Register File Criticality on Embedded Microprocessor Reliability

Filipe M. Lins, Lucas Tambara, Fernanda L. Kastensmidt, and Paolo Rech
UFRGS, Porto Alegre, Brazil

Abstract—**We select six representative benchmarks, each compiled with three different levels of compiler optimization. We performed exhaustive fault-injection campaigns to measure the registers Architectural Vulnerability Factor of each configuration, identifying the registers that are more likely to generate Silent Data Corruption or Single Event Functional Interruption. Finally, we irradiate with heavy ions two of the selected benchmarks, compiled with two levels of optimizations, and correlate experimental results with fault-injection analysis.**

I. Introduction

Commercial-Off-The-Shelf (COTS) systems became attractive for safety-critical applications, like biomedical implantable devices, automotive control systems, and aircraft or satellite stabilizers and control circuitry. The main reason for preferring a COTS device to a specifically designed rad-hard chip is that the latter are typically very expensive, as they require unique circuit design and lithography to meet the reliability requirements, and the produced volumes are very low. On the contrary, COTS components are low-cost, flexible, have fast time-to-market and low power consumption.

Examples of COTS systems used in PhoneSat nano-satellite [14] are Programmable System on Chips that are composed (PSoC) of a CPU core and mixed-signal arrays of configurable integrated analog and digital peripherals. These systems are susceptible to radiation xxxxx Faults can occur in XXX and errors are observed xxxx.

The processor is the core of computing systems and, thus, it must be very reliable. When an error occur in the processor, the executed code can fail producing a Silent Data Corruption (SDC), a Single-Event Functional Interrupt (SEFI). Modern embedded processor are built with Reduced Instruction Set Computing (RISC) architecture. RISC allows only few basic instructions to be executed, and inputs can come only from register file. In other words, RISC is a load-store architecture: data in the main memory must be load to the register file to be digest and data in register file must be stored in the main memory to be read by the user. Hence, register file is a critical resource for modern computing architecture. As any data to be processed will need to pass through the register file, knowing the probability of an error in a register to propagate to the output can be sufficient to characterize the vulnerability of an application. While being a critical resource, registers are not easily protected. Unlikely caches or DRAM which can be easily protected with Error Correcting Codes (ECC), register file are integrated in the logic circuit of the processors increasing the efforts and penalties of adding ECC [3].

In this paper we focus in embedded System on Chips (SoCs) register file criticality. Register file has been identified among the most critical resources in modern computing systems [19], [16]. Through a homemade fault injection platform we identify the criticality of each available general purpose register measuring the probability for the injection to generate a Silent Data Corruption (SDC) or Single Event Functional Interruption (SEFI) in Matrix Multiplication (MxM), Advanced Encryption Standard (AES), Quicksort, Fast Fourier Transform (FFT), Fibonacci, and Joint Photographics Experts Group (JPEG) algorithms.

RISC architectures became popular thanks to the advances in compiler efficiency. To succeed in executing complex algorithms using few, simple, two-inputs instructions the compiler needs to significantly modify the source code. In recent years, compilers allow user to select from different levels of optimization to be applied to the code. Optimization is achieved by modifying the number, the use, and the name of registers. The optimized code can hugely increase the performances of the applications but it increase the use of registers [2]. We use our fault injector to investigate the impact of three compiler optimizations (O0, O2, O3) on the number of critical registers and on their criticality (i.e., the impact of injection on the application output). Finally, we expose the SoC executing MxM and AES compiled with different compiler optimizations for heavy ion beam. Results show trends similar to fault injection.

The main contributions of this paper are: (1) an interruption-based fault injector that access embedded processor memories; (2) the evaluation of compiler optimization effects on registers criticality; (3) the correlation of fault injection evaluation with heavy ion experiment results.

The remainder of the paper is organized as follows. Section II provides a background on RISC architectures and compiler optimization, Section III describes the homemade fault-injection platform, the heavy ions experimental setup, and the tested codes. Section IV presents and discusses fault-injection and heavy ions experimental results while Section V concludes the paper.

II. Background on Compiler Optimizations

GCC compiler enables a plethora of optimization. Some optimizations reduce the size of the resulting machine code, while others try to create code that is faster, potentially increasing its size as a result (loop unrolling increases the assembly code but reduces the execution time).

Table I shows the most effective compiler operations for the different levels we consider (01, O2, O3). *Copy Propagation*

Optimization	O0	O1	O2	O3
Copy Propagation Registers		X	X	X
Alignments			X	X
Loop			X	X
Registers Move			X	X
Inline Function				X
Rename Registers				X

TABLE I: GCC optimizations and the levels at which they are enabled.

Registers reduces scheduling dependencies and occasionally eliminates copies; *Rename Registers* attempts to avoid false dependencies in scheduled code using available registers; and *Registers Move* maximizes the amount of register tying. Alignments in functions, loops, jumps, and labels maximize the number of used registers. *Loop optimizations* move constant expressions out of loops, and optionally do strength-reduction and loop unrolling.*Function Inline* integrates all simple functions into their callers.

O0 is the level which have not any optimization option, the compiler's goal is to reduce the cost of compilation.

O1 is the first level of optimization and produces an optimized binary in a short amount of time.

O2 performs all others supported optimizations within the given architecture that do not involve a space-speed trade-off. For example, loop unrolling and function inline are not performed.

O3 is the third and highest level enabled, which empathizes speed over size. This includes optimizations enabled at O2 and register rename. The optimization inline-functions also is enabled in O3, which can boost performance but also can drastically increase the size of the assembly code.

III. Methodology

The methodology consists of using fault injection at software level to inject bit-flips in the register file of a processor and analyze the error effect.

A. Device Under Test

The present study is based on the commercial-off-the-shelf Zynq-7000 APSoC designed by Xilinx in a TSMCs 28nm technology node. The Device Under Test (DUT), an XC7Z020-CLG484 part, is embedded in a commercially available ZedBoard Development Board. We focus our study on one of the embedded dual-core 32-bit ARM Cortex-A9 processor.

B. Fault Injector

In order to evaluate registers criticality on embedded SoC we build an interruption-based fault injector platform able to modify the values stored in the ARM internal registers. Injection is triggered by an interruption which launches a dedicated procedure that modifies a randomly selected register value.

In the literature there are several interesting fault injection that works at RTL level (like Amuse [4]), instruction level (like IVM [8]), or simulation (like gem5 [11]). Injected failures in real hardware have several advantages compared to instruction-level or RTL simulation injections. (1) As the application under test is executed on the hardware itself, the fault injection experiment is more rapid. Our fault injection takes only about $63,000$ additional clock cycles (0.02 ms) than the original code to be characterized. (2) Fault injection at instruction level is typically less precise than low level injections. High level fault injection, when compared to lower level fault injection, approaches magnitude orders faster and allows execution of large workload portions to study the effect of faults to the final program output [5], [13], However, results are more related to software or algorithm vulnerability, which does not deeply consider the architecture in which the code is running [17]. (3) Gate level injections are extremely precise, but they require the RTL level description of the circuit, which is hardly available for COTS systems, such as the one we address in this paper. Finally, even if the RTL level description is available, an exhaustive fault injection at gate level is extremely time-consuming.

The fault injection developed in this work is implemented at software level based on the previous work of [20]. Many changes and updates were performed. XXXX This technique allows injecting faults in registers that are accessible by the application and memory.

This work focuses on injecting faults only in the register file. The choice of focusing only on register file has three main reasons. (1) Registers are among the most critical resources for embedded processors. Corruption on data stored in a register is likely to affect the execution output or the program flow. In other words, the Architectural Vulnerability Factor (AVF) of register is high [9]. (2) Although caches are considered among the most vulnerable parts of the Zynq ARM [15], they can be protected with parity or ECC. On the contrary, register file must be compact, dense, and fast. These constraints make ECC implementation challenging and not always possible. (3) In ARM and other architectures is based on load/store which forces that all, data in the cache must be loaded in a register to be processed. The higher the AVF of a register, the higher the probability that a corrupted data loaded from the cache on that register will propagate to the output. While injection on register cannot provide the overall sensitivity of a device, it helps understanding AVF assembly codes.

The main components included in our fault injection platform are: (1) Zynq-7000 Processing System, which is the embedded ARM; (2) Advanced eXtensible Interface (AXI) Peripheral, which enables the serial communication with the host PC; (3) AXI Timer, which is a timer module attached to the AXI4-Lite interface, used to trigger the interruption that will inject the failure.

The flowchart of our interruption-based fault injector is show in Figure 3. The main steps performed in order to inject faults and measure the register criticality are:

(1) Configure the timer. Set a random value for injection, which is basically the clock cycle number at which the interruption is triggered.

(2) Execute application and inject fault. A precomputed

		Performance Info			Area Usage		Fault injection		Reliability Metric	
App.	Opt.	# Inst.	# Clock Cycles	Exec. Time (ms)	Mem. Footprint	Reg. File Usage	AVF TOTAL SDC	AVF TOTAL SEFI	MWBF	Best Result
MxM	O0	271473	40606	1,22E-01	94768	0,38	**0,12**	**0,00**	**108766,10**	
	O2	73988	8010	2,40E-02	94332	0,85	0,41	0,13	123245,95	
	O3	73515	**7550**	2,26E-02	94948	0,85	0,36	0,07	165442,91	**1,52**
AES	O0	97782	17958	5,39E-02	110080	0,38	0,12	0,03	197308,63	
	O2	33066	4402	1,32E-02	97972	0,31	**0,03**	**0,11**	**847678,40**	**4,30**
	O3	31757	**3823**	1,15E-02	101056	0,38	0,04	0,19	605646,01	
QuickSort	O0	330946	107882	1,62E-01	48504	0,31	**0,04**	**0,03**	**145591,94**	**2,15**
	O2	148302	**38104**	5,72E-02	48164	0,92	0,35	0,03	73200,33	
	O3	166582	40264	6,05E-02	50492	0,85	0,36	0,03	67576,37	
FFT	O0	78931	14496	4,35E-02	94040	0,38	**0,06**	**0,03**	**403952,93**	**3,24**
	O2	46928	6782	2,03E-02	92604	1,00	0,39	0,24	124639,44	
	O3	46870	**6738**	2,02E-02	92668	1,00	0,36	0,23	133154,49	
Fibonacci	O0	10577461	1131645	3,40E+00	46516	0,31	**0,08**	**0,03**	**4280,18**	
	O2	4806704	705765	2,12E+00	46348	0,62	0,23	0,07	2542,28	
	O3	4217336	**433657**	1,30E+00	47352	0,85	0,12	0,07	6619,98	**1,55**
JPEG	O0	5008939	724533	2,17E+00	112432	1,00	**0,11**	**0,06**	**4263,79**	**106,34**
	O2	21514145	251634	7,56E+01	107764	1,00	0,40	0,13	40,10	
	O3	2452638	**229917**	6,90E+01	121692	1,00	0,34	0,18	44,29	

TABLE II: Fault Injection Results

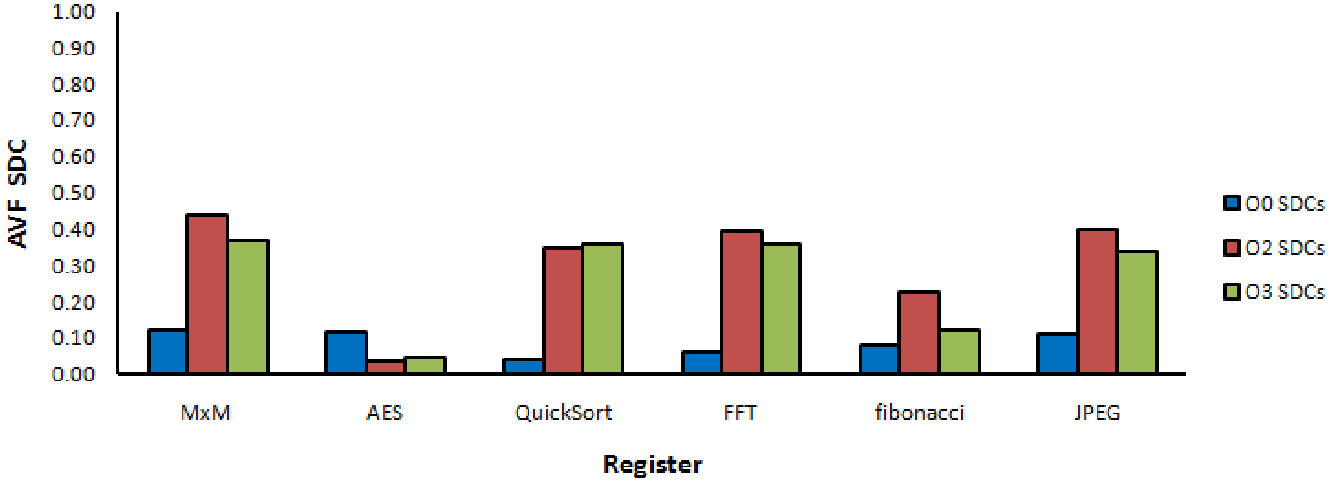

Fig. 1: Total Architectural Vulnerability Factor SDCs

input vector is sent to the ARM. Application execution is triggered and at each clock cycle the timer is decreased. When the counter reaches 0 the system calls the interruption, which randomly selects a register and one bit (multiple bit corruptions are also possible but not considered in this paper). The selected bit is flipped, interruption returns, the system restores the context and proceeds with application execution.

(3) Error detection. When application execution is completed, the output is compared with a precomputed golden copy. A mismatch between the two is identified as a Silent Data Corruption (SDC). Besides the corrupted and expected output value we also log the register where we injected the fault that causes the observed SDC. Whenever ARM becomes unresponsive or sends garbage through serial communication, a SEFI is detected by a watchdog in the host PC and the system is rebooted.

C. *Heavy Ions Experiment*

Heavy ion experiments were performed at were performed at the Nuclear Physics Open Laboratory (LAFN), São Paulo University (USP), Brazil [1] with a flux in the range of 102 to 105 $particles.cm^{-2}.s^{-1}$, as recommended by the European Space Agency (ESA) for SEU tests [18] We irradiate the Zynq with ^{12}C beam, scattered by a 184 $\mu g.cm^2$ gold target, with Linear Energy Transfers (LETs) from 2.6 to 17 $MeV.mg^{-1}.cm^{-2}$, and penetration in Si from 16.6 to 47 μm.

D. *Selected Benchmarks*

The proposed software benchmark includes a collection of different, simple algorithms that are realistic software codes and a few standard software benchmarks. We use our interruption-based fault injector and heavy ion setup to evaluate compiler optimization effects on register file relia-

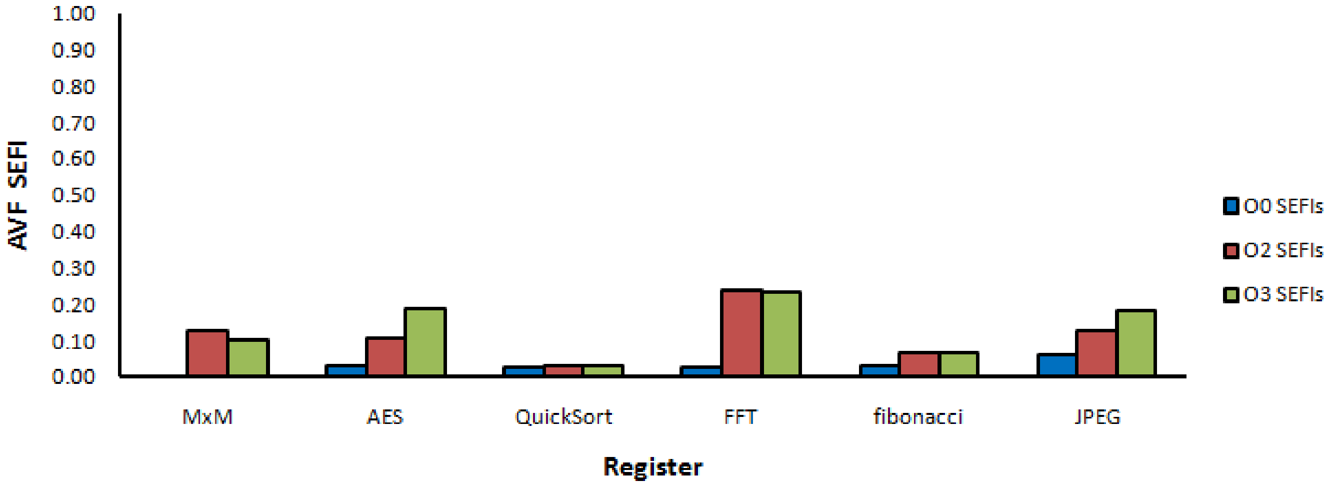

Fig. 2: Total Architectural Vulnerability Factor SEFIs

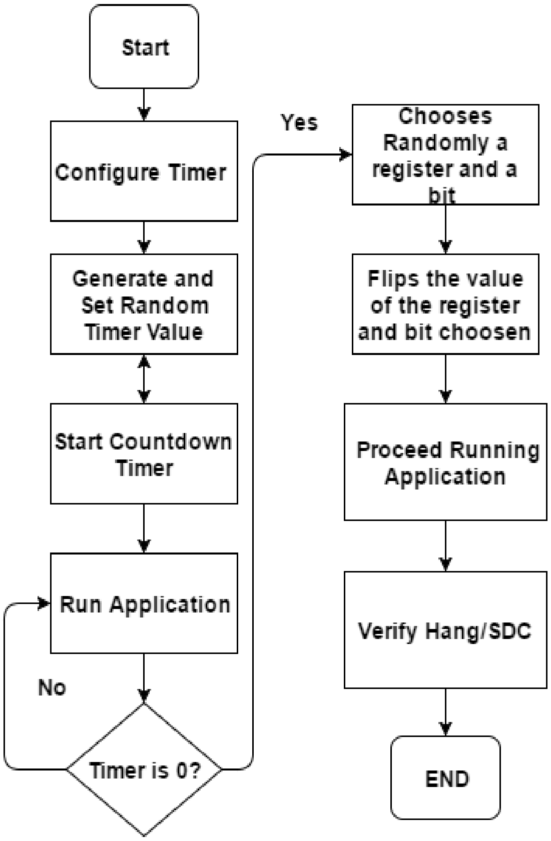
Start
Configure Timer
Generate and Set Random Timer Value
Start Countdown Timer
Run Application
No
Timer is 0?
Yes
Chooses Randomly a register and a bit
Flips the value of the register and bit choosen
Proceed Running Application
Verify Hang/SDC
END

Fig. 3: Fluxogram Fault Injector

bility of Matrix Multiplication (MxM), Advanced Encryption Standard (AES), Quicksort, Fibonacci, JPEG and FFT. **Matrix Multiplication** is an important tool for several applications such as signal and control algorithm, weather forecasting, and finite element analysis [6], [7]. **AES** is a specification for the encryption of electronic data. **Quicksort** is an algorithm that uses divide and conquer strategy to sort efficiently. **Fibonacci** is a sequence of numbers each subsequent number is the sum of the previous two. **JPEG** is a commonly used method of lossy compression for digital images, particularly for those images produced by digital photography. **FFT** is a algorithm computes the discrete Fourier transform (DFT) of a sequence. All of the algorithms are implemented in C and the inputs vectors are pre-defined.

IV. Register File Reliability Evaluation

A. *Fault Injection Results*

Applying the methodology described in Section III-B, we measure, and report in Table III, the AVF (i.e., the probability of a radiation-induced error appears in the final program output [10]) for MxM and AES (Quick Sort will be included in the final paper). We inject about $100,000$ faults per configuration (i.e., code and optimization level) .

It can be seen in Figure 1 and 2 that in the majority application the Total AVF increases between O0 and O2. However, the Total AVF decreases in most applications between O2 and O3. It is necessary to look in Table II that the execution time decreases between O0 and O2 but, when we look O2 to O3 there is most often a small decrease in the execution time.

When we look at Table II in the MWBF column we can realize that there is a trade off between the AVF and the execution time. The application with the lowest AVF or the lowest execution time will not imply that we have the lowest MWBF. When we look at the best result column we realized that not always greater optimization implies a better MWBF.

As shown in Table III, while O2 significantly impact the probability of radiation-induced SDCs on MxM (O2 increases SDC AVF for MxM of about 3.7 times), O2 and O3 have basically the same AVF (difference is lower than 10%). Furthermore, the AVF for SEFI also increases when the O2 optimization is used and decreases between O2 and O3. For AES, AVF for SDC decreases 0.75 times when using O2 and increases of 1.33 times between O2 and O3. It is worth noting that AES is a control-flow based algorithm

			Fault Injection				Radiation Experiments			
App.	Opt.	Exec. Time	AVF SDC	AVF SEFI	MWBF SDC	MWBF SEFI	σ SDC	σ SEFI	MWBF SDC	MWBF SEFI
MxM	O0	2.1 ms	0.12	0	6.83×10^4	0	$(1.16 \pm 1.50) \times 10^{-5}$	$(0 \pm 1.50) \times 10^{-5}$	1.14×10^{10}	0
	O2	1.8 ms	0.44	0.13	9.45×10^4	3.20×10^5				
	O3	0.6 ms	0.37	0.11	1.19×10^5	3.99×10^5	$(2.62 \pm 3.14) \times 10^{-5}$	$(0 \pm 3.14) \times 10^{-5}$	1.76×10^{10}	0
AES	O0	5.2 ms	0.12	0.03	1.55×10^5	6.18×10^5	$(8.93 \pm 13.4) \times 10^{-7}$	$(8.93 \pm 13.4) \times 10^{-6}$	5.96×10^{10}	5.96×10^{10}
	O2	0.8 ms	0.03	0.11	2.49×10^6	6.79×10^5				
	O3	0.4 ms	0.04	0.19	2.17×10^6	4.56×10^5	$(8.35 \pm 1.09) \times 10^{-6}$	$(2.20 \pm 2.86) \times 10^{-6}$	8.29×10^{10}	3.15×10^{11}

TABLE III: Results in Fault Injection and Radiation Experiments

that performs a filter on data. As such, AES is intrinsically less prone to experience SDCs than MxM. For O2 and O3 implementations, the SDC AVF for AES is 10 times lower than the AVF of MxM. The peculiarity of being control-flow based makes AES prone to experience SEFIs. The SEFI AVF for AES compiled with O2 and O3 is about 3.5 times the AVF for SDCs. It is interesting to notice that the AVF for SEFI increases of 3.7 times when the O2 optimization is used and still increases of 1.7 times between O2 and O3. We believe that O0 implementation of AES is too naive to produce a representative code. Eventually, this is due to high memory latency, which reduces the probability of SEFI and increases the SDC probablity. In fact, while waiting for data the ARM is in idle state, and no operation is executed. So, data is exposed and a corruption may lead to SDC, but SEFI are unlikely.

In order to better understand register criticality, we measure AVF for SDC and SEFI of each register, for both MxM and AES. Then, we can not only identify registers whose corruption is more likely to generate and SDC or a SEFI, but also if the distribution in terms of critical registers number is affected by the optimization.

AVF SDCs for each register and how they change for each optimization in MxM is shown in Figure 4. It is clear that O2 and O3 application increases the number of critical registers. This is expected, as O2 and O3 try to use all available registers to improve performances (details in Section **??**). O3 actually increase the number of critical registers in relation to 02 (10 for O2 and 13 for O3). However, 6 out of 10 registers (i.e., R0, R1, R2, R4, R9, and R10) that are critical in both O2 and O3 have a much lower AVF in O3, which is why SDC AVF for MxM is similar between O2 and O3. The SEFI AVF for each register in MxM is shown in Figure 5. O2 and O3 significantly increases SEFI AVF (in accordance with data in Table III). As for SDCs, O2 increases significantly the AVF of some registers, while O3 distribute the AVF among most of the available registers.

SDCs AVF in each register and how they change in each optimization in AES is shown in Figure 6. Interestingly, in contrast with MxM, AES O2 and O3 not only reduce the number of critical registers, but also their AVF. As said, we believe O0 for AES to be too naive, eventually masking the filter characteristic of the code. SEFIs AVF in each register in AES is shown in Figure 7. Much like MxM, O2 and O3 increase the number of critical registers as well as their distribution.

B. Heavy Ion Experiment Results

Table III reports radiation experiment results. Due to beam limitations, we were able to only test two optimizations per code. While radiation experiments inject failures in all the device, our fault injection is focused on register file. Therefore is unfeasible to make a one-to-one comparison between the two methodologies. However, data must be loaded in registers to be processed. As a result, the higher the AVF of a register, the higher the probability that a corrupted data loaded from the cache of that register will generate an output error. Thus, we can expect similar effects caused by optimizations on AVF calculated through fault injection and the cross section measured with heavy ion beam.

As reported in Table III, the SDC cross section for MxM increases 2.25 times and decreases 9.35 times from O0 to O2. MxM data is in accordance with fault injection. On the contrary, AES shows the opposite trend. We believe that the AES trend is biased by the low number of events detected, caused by the intrinsic AES SDC reliability. SEFI cross section in AES decreases 4 times, while we saw no SEFI for MxM.

Although optmizations increase register AVF and the SoC heavy ions cross section, they are beneficial, as they reduce the code execution time (reported in the third column of Table III). As a result, while the probability for one impinging particle to generate an observable error increases with optimizations, execution time is reduced, reducing the exposure time of the device. The Mean Workload Between Failures (MWBF), defined as the amount of data computed correctly before experiencing an output error, evaluates the trade-off between performances and error rate [12]. MWBF values for SDC and Crashes are reported in Table III, for both fault injection and radiation experiments. O3 is definitely the best option for MxM, as the MWBF for SDCs and SEFI increased 130% and 50% compared to O2, respectively. For AES, O2 is the best option, mainly because O2 brings a much higher execution time improvement with respect to O2 than with respect to O3. This is caused by AES intrinsic control-flow characteristic that makes O3 ineffective. It is promising that compiler optimization improves MWBF. This result attests that when modern processors are used efficiently the overall system reliability is improved. In the final paper we will analyze the reason for MWBF increase by disassembling the code and detect registers read/write variations that could be beneficial to the SoC reliability.

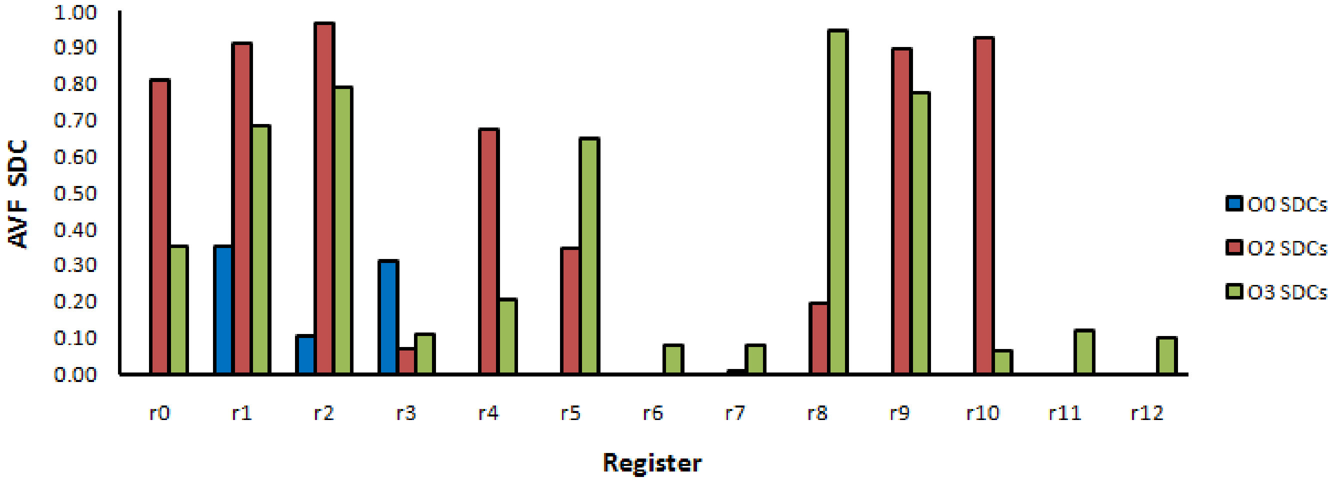

Fig. 4: MxM Architectural Vulnerability Factor SDCs

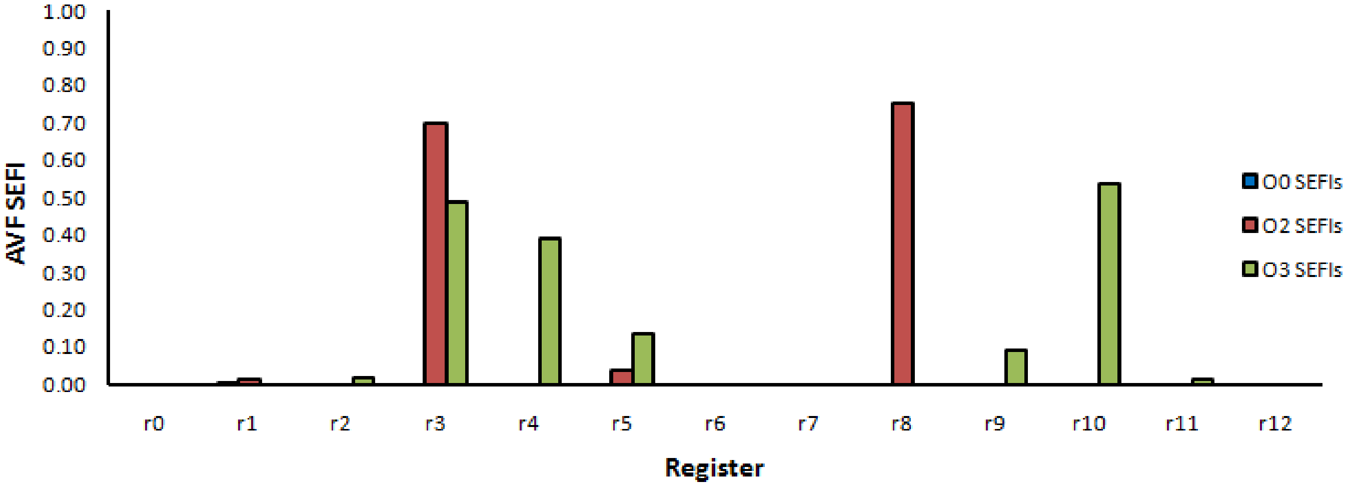

Fig. 5: MxM Architectural Vulnerability Factor SEFIs

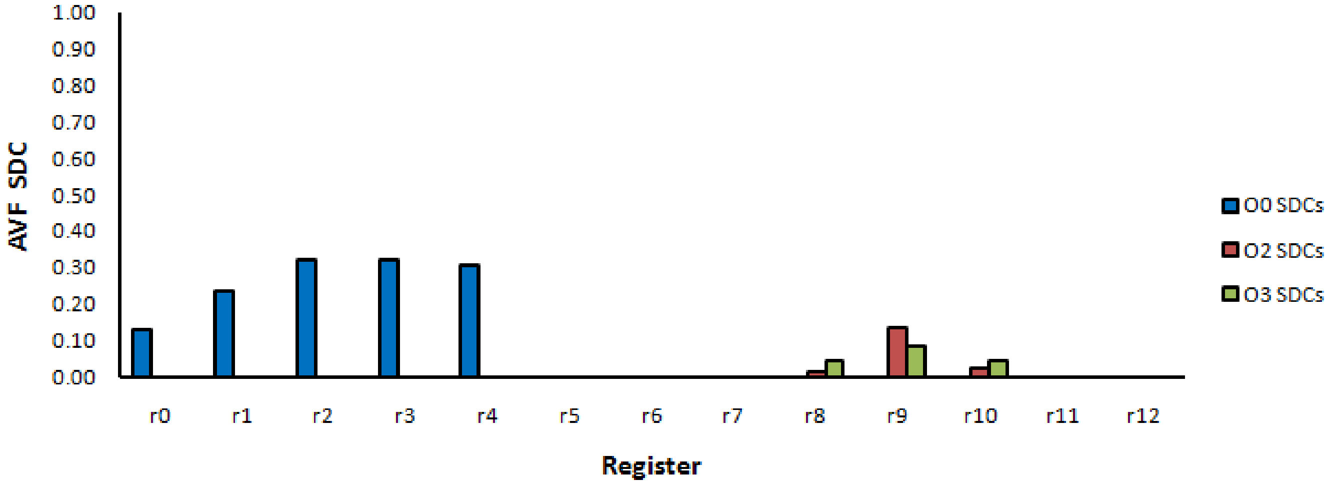

Fig. 6: AES Architectural Vulnerability Factor SDCs

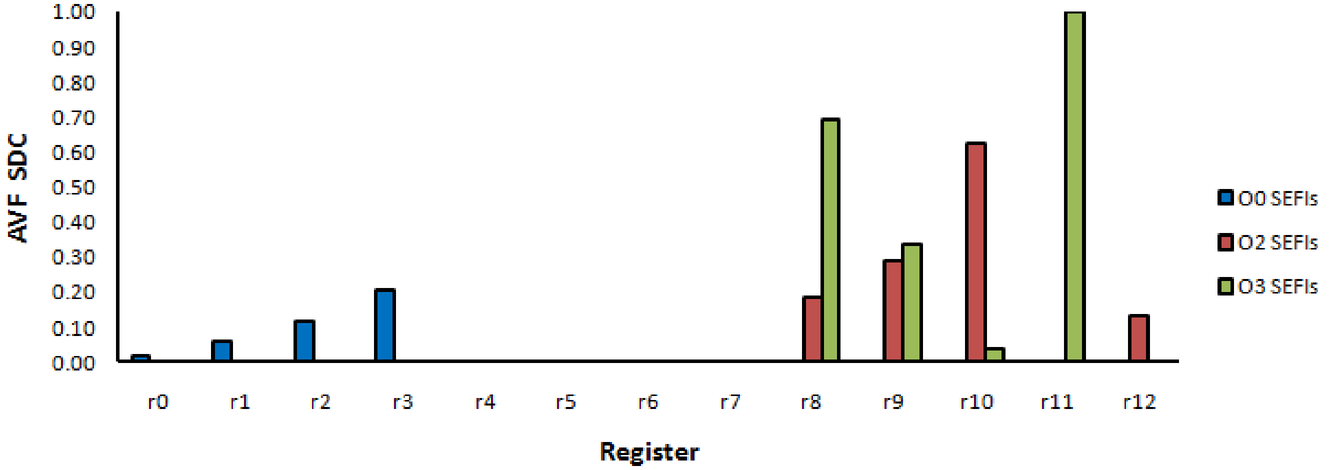

Fig. 7: AES Architectural Vulnerability Factor SEFIs

V. Conclusion

In this paper we present a fault injector to evaluate the criticality of registers and how optimizations changes that. In the future, the fault injector is going work with inject faults in caches, comparing with radiation experiments and using more benchmarks in our tests.

References

[1] V. Aguiar, N. Added, N. Medina, E. Macchione, M. Tabacniks, F. Aguirre, M. Silveira, R. Santos, and L. Seixas, "Experimental setup for single event effects at the são paulo 8ud pelletron accelerator," *Nuclear Instruments and Methods in Physics Research Section B: Beam Interactions with Materials and Atoms*, vol. 332, pp. 397–400, 2014.

[2] M. Alipour, M. E. Salehi *et al.*, "Design space exploration to find the optimum cache and register file size for embedded applications," *arXiv preprint arXiv:1205.1871*, 2012.

[3] G.-H. Asadi *et al.*, "Balancing performance and reliability in the memory hierarchy," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software, 2005*, ser. ISPASS '05. Washington, DC, USA: IEEE Computer Society, 2005.

[4] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela, and C. Lopez-Ongil, "Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection," *IEEE Transactions on Computers*, vol. 61, no. 3, pp. 313–322, 2012.

[5] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos, "Differential fault injection on microarchitectural simulators," in *Workload Characterization (IISWC), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 172–182.

[6] J. Krüger and R. Westermann, "Linear Algebra Operators for GPU Implementation of Numerical Algorithms," in *SIGGRAPH 2003*.

[7] J. Liepe, C. Barnes, E. Cule, K. Erguler, P. Kirk, T. Toni, and M. P. Stumpf, "Abc-sysbioapproximate bayesian computation in python with gpu support," *Bioinformatics*, vol. 26, no. 14, pp. 1797–1799, 2010.

[8] M. Maniatakos, N. Karimi, C. Tirumurti, A. Jas, and 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, 2011.

[9] S. S. Mukherjee *et al.*, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003.

[10] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003, p. 29.

[11] K. Parasyris, G. Tziantzoulis, C. D. Antonopoulos, and N. Bellas, "Gemfi: A fault injection tool for studying the behavior of applications on unreliable substrates," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. IEEE, 2014, pp. 622–629.

[12] P. Rech, L. L. Pilla, P. O. A. Navaux, and L. Carro, "Impact of GPUs Parallelism Management on Safety-Critical and HPC Applications Reliability," in *DSN 2014*, Atlanta, USA, 2014.

[13] F. Rosa, F. Kastensmidt, R. Reis, and L. Ost, "A fast and scalable fault injection framework to evaluate multi/many-core soft error reliability," in *2015 IEEE International Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFTS)*. IEEE, 2015, pp. 211–214.

[14] A. G. Salas, W. Attai, K. Y. Oyadomari, C. Priscal, R. S. Schimmin, O. T. Gazulla, and J. L. Wolfe, "Phonesat in-flight experience results," 2014.

[15] T. Santini, P. Rech, G. Nazar, L. Carro, and F. Rech Wagner, "Reducing embedded software radiation-induced failures through cache memories," in *Test Symposium (ETS), 2014 19th IEEE European*. IEEE, 2014, pp. 1–6.

[16] A. Serrano-Cases, F. Restrepo-Calle, S. Cuenca-Asensi, A. Mart *et al.*, "Dependability evaluation of cots microprocessors via on-chip debugging facilities," in *2016 17th Latin-American Test Symposium (LATS)*. IEEE, 2016, pp. 27–32.

[17] V. Sridharan and D. R. Kaeli, "Quantifying software vulnerability," in *Proceedings of the 2008 workshop on Radiation effects and fault tolerance in nanometer technologies*. ACM, 2008, pp. 323–328.

[18] L. A. Tambara, F. L. Kastensmidt, N. H. Medina, N. Added, V. A. Aguiar, F. Aguirre, E. L. Macchione, and M. A. Silveira, "Heavy ions induced single event upsets testing of the 28 nm xilinx zynq-7000 all programmable soc," in *Radiation Effects Data Workshop (REDW), 2015 IEEE*. IEEE, 2015, pp. 1–6.

[19] J. Tan, N. Goswami, T. Li, and X. Fu, "Analyzing soft-error vulnerability on gpgpu microarchitecture," in *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, Nov 2011, pp. 226–235.

[20] R. Velazco, S. Rezgui, and R. Ecoffet, "Predicting error rate for microprocessor-based digital architectures through ceu (code emulating upsets) injection," *IEEE Transactions on Nuclear Science*, vol. 47, no. 6, pp. 2405–2411, 2000.