT

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's European Social Fund (ESF) and Greek national funds under the "Thales/HOLISTIC" project and the "DIaSTEMA" project.

REFERENCES

[1] J.Abella, J.Carretero, P.Chaparro, X.Vera, A.Gonzalez, "Low Vccmin Fault-Tolerant Cache with Highly Predictable Performance", MICRO, 2009.

[2] R.C.Baumann, "Soft Errors in Advanced Computer Systems", IEEE Design & Test of Computers, vol.22, no.3, pp. 258-266, May/June, 2005.

[3] A.Biswas, P.Racunas, R.Cheveresan, J.Emer, S.S.Mukherjee, R.Rangan, "Computing architectural vulnurability factors for address-based structures", ISCA, 2005.

[4] M-T.Chang, P.Rosenfeld, S-L.Lu, B.Jacob, "Technology comparison for large last-level caches (L3Cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM", HPCA, 2013.

[5] Z.Chishti, A.R.Alameldeen, C.Wilkerson, W.Wu, S.-L.Lu, "Improving Cache Lifetime Reliability at Ultra-low Voltages", MICRO, 2009.

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

[7] C.Constantinescu, "Trends and challenges in vlsi circuit reliability", IEEE Micro, 23:14-19, July, 2003.

[8] S.Feng, S.Gupta, A.Ansari, and S.Mahlke, "Shoestring: Probabilistic Soft Error Reliability on the Cheap," ASPLOS, 2010.

[9] N.Foutris, D.Gizopoulos, J.Kalamatianos, V.Sridharan, "Assessing the impact of hard faults in performance componets of modern microprocessors", ICCD, 2013.

[10] N.Foutris, D.Gizopoulos, X.Vera, A.Gonzalez, "Deconfigurable microprocessor architectures for silicon debug acceleration", ISCA, 2013.

[11] N.Foutris, D.Gizopoulos, A.Chatzidimitriou, J.Kalamatianos, V.Sridharan, "Performance Assessment of Data Prefetchers in High Error Rate Technologies", SELSE, 2014.

[12] X.Fu, T.Li, J.A.B.Fortes, "Sim-SODA: A unified framework for architectural level software reliabilty analysis", Workshop on Modeling, Benchmarking and Simulation, 2006.

[13] G.Georgakos, P.Huber, M.Ostermayr, E.Amirante, F.Ruckerbauer, "Investigation of increased multi-bit failure rate due to neutron induced SEU in adcanced embedded SRAMs", VLSIC, 2007.

[14] S.Hari, R.Venkatagiri, S.Adve, H.Naeimi, "GangES: Gang Error Simulation for Hardware Resiliency Evaluation", ISCA, 2014.

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

[16] E.Ibe, S.S.Chung, S.Wen, H.Yamaguchi, Y.Yahagi, H.Kameyama, S.Yamamoto, T.Akioka, "Spreading diversity in multi-cell neutron-induced upsets with device scaling", CICC, 2006.

[17] M-L.Li, P.Ramachandran, S.K.Sahoo, S.V.Adve, V.S.Adve, Y.Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design", ASLPOS, 2008.

[18] 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.

[19] M-L.Li, S.V.Adve, P.Bose, J.A.Rivers, "Architecture-level soft error analysis: examining the limits of common assumptions", DSN, 2007.

[20] M-L.Li, S.V.Adve, P.Bose, J.A.Rivers, "SoftArch: An architecture-level tool for modeling and analysing soft errors", DSN, 2005.

[21] Y.Luo, S.Govindan, B.Sharma, M.Santaniello, J.Meza, A.Kansal, J.Liu, B.Khessib, K.Vaid, O.Mutlu, "Characterizing Apllication Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous – Reliability Memory", DSN, 2014.

[22] N.Mahatme, B.Bhuva, Y.Fang, A.Oates, "Analysis of multiple cell upsets due to neutrons in SRAMs for deep-n-well process", IRPS, 2011.

[23] M.Maniatakos, N.Karimi, C.Tirumurti, A.Jas, Y.Makris, "Instruction-level impact analysis of low-level faults in a modern microprocessor controller", IEEE ToC, 2010.

[24] A.Mayberry, M.Laquidara, C.Weeds, "Characterizing the microarchitectural side effects of operating system calls", ISPASS, 2013.

[25] S.Mirkhani, S.Mitra, C-Y.Cher, J.Abraham, "Effective Statistical Estimation of Soft Error Vulnurability for Complex Designs", SELSE, 2014.

[26] S.S.Mukherjee, C.T.Weaver, J.Emer, S.K.Reinhardt, T.Austin, "A systesmatic methodology to compute the architectural vulnerability factors for a high-performance microprocessors", MICRO, 2004.

[27] A.Nair, S.Eyerman, L.Eeckhout, L.K.John, "A first-order mechanistic model for architectural vulnurability factor", ISCA, 2012.

[28] S.R.Nassif, N.Mehta, Y.Cao, "A Resilience Roadmap", DATE, 2010.

[29] G.Papasso, D.Rossi, C.Metra, M.Omana, "A model for transient fault propagation in combination logic", IOLTS, 2003.

[30] K.Parasyris, G.Tziantzoulis, C.Antonopoulos, N.Bellas, "GemFI: A Fault Injection Tool for Studying the Behavior of Applications on Unreliable Substrates", DSN, 2014.

[31] A.Patel, F.Afram, S.Chen, K.Ghose, "MARSSx86: a full system simulator for multicore x86 CPUs", DAC, 2011.

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

[33] M.Rebaudengo, M.Sonza Reorda, M.Violante, "An accurate analysis of the effects of soft errors in the instruction and data caches of a pipelined microprocessors", DATE, 2003.

[34] A.Savino, S.Di Carlo, G.Politano, A.Benso, A.Bosio, G.Di Natale, "Statistical reliability estimation of microprocessor-based systems", IEEE Transactions on Computers, 2012.

[35] V.Sridharan, D.R.Kaeli, "Using hardware vulnerability factors to enhance AVF analysis", ISCA, 2010.

[36] J.Srinivasan, S.V.Adve, P.Bose, S.J.Patel, "The impact of technology scaling on lifetime reliabilty", DSN, 2004.

[37] J.Stevens, P.Tschirhart, M-T.Chang, I.Bhati, P.Enns, J.Greensky, Z.Cristi, S-L.Lu, B.Jacob, "An integrated simulation inftrastructure for the entire memory hierarchy: cache, dram, nonvolatile memory, and disk", Intel Technology Journal, vol.17, no 1, 2013.

[38] J.Suh, M.Annavaram, M.Dubois, "MACAU: A markov model for reliablity evaluations of caches under single-bit and multi-bit upsets", HPCA, 2012.

[39] J.Suh, M.Manoochehri, M.Annavaram, M.Dubois, "Soft error benchmarking of L2 caches with PARMA", SIGMETRICS, 2011.

[40] N.Touba, K.Mohanram, "Partial error masking to reducee soft error failure rate in logic circuits", DFT, 2003.

[41] G.Yalcin, O.S.Unsal, A.Cristal, M.Valero, "FIMSIM: A fault inejction infrastructure for microarchitectural simulators", ICCD, 2011.

[42] N.Wang, A.Mahersi, S.J.Patel, "Examining ACE analysis reliability estimates using fault-injection", ISCA, 2007.

[43] N.J.Wang, J.Quek, T.M.Rafacz, S.J.Patel, "Characterizing tge effects of transient faults on a high-performance processor pipeline", DSN, 2004.

[44] C.Wilkerson, H.Gao, A.R.Alameldeen, Z.Chishti, M.Khellah, S.-L.Lu, "Trading off Cache Capacity for Reliability to Enable Low Voltage Operation", ISCA, 2008.

[45] M.T.Yourst, "PTLsim: A cycle Accurate Full System x86-64 Microarchitectural Simulator", ISPASS, 2007.