Characterizing Fault Propagation in Safety-Critical Processor Designs

Jaime Espinosa[†‡], Carles Hernandez[‡], Jaume Abella[‡]
[†]Universitat Politècnica de València
[‡]Barcelona Supercomputing Center (BSC-CNS)

Abstract—**Achieving reduced time-to-market in modern electronic designs targeting safety critical applications is becoming very challenging, as these designs need to go through a certification step that introduces a non-negligible overhead in the verification and validation process. To cope with this challenge, safety-critical systems industry is demanding new tools and methodologies allowing quick and cost-effective means for robustness verification. Microarchitectural simulators have been widely used to test reliability properties in different domains but their use in the process of robustness verification remains yet to be validated against other accepted methods such as RTL or gate-level simulation. In this paper we perform fault injections in an RTL model of a processor to characterize fault propagation. The results and conclusions of this characterization will serve to devise to what extent fault injection methodologies for robustness verification using microarchitectural simulators can be employed.**

I. Introduction

Increasingly complex modern electronic designs for safety-critical markets must adhere to the strict requirements of functional safety standards. Thus, those designs have to undergo an expensive and time-consuming certification process, which is against the always stringent need to reduce the time to market. In that context, new tools and procedures have to be devised for a quick and cost-effective way to test whether robustness properties are achieved throughout the whole design flow.

Simulation-based fault injection is considered a suitable methodology for the robustness verification process, as quick and cheap corrections on misbehavior can be made. Unfortunately, such fault injection is often carried out at the gate level, and so the testing process can be excruciatingly slow and requires unduly high computing resources. When applied at a higher level of abstraction such as Register Transfer Level (RTL), the burden is reduced but it is still overwhelming for repeated use. This fact renders impractical fault injection after each design modification. If designers are intended to verify each modification, a sheer increase in simulation speed is needed while still obtaining acceptably accurate results. Considering the constraints mentioned above, microarchitectural simulators arise as one of the most promising approaches to partially cope with the increasing complexity of the verification and test process of complex systems. The main benefits of this low-cost verification step reside on the reduction of the verification time and on the ability to start the verification process long before having the RTL description of the processor, thus saving costs.

The use of microarchitectural simulators to estimate failure rate metrics is challenging as the modelled processor lacks most of the information required for accurately injecting faults. In fact, the majority of the potential injection nodes that are present at more detailed abstraction levels like RTL or gate-level are missing. Typically, fault injection experiments using microarchitectural simulators focus on the register file [8][19] and on the different memory structures [18][1]. However, these fault injection experiments, while useful to test the effectiveness of fault-tolerant capabilities and the like, are not suitable to estimate failure rate metrics as required by certification standards. For this to happen it is required to quantify how likely is that a fault present at any possible processor net, gate, or flip/flop propagates to the register file or the different memory structures.

In this paper we focus on the characterization of processor fault's behavior as a first step towards increasing the confidence on failure rate estimates given by microarchitectural simulations. In particular, we focus on how faults propagate to the different system's outputs and how many of these faults propagate to the processor register file and/or control/status registers. For the characterization of fault propagation, faults have been injected in an RTL description of the LEON3 processor [21] using simulation commands as described in [11].

Results in this paper show that even though a significant fraction of the faults originated in the core show up in the register file and/or control/status registers, injecting faults only in such registers is not enough to accurately model the impact of core faults, since a considerable amount of system failures are not preceded by any error manifestation in the mentioned registers.

The rest of the paper is organized as follows. Section II reviews the state-of-the art in fault injection in the RTL and microarchitectural simulators. Sections III and IV present the methodology used in this paper for characterizing processor's fault propagation and the results of such analysis, respectively. Finally, in Section V some conclusions are drawn.

II. Background on Simulation-based Robustness Verification

Safety-relevant systems need to go through a certification process. For instance, in automotive systems the ISO26262 functional safety standard [10] specifies the safety requirements that the different system components need to fulfil in relation with the overall system's safety. In the case of avionics systems the standard that defines the methods and tools to certify electronic products is the DO-178B [17]. Regardless of the application domain, simulation-based fault injection is a certification-friendly methodology for the safety requirements verification when analytical methods are not considered to be sufficient. This is, for instance, specified in the case of automotive systems in ISO26262 Part 5 Table 3. Note that this is the case for complex hardware components verification like a microcontroller. Fault injection through simulation can be performed using different levels of abstraction like functional, RTL, or gate-level. The current state of practice uses RTL and gate-level experiments to test hardware robustness as these methodologies have been shown to provide enough accuracy when related to the silicon level [14]. In the same way, if microarchitectural simulators are to be considered for the robustness verification process, results obtained at this level must be correlated with the ones obtained at lower levels of abstraction like the RTL or gate-level. In this paper we focus on relating RTL fault injection to experiments carried out at the microarchitectural level.

Several techniques exist to perform RTL level fault injection. A widely-used method is the injection in the HDL through simulator commands [11], which works well for most of the fault models described in the literature. In fact, some additional fault models such as those involving several injection points –short-circuit, multi-bit injection– can be applied if the more intrusive technique of saboteurs is used [2], but an instrumentation of the model –and the consequent decrease in simulation speed– is entailed.

Fault models representativeness has been validated for logic/RTL levels [6]. On the contrary, for higher abstraction levels like the microarchitectural one some works have pointed out the difficulties of correlating these results with the ones obtained at the physical level [12]. The majority of works carried out with microarchitectural simulators focus on processor's reliability estimation. Processor's reliability is estimated by the determination of the architectural vulnerability factor (AVF) [13]. The AVF is determined by the fraction of the architectural bits contributing to the processor's reliability. A similar approach is the one in [3] where the concept of instruction vulnerability factor (IVF) is proposed to evaluate how faults in every instruction affect the final application output. Likewise, in [16] the IVF is used to define a compilation process taking into account ISS reliability information. Finally, a truly existing correlation between fault injections experiments performed in an RTL processor description and the information available at the ISS was shown in [5] for the case of permanent fault models, though limited to final number of failures. Other detailed studies targeted to soft error models did not find such correlation by using single bit-flip injections in registers, suggesting other fault models should be devised for high-level injections [4].

A. Fault injection at the RTL

A circuit described at the functional level does not provide information on the internal components, but only a method to obtain outputs from inputs. Conversely, RTL description of a

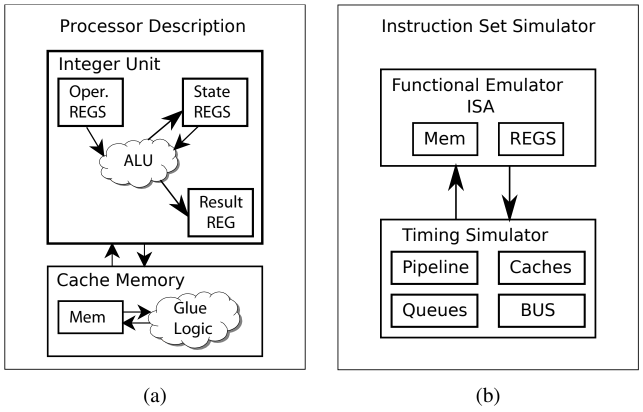


Fig. 1. (a) RTL processor description (b) Microarchitectural processor description

circuit comprises contents of registers and combinational logic, expressed in terms of logic functions and connections as shown in Figure 1(a). Specifically, the detail on the intermediate steps in terms of internal signals and operands, which allows for later synthesis of the design, renders it an ideal candidate for fault injection. Two are the main benefits:

- First, it is the lowest level –most detailed– and closest to the level where faults happen in the real system –the physical level– which, without loss of generality, achieves a good degree of representativity.
- Second, since the next level in detail –the gate level– does include the *implementation technology* in the description of the system, results of injection in RTL stay valid across different implementations, platforms, etc.

B. Fault injection at the ISS Level

Typically, a microarchitectural simulator consists of two differentiated parts: the functional emulator and the timing simulator (see Figure 1(b)). The functional emulator contains the full description of the instruction set architecture (ISA) and keeps the architectural state of the processor (i.e. architectural registers and memory data). A functional emulator is able to run application code that has been compiled for a particular architecture and to perform its execution in such a way that the memory data and architectural registers contain an exact representation of the real processor state. In other words, the functional emulator is the interpreter. The timing simulator interacts with the functional emulator and mimics with some degree of accuracy the timing behavior of the different instructions during their execution. To do so, the timing simulator models the cache memories, the processor pipeline, the register file structure, and several other queues and structures depending on the target degree of accuracy. Thus, it allows computing information like the number of execution cycles, cache hits/misses and the like. Some implementations of an ISS may have functional and timing simulation integrated, although this typically challenges their flexibility.

Therefore, fault injection in a ISS needs to be typically performed in registers and memory in the emulator, and propagation information can be obtained, including its timing, based on the event modeled by the timing simulator.

III. Characterizing Fault Propagation

As mentioned before, in this paper we want to show to what extent microarchitectural simulators can be employed in the robustness verification flow of safety-critical systems. In particular, we elaborate on the potentials of injecting faults in the processor architectural registers. The emulator part of a microarchitectural simulator only includes registers (inside the cores) and memory (outside the cores). Therefore, our view is that by characterizing which faults that reach core boundaries are reflected, either in the general purpose registers and/or in the control/status registers, we will know to what extent core faults behavior can be captured using the emulator part only.

Throughout the paper we use the common nomenclature to distinguish between faults, errors, and failures. We consider *faults* those upsets that can take place at any point of the design and are actually what is being injected; *errors*, the mismatches in the values of user registers (stored in the register file) or system registers in this case, and *failures*, the mismatches at the considered outputs. In our case, address and data buses are taken as system outputs. Note that this is the exact point at which light-lockstep cores outputs are compared for error detection purposes. Microcontrollers implementing light-lockstep compare any off-core activity (i.e., memory read/write, I/O read/write), but cannot detect faults that do not propagate outside cores . Processors implementing light-lockstep like the Infineon AURIX [9] and the STMicroelectronics SPC56XL60/54 family [20] are widely used for safety-relevant applications in the automotive domain.

We are interested in analyzing the influence of faults in the system towards the incorrect delivery of results, i.e. the appearance of failures. As studied earlier in literature, determining how faults in a system propagate through logic paths is not a straightforward task. The relationship between faults and errors depends on several factors. First of all, the actual implementation of the processor determines heavily which nodes are connected with which others, so that a path for propagation exists. Second, the system architecture according to the executed instructions and data determines the paths that are exercised and thus can propagate faults in nets to other structures and/or system outputs. Finally, in case faults are transient, the exact point in time when the fault occurs is also very relevant to have an error captured that can later potentially become a failure. In fact, the shorter the fault duration the lower the probability that it will be captured at a sequential element due to *time filtering*. In this study to facilitate the analysis we consider only permanent faults. However, we strongly believe that main conclusions drawn for permanent faults will remain also valid in the case of transient faults but the confirmation of this hypothesis is let as future work.

To characterize fault propagation we follow a methodology that consists of injecting faults in all possible nets of an RTL processor description. From the injectable nets we have excluded the register file and cache memory structures due to the following reason: errors occurring within these structures are effectively detected and/or corrected by employing redundacy mechanims (e.g., error correction codes) and this is the case in most of the processors targeting safety-critical applications [22][9]. Moreover, available nets in these structures do not realistically represent their area. Memory structures are typically implemented using SRAMs cells to miminimize area and power and the RTL includes only an instantiation of these components as a black box and/or its behavioural description.

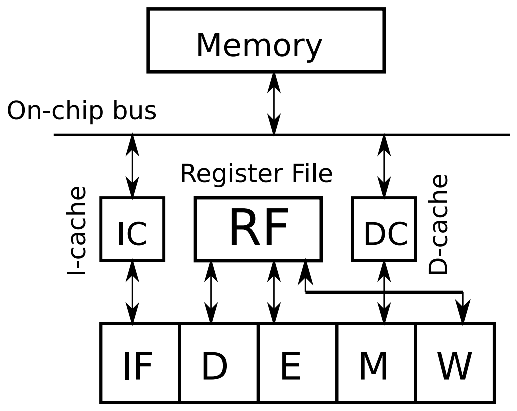


Fig. 2. Generic processor pipeline scheme. IF (instruction fetch), D (decode), E (execution), M (Memory), W (Write-back).

For every fault injection where a net has been forced to a given value, we compare the outputs of architectural registers (general purpose and control registers) and the data and address signals of the core at the on-chip boundaries to the ones obtained with a fault-free simulation.

In typical processor architectures memory operations are performed reading or writting architectural registers. Based on this, an inmediate hypothesis that can be drawn is that roughly all errors that will be visible at the on-chip bus boundaries will also be reflected in the register file and/or control/status registers. If this hypothesis is confirmed it would mean that faults in the core can be easily mimicked using microarchitectural simulators or even functional simulators as both types of simulator tools have access to the architectural register file of the processor. However, if we pay attention to a typical core pipeline implementation we realise that correlating fault injections performed at RTL nets using only the architectural registers might not be a straightforward task and, in fact, it might even be unfeasible. Figure 2 shows a schematic of a generic processor pipeline and its interface with the on-chip bus to reach main memory. Note that, while in a typical processor pipeline all memory operations are performed through writing and reading general-purpose registers, the actual implementation makes on-chip bus communication to occur through D-cache and I-cache modules in the fetch and memory stages, respectively. The previous implementation view illustrates that not all faults affecting core nets will reach the architectural registers as some of these nets have logic paths that go directly to the on-chip bus. In the next section we analyze and evaluate in detail fault propagation for the faults occuring within the core and show the exact fraction of faults that can be covered by performing error injection at the architectural registers.

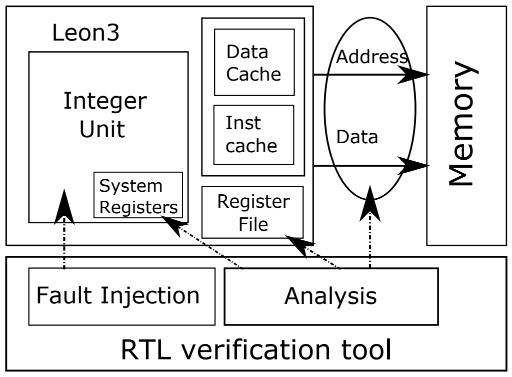


Fig. 3. RTL robustness verification framework

IV. Experimental Results

In this section we present results characterizing fault propagation in order to confirm the hypothesis presented in the previous section.

A. Experimental Setup

For our experiments we use an RTL model of a 32-bit LEON3 SparcV8 microcontroller, since it is used in the context of safety-relevant systems and both the microarchitectural simulator and the RTL description of this processor are available [21]. The LEON3 processor consists mainly of a 7-stage pipeline for integer operations (integer unit, IU) plus data and instruction caches. In this processor all instructions use all pipeline stages, since we use a minimal configuration where a floating point unit is not present. The RTL processor description follows the structural VHDL design guidelines and it models our target of injection (IU) as an entity. The test framework used in the paper is the one shown in Figure 3. Injection and analysis points in this framework are consistent with the methodology explained in Section III. To make analysis costs of register faults affordable we have used the LEON3 with a flat register file configuration[1] as this reduces the number of total registers that need to be tracked in every simulation.

The workload chosen for investigation includes programs from 2 different benchmark suites: the Mälardalen WCET group suite [7], suitable to test real-time system properties and the EEMBC Autobench suite [15], which reflects current real-world demand of some automotive CRTES. The selected programs are: a finite impulse response filter over a 700 items long sample (*fir*), a matrix multiplication of 4x4 size (*matmult*), a matrix initialization of 20 elements (*initmat*), a vehicle speed calculator (*rspeed*) and a CAN bus reader (*canrdr*).

Regarding the faultload, several permanent hardware fault models have been selected, specifically single stuck-at-1, stuck-at-0 and open line. These have been injected using simulator commands as in [11]. The campaign for each fault model and workload has consisted of one experiment per injection node in the IU (since permanent faults are applied), totaling 5,246 nodes. As the focus of the experiments is to characterize fault propagation, each experiment applies a single injection in a fixed instant: just before the execution of the main procedure, after the initialization.

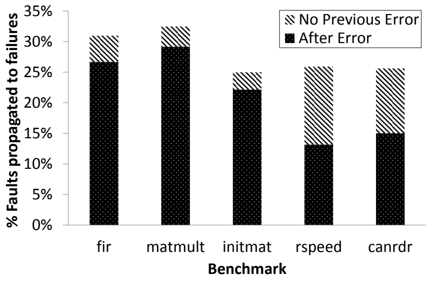
[1]Note that a typical SPARC configuration uses a windowed register file configuration with around 144 32-bit registers. Tracking the contents of 144 32-bit registers even for relatively small benchmarks is unfeasible.

B. Results

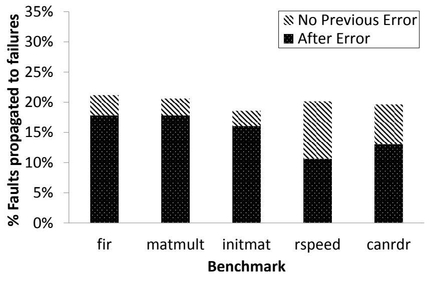
Capturing failure probability. Figure 4 shows the percentage of experiments ending in failure, broken down into those that showed a previous error –in the register file or system registers– and those that did not. This result is specially important as it provides relevant information about how accurately we can capture the behavior of faults occurring within the core pipeline by injecting errors in the architectural registers of a microarchitectural simulator. Plots in the first row (so (a), (b) and (c)) show the percentages of experiments ending in failure when faults are injected in all the nets available in the LEON3 IU. As shown in these plots a non-negligible number of failures (dashed lines) were not preceded by an error. The percentage of experiments not showing an error that ended in a failure ranges from 13% (stuck-at-1 faults in *rspeed*) to just 2% (open line faults in *matmult*). This indicates that the effect of faults within the core cannot be captured with register-based error injection solely. Furthermore, we note that in all core nets in the IU also the inputs to the Data cache and Instruction cache modules are being injected directly. To weight the impact this fact causes we remove these nets from the injection in the plots shown in the second row ((d), (e) and (f)). As expected, the number of failures not reflected in architectural register errors decreases.

In particular, the fraction of these failures decreases by around 1 to 3 percentage points with respect to the case when all nets are injected. Thus, the number of remaining failures without error is still significant. Even more important, the fraction of failures without error changes across benchmark and does not correlate with the number of failures with error. For instance, the fraction of failures without error for *rspeed* w.r.t. the failures with error or w.r.t. the total number of injections is much higher than for *initmat* for all fault types. This indicates that injecting faults only in the register file with a simulator does not provide information about failures without error and such information cannot be inferred indirectly.

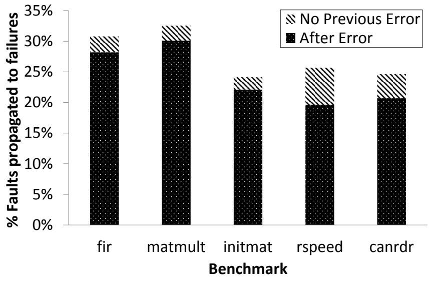
Error's profile. After injection and analysis we find the percentage of faults that are propagated to errors for each campaign. Figure 5 shows the percentage of faults that cause one or more errors for the 3 fault models considered in this study. As shown in the figure, the percentage of faults that propagate to errors is slightly superior to the percentage of faults that propagate to failures through errors – black columns in Figure 4 (a), (b) and (c) – which agrees with the expected fact that some error manifestations do not end up causing a system failure. In any case, the fraction of experiments with errors not causing system failure is always below 7%, meaning that for the selected fault models most of the bits in the exercised registers become critical.



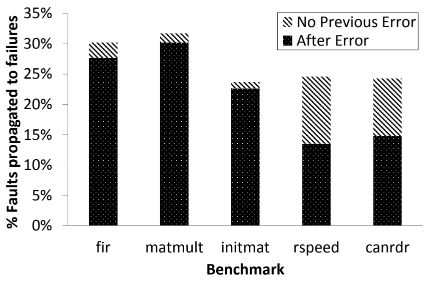
(a) Stuck-at-1



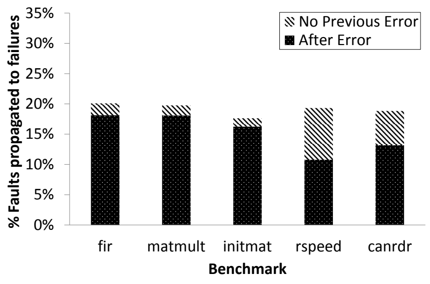
(b) Stuck-at-0



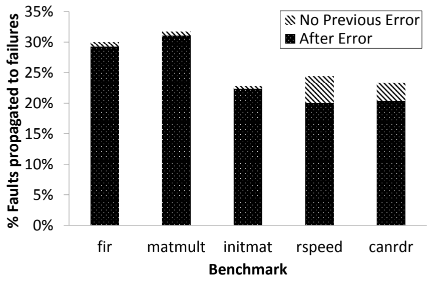
(c) Open line



(d) Stuck-at-1, No cache inputs



(e) Stuck-at-0, No cache inputs



(f) Open line, No cache inputs

Fig. 4. Percentage of failures in the experiments according to whether they caused a prior error or not.

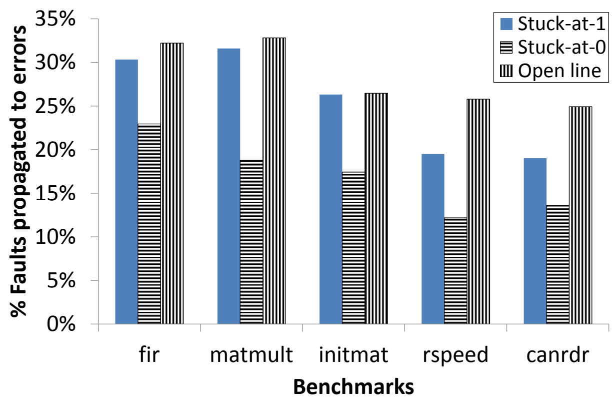


Fig. 5. Percentage of experiments which cause 1 or more errors in the registers

Figure 6 shows the error distribution across the processor architectural registers for two of the benchmarks analyzed. As expected the program counter (r.f.pc) is one of the registers that accumulates more errors. This is a consequence of two factors: (1) regardless of the benchmark executed the program counter is always severely exercised and (2) a significant number of logic paths exist between *IU* nets and the program counter. On the contrary, in the case of general purpose registers the benchmark exercised determines the exact registers to which faults are propagated. In the example of the plot *canrdr* concentrates a large fraction of the errors in two registers while in *initmat* those (several) registers frequently written during the execution accumulate a significant fraction of errors.

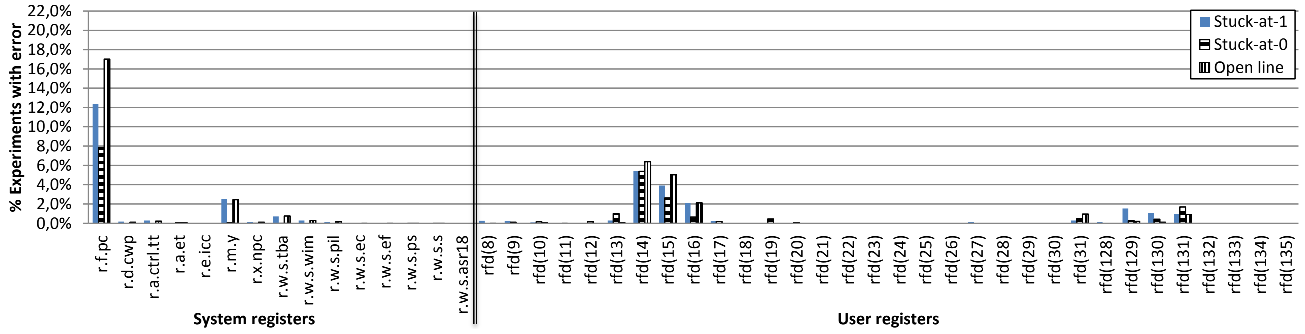
Another important conclusion that holds for the case of permanent faults is that the probability of failure does not depend on how errors reach architectural registers, i.e. how likely registers are affected by an error, but only on the type and amount of instructions that are exercised. In fact, this is in line with the work in [5] that showed that the probability of failure for permanent faults can be approached by knowing how *diverse* the set of instructions exercised by a given benchmark is.

V. Conclusions

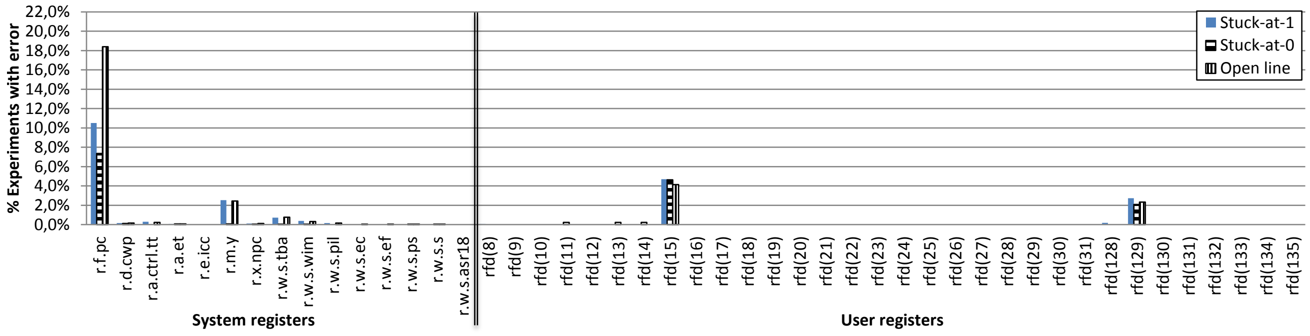
The use of microarchitectural simulators has recently arised as a promising approach to reduce the costs associated with the robustness verification of safety critical processors. However, for this low-cost simulation approach to be adopted its accuracy must be validated. In this paper we characterize fault propagation for those faults occurring within the core in order to understand to a what extent microarchitectural simulators can be used in the robustness verification process. To do so, we have injected faults in an RTL processor description and analyze the percentage of faults propagating to errors and failures.

Results in this paper show that while a significant number of faults originated within the core can be covered with verification methodologies focusing on error injection in architectural registers, the achieved coverage is not enough to provide very accurate results. A potential candidate to increase the confidence on verification methodologies using microarchitectural simulators is the use of a combined approach and perform error injection in both cache modules and architectural registers. The inmediate practical consequence of this is that functional emulators only are not sufficient to mimic the behavior of all potential core faults and thus, more detailed simulation tools like microarchitectural (timing) simulators are required. We let as future work the validation of the combined error injection approach.

While the results in this work have been obtained for a specific architecture, it includes similar features to the bulk



(a) initmat



(b) canrdr

Fig. 6. Errors distribution in system and user registers for different benchmarks

of architectures in the domain, so conclusions can be easily extrapolated to other safety-critical architectures.

Acknowledgements

The research leading to these results has received funding from the ARTEMIS Joint Undertaking VeTeSS project under grant agreement number 295311. This work has also been funded by the Ministry of Economy and Competitiveness of Spain under contract TIN2012-34557 and HiPEAC. Jaume Abella is partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

References

[1] J. Abella, E. Quiones, et al. Rvc-based time-predictable faulty caches for safety-critical systems. In *On-Line Testing Symposium (IOLTS), 2011 IEEE 17th International*, pages 25–30, July 2011.

[2] J.-C. Baraza, J. Gracia, et al. Enhancement of fault injection techniques based on the modification of vhdl code. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 16(6):693–706, June 2008.

[3] Demid Borodin and Ben H.H. Juurlink. Protective redundancy overhead reduction using instruction vulnerability factor. In *CF*, 2010.

[4] Hyungmin Cho, S. Mirkhani, et al. Quantitative evaluation of soft error injection techniques for robust system design. In *Design Automation Conference (DAC), 2013 50th ACM / IEEE*, pages 1–10, May 2013.

[5] J. Espinosa, C. Hernandez, et al. Analysis and rtl correlation of instruction set simulators for automotive microcontroller robustness verification. In *DAC*, 2015. http://people.ac.upc.edu/jabella/DAC2015BSC.pdf.

[6] Pedro Gil, Jean Arlat, et al. Fault representativeness. Technical report, DBench project, IST 2000-25425 [Online]. Available: http://www.laas.fr/DBench, 2002.

[7] Jan Gustafsson, Adam Betts, et al. The Mälardalen WCET benchmarks – past, present and future. In Björn Lisper, editor, *WCET2010*, pages 137–147, Brussels, Belgium, jul 2010. OCG.

[8] C. Hernandez and J. Abella. Live: Timely error detection in light-lockstep safety critical systems. In *DAC*, 2014.

[9] Infineon. AURIX - TriCore datasheet. highly integrated and performance optimized 32-bit microcontrollers for automotive and industrial applications, 2012. https://www.infineon.com/dgdl?folderId=db3a304412b407950112b409ae660342&fileId=db3a30431f848401011fc664882a7648.

[10] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

[11] E. Jenn, J. Arlat, et al. Fault injection into VHDL models: the mefisto tool. In *FTCS*, 1994.

[12] Man-Lap Li, P. Ramachandran, et al. Accurate microarchitecture-level fault modeling for studying hardware faults. In *HPCA*, 2009.

[13] S.S. Mukherjee, C. Weaver, et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *MICRO*, 2003.

[14] J.-H. Oetjens, N. Bannow, et al. Safety evaluation of automotive electronics using virtual prototypes: State of the art and research challenges. In *DAC*, 2014.

[15] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.

[16] S. Rehman, M. Shafique, et al. Reliable software for unreliable hardware: Embedded code generation aiming at reliability. In *CODES+ISSS*, 2011.

[17] RTCA and EUROCAE. *DO-178B / ED-12B, Software Considerations in Airborne Systems and Equipment Certification*, 1992.

[18] Daniel Sánchez, Yiannakis Sazeides, et al. Modeling the impact of permanent faults in caches. *ACM Trans. Archit. Code Optim.*, 10(4):29:1–29:23, dec 2013.

[19] B. Sangchoolie, F. Ayatolahi, et al. A study of the impact of bit-flip errors on programs compiled with different optimization levels. In *EDCC*, 2014.

[20] STMicroelectronics. *32-bit Power Architecture microcontroller for automotive SIL3/ASILD chassis and safety applications*, 2014.

[21] http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53. *Leon3 Processor*. Aeroflex Gaisler.

[22] http://www.gaisler.com/index.php/products/processors/leon3ft. *Leon3 fault-tolerant Processor*. Aeroflex Gaisler.