Analysis and RTL Correlation of Instruction Set Simulators for Automotive Microcontroller Robustness Verification

Jaime Espinosa[†], Carles Hernandez[‡], Jaume Abella[‡], David de Andres[†], Juan Carlos Ruiz[†]

[†]Universitat Politècnica de València
Valencia, Spain
{jaiesgar, ddandres, jcruizg}@disca.upv.es

[‡]Barcelona Supercomputing Center
Barcelona, Spain
{carles.hernandez, jaume.abella}@bsc.es

ABSTRACT

Increasingly complex microcontroller designs for safety-rele–vant automotive systems require the adoption of new methods and tools to enable a cost-effective verification of their robustness. In particular, costs associated to the certification against the ISO26262 safety standard must be kept low for economical reasons. In this context, simulation-based verification using instruction set simulators (ISS) arises as a promising approach to partially cope with the increasing cost of the verification process as it allows taking design decisions in early design stages when modifications can be performed quickly and with low cost. However, it remains to be proven that verification in those stages provides accurate enough information to be used in the context of automotive microcontrollers. In this paper we analyze the existing correlation between fault injection experiments in an RTL microcontroller description and the information available at the ISS to enable accurate ISS-based fault injection.

1. INTRODUCTION

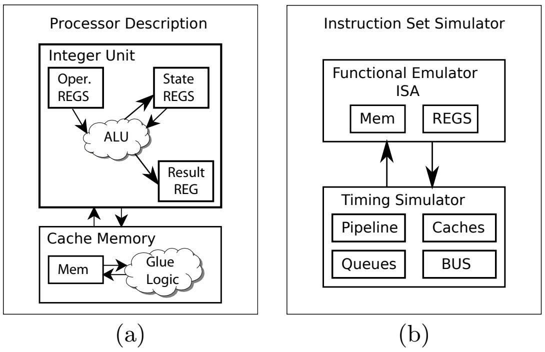
An increasing number of complex functionalities in automobiles rely on electronic components such as airbag modules, electronic parking brakes, etc [11, 19]. Thus, modern cars may include up to 100 million lines of code that need to be integrated into the least number of Electronic Control Units (ECUs) for cost contention [5]. Moreover, the amount of software in cars is expected to further increase in the future. Hence, more powerful and complex microcontrollers implemented with more integrated and less reliable technology are needed to respond to this increasing performance demand. However, hardware complexity challenges V&V processes to adhere to safety standards.

Complex and error-prone microcontrollers require the adoption of new methods and tools to enable a cost-effective robustness verification of safety-relevant systems. With the adoption of safety-related certification standards like ISO-26262 [9] in the automotive domain robustness verification has become one of the fundamental stages in the certification process for any new design. Robustness verification is carried out at different stage levels by performing intensive fault injection experiments [3]. Complex microcontroller verification challenges product design cycles, what can lead to financial loss and severe delays especially if left for the final production stages (i. e. hardware prototypes). Hence, designers have been striding to move this procedure towards the early stages of design, in order to detect design flaws or safety threats in a timely (and low-cost) manner.

Simulation-based verification has been shown to reduce costs associated with the robustness verification process as any misbehavior or defect can be corrected early. Unfortunately, simulation-based verification is often carried out at the gate level, and so the testing process is extremely time-consuming. With a higher level of abstraction such as RTL, the burden is reduced but it is still overwhelming for repeated use. This fact renders impractical fault injection after each design modification. Thus, a sheer increase in simulation speed is needed while still obtaining acceptably accurate results. Simulation-based verification using Instruction-Set Simulators (ISS) arises 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 are (1) the reduction of the verification time and (2) the ability to start the verification process long before having the RTL description of the processor, thus saving costs.

However, performing meaningful fault injection experiments using an ISS simulator is challenging as the modeled 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. For example, typical ISS-based fault injection experiments that rely on injecting faults in the register file [7][20] cannot be used to estimate failure rate metrics as required by certification standards if it is not possible to determine the probability that a given fault present at any possible microcontroller net or gate propagates to the register file.

In this paper we increase the confidence in the fault injection experiments performed with an ISS by carrying out a thorough correlation of the fault injection experiments in an RTL microcontroller description with the information available at the ISS. In particular, we propose instruction's *diversity* as a metric to enable a coarse-grain correlation of the probability that faults injected in the RTL propagate to the system outputs (i.e. the probability that a fault becomes a failure). Instruction's *diversity* is computed as the number of unique instruction types (opcodes) used by the application and represents the area the application exercises by assuming all instructions make a uniform use of microcontroller resources. Furthermore, for permanent fault models – the scope of this work – it is independent of the particular order in which instructions within this application are executed. This information is crucial to perform efficient fault



(a) (b)

Figure 1: (a) RTL processor description (b) Microarchitectural processor description

injection campaigns that simulate programs exercising only the hardware components that have been modified[1] so that impact of faults can be understood with a limited number of short simulations. While data reported does not include latent errors not manifested at off-core boundaries, mechanisms such as LiVe [7] can be used in the context of lockstep processors for safety-critical systems to enforce latent errors to manifest at off-core boundaries, where errors are detected by lockstep execution (and reported as failures in our work).

2. TOWARDS SIMULATION-BASED ROBUSTNESS VERIFICATION

In the safety-related hardware development process, fault injection is a valuable method for the verification of hardware design in the automotive domain as indicated in ISO26262 Part 5 clause 7.4.4.1 [9] for ASIL B, C and D[2]. During the development phase, simulation-based fault injection methods are typically employed instead of physical-based methods –such as injecting disturbance in power lines, electromagnetic interference (EMI), etc.– due to their repeatability, controllability and cost. Fault injection using 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 good accuracy [15]. A commonality of every simulation methodology is that it has to be related with the techniques used at silicon level for validation. For proper use of ISS to that end, these must be qualified in the same way.

2.1 Fault injection at the RTL

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

2.2 Fault injection at the ISS Level

Typically, an ISS 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.

In this paper we focus on the functional part of the ISS given that it is the highest (and so the cheapest) abstraction level. This is a necessary step to validate the suitability of an ISS for the robustness verification of safety-relevant processors. We consider little timing information (basically instructions latency). Moreover, by working mostly with the functional part of the ISS results mainly depend on the actual ISA used and remain valid for any implementation of such ISA (or the method can be ported easily). Of course, this comes at the expense of trading off some accuracy. Still, as we show later, the functional part of an ISS already provides highly-valuable information to characterize the behavior of microcontrollers in presence of faults.

2.3 ISS-based Verification

Safety-relevant systems need to go through a certification process. In automotive systems the ISO26262 functional safety standard [9] specifies the safety requirements that the different system components need to fulfill in relation with the overall system's safety. Simulation-based fault injection is one of certification-friendly methodologies for the safety requirements verification when analytical methods are not considered to be sufficient as specified in ISO26262 Part 5 Table 3. Note that this is the case for complex hardware components verification like a microcontroller. Current practice on simulation-based verification is performed at the RTL and gate-level descriptions of the circuit as these methodologies have been shown to provide good accuracy in automotive microcontrollers [1].

The use of an ISS for verification in the context of ISO26262 is challenging as the correlation of the experiments at this abstraction level with the physical level tests is not a straightforward task [12]. In this sense, a first step in that direction is to correlate with a closer level such as RTL.

Robustness verification using ISS brings several benefits

[1] Input data triggering injected faults depends on the programs used. Devising software-based tests [17] with specific coverage for the particular processor evaluated is beyond the scope of this work, so we use performance benchmarks.

[2] ASIL stands for Automotive Safety Integrity Level. There are four levels, from A to D, being D the highest one.

that can significantly contribute to the cost and complexity reduction of the verification process. We target the achievement of the following three main benefits of using ISS-based robustness verification: *(B1)* Fast simulation time, *(B2)* Detection of safety misbehavior at very early design stages of product development and *(B3)* Improvement of the hardware/software integration.

B1 speaks about the need for reducing simulation time to be able to perform the verification of increasingly complex circuits. Furthermore, increasing the simulation speed also allows the validation of more significant workloads where not only functional deviations related to safety can be detected, but also timing-related deviations [7]. Speeding up this process helps microcontroller designers evaluate the impact on safety of modifications quickly (e.g., adding new instructions). Differently, *B2* refers to the economical gain associated to the early detection of design malfunctions which is specially significant in the case of ISS-based simulation, as it does not require the actual microcontroller to be fully described. Instead, a complete definition of the ISA (or the subset of the ISA to be analyzed) suffices to perform this step. Finally, *B3* talks about the benefits of enabling ISS-based simulations in the safety-related software development. On one hand, ISS-based fault injection will help improving the modeling of the hardware/software interactions with respect to the system's safety as defined in [9]. On the other hand, as system software development relies mainly on the information available at the ISS (e.g., architectural and system registers), being able to perform meaningful fault injection experiments at the ISS level also opens the door to meaningful reliability analysis of the software components and layers long before the actual microcontroller has been deployed. Thus, software and hardware development and verification can occur in parallel to some degree, hence reducing the time-to-market, which is a key metric in the automotive domain.

3. CORRELATING RTL WITH ISS FAULT INJECTION

In this paper we consider the probability of failure P_f as the probability that a fault is propagated to off-core boundaries. We have selected off-core boundaries as the point of failure manifestation as 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 (e.g., latent faults in registers or cache memories). Microcontrollers implementing light-lockstep like the Infineon AURIX [8] and the STMicroelectronics SPC56XL60/54 family [21] are widely used for safety-relevant applications in the automotive domain.

To correlate RTL fault injection experiments with the ISS we analyze the information from the applications that are executed in the microcontroller that can be used to approximate failure manifestation probability. As the ISS decodes all instructions of the executed applications, information is available at the granularity of instructions. In this regard, we make the following hypothesis: the probability that a fault present in the microcontroller becomes a failure when executing a given set of instructions I_s is a function of the actual executed instructions I_s, their input data, and the temporal behavior of the executed instructions. Thus, $P_f = f(I_s, inputs, time)$. I_s temporal behavior includes the instruction dependences and their latency, as well as the exact point in time at which faults are present in the microcontroller. Note that our initial hypothesis about the fact that the failure probability depends on the microcontroller's *spatial* and *temporal* vulnerability, is in line with the traditional analysis of processor vulnerability factors [14] in the high-performance domain, where processors are indeed more complex than microcontrollers used in the automotive domain. In the previous P_f expression, I_s and input data determine the processor's *spatial vulnerability*, whereas the I_s temporal behavior defines the microcontroller's *temporal vulnerability.*

However, expressing P_f as a function of I_s, its input data, and I_s temporal behavior is still an overly complex function due to the value space for input data (e.g. 2^{32} different values for a 32-bit input). To reduce the problem space we consider that the data's universe can be restricted and/or upper bounded if, either we are able to introduce enough data variability, or we use corner cases for the applications' input data. Instructions temporal behavior can be captured using ISS by annotating the exact cycle at which the different instructions in a given I_s enter and leave a given microcontroller unit. However, in this paper we remove the dependence on the temporal utilization of the failure probability by focusing on permanent fault models, e.g. stuck-at-1, stuck-at-0 and open-line. We focus on permanent faults not only to remove the temporal variable but also because the number of injections to perform in every node in order to obtain significant results for transient faults is extremely high. For example, the determination of single-point fault and latent fault metrics as required by ISO26262 [9] hardware certification typically relies on the use of software-based tests (SBT) [17] and stuck-at fault models. Note that the huge execution time SBT require to achieve high coverage precludes the use of fault-models requiring very large number of fault injections.

With the assumptions above P_f can be reformulated as $P_f = f(I_s)$. This simple definition of P_f implies that the probability of an injected fault to become a failure depends on the set of instructions exercised regardless of the order in which they are executed and the particular existing dependences across instructions. In other words, our hypothesis is that the probability that a failure is triggered by a given set of instructions I_s is proportional to the processor utilization (in terms of area). This hypothesis translates the problem of determining the failure manifestation probability in the problem of determining what is the processor utilization that a given set of instructions makes.

Determining Microcontroller's Utilization. We introduce the *diversity* metric to determine the processor utilization for a given application. To relate instruction's diversity with the area exercised we consider these items:
1) The probability that a given instruction triggers a failure depends on the number of functional units a given instruction exercises. For example, all instructions have the same probability of triggering a failure at decode and fetch stages as these stages are used by every instruction [18]. On the contrary, different type of instructions, like logical and arithmetic instructions, do not necessarily use the same functional units.
2) Different functional units have different area occupation. From an RTL perspective this means that the number of fault injection points in a given functional unit is not the same for all of them and that the number of fault injection points is not necessarily proportional to the occupied

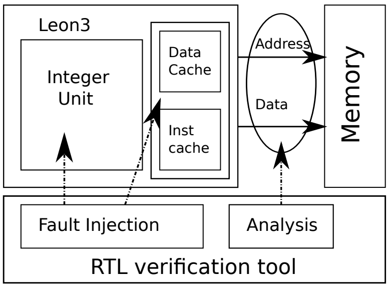


Figure 2: RTL robustness verification framework

area. The first concept speaks about the fact that in homogeneously detailed RTL representations of functional units the number of fault injection nodes is closely related to the area of a given component. The latter concept is related to the fact that heterogeneously detailed HDL descriptions of functional units lead to a decoupled relationship between injectable nodes and area occupancy.

To be able to deal with the heterogeneous processor utilization originated due to (1) and (2), *diversity* is computed for the the different functional units. Instruction's *diversity* of the m^{th} functional unit, D_m, can be computed using the ISS by dumping instructions information and finding the number of accesses to any of the available functional units for each instruction. Finally, D_m has to be related with the failure probabilities for the different processor functional units. The probability of failure of the m^{th} processor unit P_f^m can be computed using the following equation:

$$P_f = \sum_{m=1}^{N_{mod}} \alpha^m * P_f^m \qquad (1)$$

In this equation N_{mod} is the number of processor components and α^m is used to ponderate the effect of the heterogeneity in detail. α^m is in the range $[0, 1]$ and represents the fraction of the total area occupied by the processor unit m.

It is important to remark that the *diversity* metric inherently assumes that the utilization of resources within a given functional unit that instructions make is uniform.

Note that the area exercised by different instructions can be partially overlapped. Hence, executing different instruction types when few of them have been executed is likely to increase P_f, whereas executing them when many of them have been executed is less likely to increase P_f because the units accessed have been probably accessed by previous instruction types.

4. EXPERIMENTAL VALIDATION

4.1 Experimental Setup

In this section we analyze how accurately RTL fault injection experiments can be reproduced using a microcontroller ISS. To do so, we inject faults in the RTL microcontroller model and measure the percentage of injected faults propagating to failures. Any mismatch detected when writing to memory is considered a system failure. Figure 2 illustrates the fault injection methodology followed in this paper. For the analysis and correlation we use the 32-bit Leon3 sparcv8 microcontroller as both the ISS and RTL description of this circuit are available [22]. This microcontroller consists of a 7-stage pipeline for integer operations (IU). In this microcontroller all instructions use all pipeline stages. The RTL processor description follows the structural VHDL design guidelines and it models the IU and the cache memory ($CMEM$) as separate components.

	Benchmarks					
	Automotive				Synthetic	
Instructions	puwmod	canrdr	ttsprk	rspeed	membench	intbench
Total	111866	96492	96053	75058	19908	2621
Integer Unit	111862	96488	96049	75054	19908	2621
Memory	40613	33766	34905	25155	4385	19
Diversity	47	48	47	47	18	20

Table 1: Benchmarks characterization

In this study we inject faults using simulation commands as described in [10]. The choice of injected faultload is single hardware faults of permanent type, targeted to VHDL signals, ports and variables which appear at a fixed injection instant and cause either stuck-at-1, stuck-at-0 or an open line. It has been applied to all available points from the IU and CMEM microcontroller units.

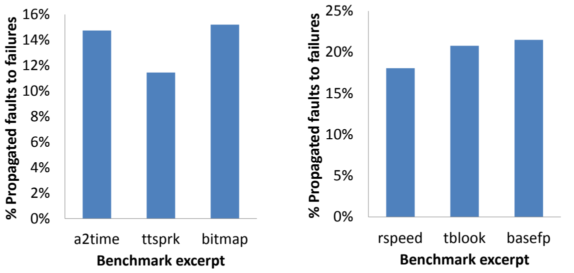
For the workload in this study we use the EEMBC Autobench suite [16] which reflects current real-world demand of some automotive CRTES and 2 synthetic benchmarks, which have been designed to use intensively memory instructions or integer instructions, and provide additional diversity values. Table 1 shows the benchmarks analyzed.

4.2 Experimental Results

In this section we proceed incrementally to validate the hypothesis made in the previous section. First, we show that the impact of inputs data variability in the probability of failure is captured for applications executing a large number of instructions. Later, we analyze the effect of the instructions temporal behavior. Finally, we show the existing correlation between the processor's utilization and the probability of failure.

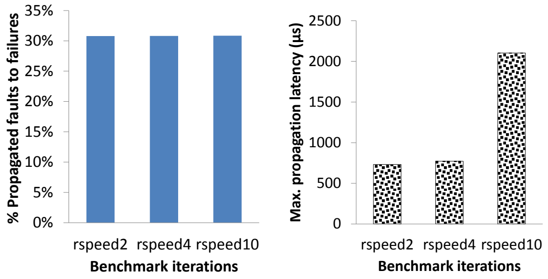
Application's data. We analyze the impact of input data variation on the probability of failure making two different experiments. For the first experiment we have injected faults in short excerpts of 2 different subsets (consisting of 3 different applications each) of EEMBC benchmarks. The selected excerpts represent the initialization phase of the benchmarks where the data to be used in the experiment are read and allocated in memory. All three applications within a subset have identical code and the only difference among them comes from the different input data they require. Each subset of applications consists of a different I_s. Figure 3 shows the effect of input data variability in the probability of failure (as I_s is fixed). Differences across benchmarks are meaningful, up to 4 percentage points (pp), so in principle the impact of data variability cannot be neglected for short applications.

We have performed a second experiment to show that input data effect can be removed when benchmarks execute a significant number of instructions. To do so, we have injected faults in the microcontroller's IU and run benchmarks with different number of iterations (2, 4 and 10 iterations). Figure 4 shows results for the *rspeed* application. As shown, P_f remains constant regardless of the executed iterations meaning that the effects of new realistic data exercised in the subsequent iterations are already included in the data space covered with 2 iterations. Further, P_f is exactly the same for the other benchmarks that use the same type of instructions. Therefore, we can conclude that for sufficiently long benchmarks, *inputs* is no longer needed in $P_f = f(I_s, inputs, time)$. Regarding fault detection latency,



(a) 8 types of instructions (b) 11 types of instructions

Figure 3: Input data variation in 2 sets of benchmark excerpts with uniform instruction types and numbers, using stuck-at-1 injections at integer unit



(a) (b)

Figure 4: Input data variation impact analyzed with 2, 4, and 10 full iterations of benchmark rspeed using stuck-at-1 injections at integer unit

maximum latency grows with the number of iterations (see plot (b)) due to those faults affecting data that is not used until the last part of the program, after the iterations, in line with the observations in [7]. Thus, 2 iterations provide the same information as 10, but allow reducing fault injection and analysis time.

Temporal Behavior. The next independent variable to clear from $P_f = f(I_s, inputs, time)$ is *time*. In the case of permanent faults one expects a fault to become a failure regardless of when it is triggered. In order to prove this, we have evaluated *ttsprk* and *puwmod* benchmarks that have exactly the same **diversity**, so they execute the same type of instructions, but they execute them in different order. As shown in Figure 5, the percentage of propagated faults for both benchmarks is roughly identical for different types of permanent faults. A different case would happen with *transient* faults, as their impact can vary greatly depending on the instructions being executed at the moment faults hit the system. We let the analysis of the impact of transient faults as future work.

Microcontroller Utilization. Finally, we check the hypothesis that the probability of failure mainly depends on the instruction set (I_s) used in the benchmark. To do so, we study the correlation between utilization of the different instructions – which relates to the spatial utilization of the microcontroller – and P_f. Furthermore, we also check that the correlation holds when applied to the IU and CMEM modules separately. We identify instruction **diversity** as the appropriate metric to determine the processor's spatial utilization.

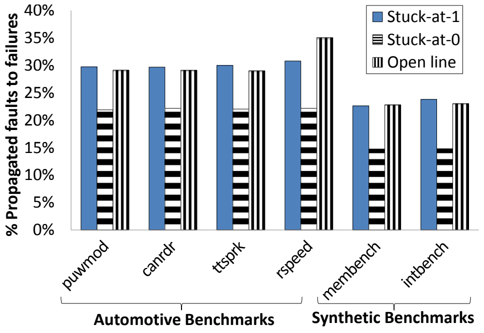


Figure 5: Fault injection experiments for different benchmarks and fault models at IU nodes.

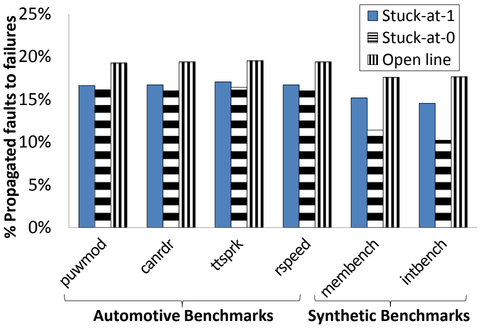


Figure 6: Fault injection experiments for different benchmarks and fault models at CMEM nodes.

Figures 5 and 6 present RTL injection results for the IU and CMEM, respectively for stuck-at-0, stuck-at-1, and open-line fault models. The first observation is that, for the automotive benchmarks, P_f is almost constant despite the fact that the executed benchmarks present different number and distribution of the executed instructions (see Table 1). However, if we pay attention to the instruction **diversity** we realize that these 4 benchmarks use almost the same number of different instructions as given by the **diversity** factor. To prove that P_f is coupled with the instruction **diversity** we also used two different synthetic benchmarks. As these benchmarks are designed to used different I_s we observe some variability in the P_f.

Finally, Figure 7 correlates P_f for the different benchmarks used in this study with the instruction **diversity**. To increase the number of points in the plot we also consider the benchmarks excerpts shown before. In these benchmarks the effect of input data variability is minimized by including the P_f value of all 3 benchmarks of each subset.

Simulation time.In order to obtain the fault injection data for the complete benchmark executions, up to 25,478 hours of computing time have been employed, distributed in 2 massively-parallel clusters and 2 powerful workstations.In contrast, less than 300 computing hours on a single workstation is enough for performing the same number of experiments with an ISS. This illustrates the importance of qualifying low-cost methods of achieving accurate results.

5. RELATED WORK

Fault injection methodologies are widely employed for the microcontrollers robustness verification in the automotive domain [15]. Fault injection experiments can be performed at several abstraction levels to exploit the existing accuracy cost trade-off [1]. RTL and gate-level fault injection experiments are the most adopted approaches to perform the certification of hardware products against certification stan-

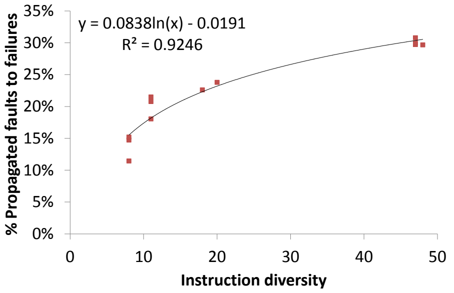


Figure 7: Propagated faults in terms of instruction diversity for the stuck-at-1 model in IU.

dards [9]. Practitioners have performed fault injection at the logic and RTL levels using different techniques. A widely-used method is the injection in the HDL through simulator commands [10], which works well for most of the fault models described in the literature. Furthermore, 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] where an instrumentation of the model – and the consequent decrease in simulation speed – is required.

Fault models representativeness was validated for logic/RTL levels [6]. For higher abstraction levels like the ISS previous work pointed out the difficulties of correlating the results with experiments at the physical level [12]. The majority of works at the ISS level focus on processor's reliability estimation, which is obtained by the determination of the architectural vulnerability factor (AVF) [14]. The AVF is determined by the fraction of the architectural bits contributing to the processor's reliability. A similar approach is the one in [4] where the concept of instruction vulnerability factor (IVF) is proposed to evaluate how faults in every instruction affect the final application output. Likewise, in [18] the IVF is used to define a compilation process taking into account ISS reliability information. An attempt of correlating ISS and logic/RTL was done in [13] focusing on the correspondence between instruction and low-level fault models. In this paper we focus on showing the correlation of results of RTL fault injection and the data available at the ISS level.

6. CONCLUSIONS

Microcontroller verification based on fault-injection is a key approach in the automotive domain, specially for the most critical functionalities as detailed in ISO26262. However, early detection of design flaws is incompatible with having a detailed description of the microcontroller such as RTL or gate-level ones. Moreover, fault injection in RTL or gate-level designs is painfully slow. Therefore, there is a need for having low-cost models of hardware that can be had at early stages of the design and provide accurate-enough information. The ISS is one of those as it is needed to allow software providers to start their developments before the hardware is ready.

In this paper we apply correlation between fault injection in the ISS and in the RTL showing that highly accurate results can be had for different permanent fault models. In the study we prove that the order of instructions in the execution and their input data are roughly irrelevant for permanent faults. Instead, the different types of instructions exercised by the benchmarks run in the ISS are the key difference towards measuring fault propagation.

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 Science and Technology 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.

7. REFERENCES

[1] ARTEMIS Joint Undertaking. *VeTeSS project: www.vetess.eu.*

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

[3] Alfredo Benso et al. *Fault Injection Techniques and Tools for Embedded Systems Reliability Evaluation.* Kluwer Academic Publishers, 2003.

[4] D. Borodin et al. Protective redundancy overhead reduction using instruction vulnerability factor. In *CF*, 2010.

[5] R.N. Charette. This car runs on code. In *IEEE Spectrum online*, 2009.

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

[7] C. Hernandez et al. Live: Timely error detection in light-lockstep safety critical systems. In *DAC*, 2014.

[8] Infineon. AURIX - TriCore datasheet. highly integrated and performance optimized 32-bit microcontrollers for automotive and industrial applications, 2012. `http://www.infineon.com/`.

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

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

[11] G. Leen et al. Expanding automotive electronic systems. *IEEE Computer*, 35(1), 2002.

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

[13] Michail Maniatakos, et al. Instruction-level impact analysis of low-level faults in a modern microprocessor controller. *IEEE Transactions on Computers*, 60(9):1260–1273, 2011.

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

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

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

[17] M. Psarakis, et al. Microprocessor software-based self-testing. *Design Test of Computers, IEEE*, 27(3):4–19, May 2010.

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

[19] S. Rohr, et al. An integrated approach to automotive safety systems. *SAE Automotive Engineering International magazine*, September 2000.

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

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

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