Exploring the Impact of Soft Errors on NoC-based Multiprocessor Systems

Felipe T. Bortolon*, Geancarlo Abich*, Sergio Bampi*, Ricardo Reis*, Fernando Moraes†, Luciano Ost‡
* UFRGS – Porto Alegre – RS – Brazil – {*ftbortolon, gabich, bampi, reis*}*@inf.ufrgs.br*
† PUCRS – Porto Alegre – RS – Brazil – {*fernando.moraes*}*@pucrs.br*
‡ University of Leicester – United Kingdom – {*luciano.ost*}*@leicester.ac.uk*

***Abstract*—Software reliability is an essential design metric in emerging large-scale multiprocessor embedded systems. Designers should identify soft error susceptibility of multiple applications executing in parallel early in the design time to ensure reliable system operation. This work proposes a non-intrusive fault injection engine that enables to conduct bespoke soft error analysis, allowing to identify and understand the soft error propagation through the processing elements (PEs). The proposed fault injection campaign evaluates the impact of soft errors considering real benchmarks in an RTL model of a distributed-memory NoC-based multiprocessor. Experiments demonstrate that 19% of soft errors are propagated to other PEs, where 31.6% of them led to erroneous computation and 58.4% to a system crash. Thus, the fault analysis must consider not only its local effect on the processor and memory but also how the fault propagates to other system components.**

***Index Terms*—Soft error, Reliability, HDL-based simulation, Multiprocessor systems.**

I. Introduction

Multiprocessor systems are widely applied in many industrial segments, including mobile, medical and automotive sectors due to their energy efficiency and performance. Such systems increase performance by employing multiple processors, which vary regarding structure and energy-efficiency. While multiple processors enhance system performance and decrease design cost, the increasing number of processor cores and internal memory cells along with the growing chip power densities and the continuous technology shrink makes underlying systems more susceptible to soft errors [1].

Aiming to access the soft error vulnerability of single [2] and multicore processors [3], simulation-based approaches have been used considering different levels of abstraction and trade-offs. While high-level approaches provide better modeling capabilities and simulation performance [3], [4] with accuracy penalties [5], hardware description language (HDL) and gate-based models have the opposite characteristics [6]–[8]. Despite the performance penalty, the latter approaches possess a higher modeling fidelity and result accuracy than the first one. In general, researchers have focused on evaluating the soft error vulnerability of single processor systems due to several reasons, including the high simulation cost and the lack of free and stable multiprocessor platforms. To enable the development of efficient soft error mitigation techniques, emerging tools and techniques must provide engineers with effective means to identify most common sources of errors and their impact on the entire system behavior. Underlying requirement is even more vital in multiprocessor systems, which have several concurrent events and race conditions, posing evaluation challenges.

Considering this context, the *goal* of this work is to present a non-intrusive fault injection engine that allows software engineers to conduct customized soft error analysis of NoC-based multiprocessor systems. The promoted engine enables to identify and understand the soft error propagation through the multiprocessor system components early in the design cycle.

The *contributions* of this paper include: (*i*) development of a portable non-intrusive Fault Injection Engine for Multiprocessor Systems; (*ii*) detailed evaluation of the impact of soft errors on a real NoC-based multiprocessor platform; (*iii*) a novel soft error propagation analysis among the system components, considering real applications.

The remaining of this paper is organized as follows. Section II presents the proposed fault injection engine. Afterwards, Section III discusses the fault injection campaign results. Section IV evaluates the error propagation across the PEs. Finally, Section V draws the conclusions and remarks.

II. Proposed Fault Injection Engine

To overcome the lack of soft error reliability analysis support for multiprocessor platforms, a non-intrusive and automated engine that enables to inject faults and evaluate their impact on underlying systems was developed. It automatically classifies the effects of a fault injection based on the behavior of available signals, e.g., interruption signal. The proposed environment is autonomous and requires little interaction with it, after its configuration.

Although developed engine enables the injection of faults (i.e. bit-flips) in memory and general micro-architecture CPU components, this work only considers the injection of single bit-flips in registers elements [9]. Section II-A describes the fault injection flow and the classification used throughout the paper. Next, Section II-B explains the hardware-independent fault injection mechanism and how interrupt signals (IRQ) aid in their classification.

A. Execution Flow

The method to simulate a fault injection campaign consists of the four-phase flow depicted in Fig. 1. Given as input an HDL design, the system kernel, and an application source code, it produces a report detailing which types of soft errors the system is most vulnerable. Note that since this work focuses on exploring the impact of soft errors on NoC-based multiprocessor systems, this Section does not discusses the platform creation, i.e., outside the dashed rectangle.

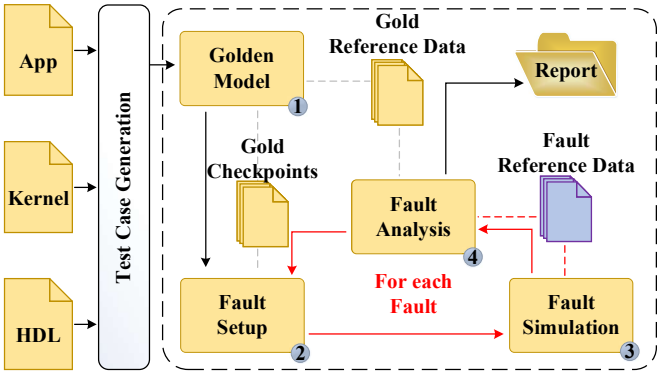

Fig. 1: Fault injection flow.

In the first phase, **Golden Model**, the target architecture is simulated in the absence of faults to extract the reference system behavior. The obtained information comprises the total simulation time, instruction count, and the states of the memory and the register bank at the moment the application finishes the execution. This information is part of the input for the last phase of the flow. Afterwards, the **Fault Setup** phase defines the fault configuration or, in other words, the fault location (register), position (register bit) and time. A random uniform function generates those parameters, which is a well-accepted fault injection technique that covers several types of failures on a system at a low computational cost [10]. The third phase, called as **Fault Simulation**, simulates the target architecture in the presence of a fault with the previously defined characteristics. This phase, collects the same design information as the first for comparison. Note that the **Fault Simulation** and the **Fault Analysis** are an interactive process. For each fault defined in the **Fault Setup** phase, one simulation is executed, and the results are analyzed.

For each fault injection campaign, the results are compared against the gold reference model, aiming to identify any misbehavior. This work adopts the Cho et al. [11] classification, which categorizes faults into five groups:

- *Vanished*, when no fault traces are left;
- *ONA* (Output not Affected), when the resulting memory is not modified, however, one or more remaining bits of the architectural state is incorrect;
- *OMM* (Output Mismatch), when the application finishes without any errors but the resulting memory is affected;
- *UT* (Unexpected Termination), when the application finishes abnormally with an error indication;
- *Hang*, when the application does not finish, and it is preempted using a timeout.

Cho's classification is employed because it provides a macro perspective of the most sensitive parts of the system at a small analysis cost. Therefore, the target multiprocessor system only needs a few interrupt signals to allow detecting these errors, and results offer to the designer insights on the hardware and software combination susceptibility.

B. Injection/Classification Engine

For several years now, Tcl has become the *de facto* standard embedded command language for Computer-Aided Design (CAD) applications, which is offered by several vendors. Accordingly, both the developed fault injection and the classification techniques rely on Tcl language. Therefore, it is important to clarify that any multiprocessor system can use developed scripts given it comply with certain aspects.

First, the registers must be accessible through the simulator tool. Second, the PE must provide some interrupt signals that the Tcl script uses to classify faults. Finally, the simulator must offer specific commands to insert faults without mutants or saboteurs such as the Mentor Graphics Questa simulator, used throughout this work.

As previously detailed, the Tcl scripts are responsible for generating the different single-bit flip configurations, inserting and classifying them. More specifically, to inject a fault, the simulator initially simulates up to the injection time, then it analyzes the victim register value and flips the target bit using the **force -deposit** command. Afterwards, the framework monitors some interrupt signals from the system to identify and classify system faults. To comply with Cho's fault organization, the designer has to specify in the scripts the location of the register bank, the memory, the program counter, and the interrupt signals to identify hardware failures, e.g., invalid memory access, arithmetic abort, etc. If the simulation completes at the expected time, without any rising IRQ, the Tcl scripts analyze both the memory and register bank from the Fault Simulation to verify whether they hold the same values as the Gold Model. The engine uses those information to classify the fault and repeats these steps until all necessary simulations complete.

Although HDL simulation offers a high precision, the time it requires to complete a fault injection campaign is impractical for large scale platforms. Therefore, the developed simulation infrastructure utilizes checkpoints to boost up the fault injection simulation. The use of checkpoints consists in collecting platform components context during the Gold Model in order to restore the appropriate context later during the fault injection campaign, reducing the amount of re-executed events.

III. Experimental Setup and Results

To validate and evaluate the proposed engine, this work adopts the public-available HeMPS platform [12]. Fig. 2 illustrates the HeMPS, a homogeneous NoC-based multiprocessor system connected to an application repository.

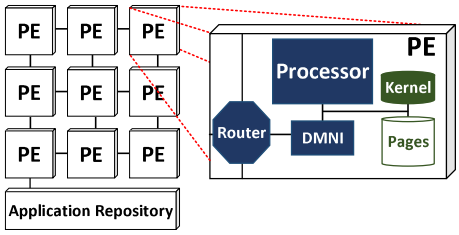

Fig. 2: Platform Example.

Each PE contains a MIPS-like processor, a DMNI (a network interface with DMA capabilities), a local memory and, a NoC router. The PEs executes a small operating system (μkernel) and may assume either manager or slave roles. A manager PE (MP) receives application requests from the

external memory (application repository) and executes task mapping. Slave PEs (SP) execute the application tasks, enabling multitasking and task communication through message passing.

A. Experimental Setup

The experiments adopt a 2x2 HeMPS platform, with four PEs. From those, one is the MP (PE0) and the remaining SPs (PE1, PE2, and PE3), which are able of executing two tasks simultaneously. The experiments consider three applications: DTW, Dijkstra, and MPEG. The MP maps the applications' tasks onto the SPs. Fig. 3 presents the task graph of each application, as well as their task mapping. Each application is evaluated separately, with the injection of a single-bit flip per simulation. Moreover, the experiments consider that faults can be injected in any of the available PEs individually.

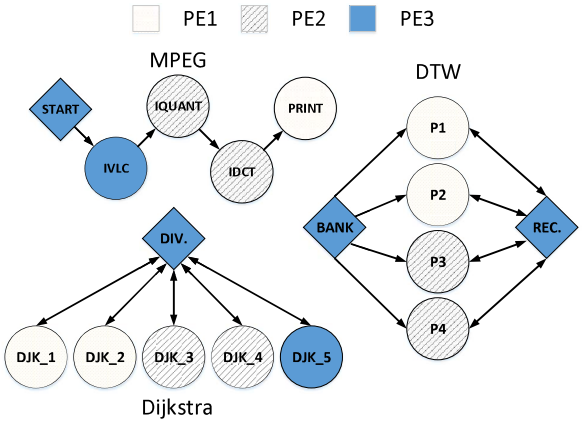

Fig. 3: Task graph and mapping (color/fill) for all applications. PE0 is the manager processor, not executing applications' tasks.

The benchmarks have distinct behaviors to ensure a broader range of results. The MPEG has intense memory accesses but small communication. On the other hand, the DTW performs significant communication between tasks in a sequential consumer-producer fashion. In turn, similar to the DTW the Dijkstra has high communication between tasks but in a distinct manner. Instead of a sequential data distribution, only one task concentrates the requests for data computation, and thus the returning result. Comparing to the MPEG, the last two do not execute as much memory write operations.

B. Results

While high-level simulation approaches can afford millions of fault injection scenarios [3], HDL-based simulation restricts the experiments to few thousands of faults, considering small multiprocessor systems. Fig. 4 presents in the Y-axis the percentage of fault occurrence and in the X-axis the PE number where a fault was injected, using 100, 500 and 1,000 simulations, evaluating the number of required fault scenarios to obtain representative results. The average difference between 1,000 simulations w.r.t 500 and 100 simulations is 3% and 9%, respectively. Thus, the simulation of 100 fault scenarios provides a good insight on the overall system behavior with reasonable simulation time and small accuracy loss.

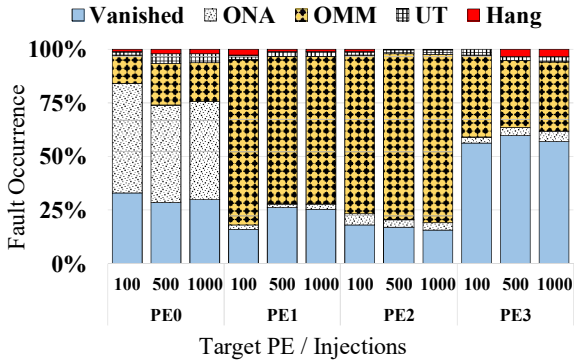

Fig. 4: Evaluation of fault occurrence precision according to the number of fault injection simulations (executed application: MPEG).

Fig. 5 presents the fault inject campaign results for DTW (I), Dijkstra (II), and MPEG (III) considering 100 fault simulations, grouped according to the PE where the fault is injected. As Fig. 3 shows, applications have distinct behaviors and mappings. Thus, the results discussion is made separately for each application.

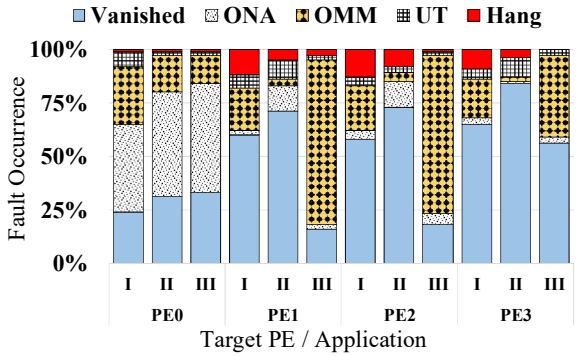

Fig. 5: Fault injection campaign results grouped according to where the fault is injected (PE) for each application: (I) DTW, (II) Dijkstra, and (III) MPEG.

The results from **MPEG** (columns III) exhibit a higher occurrence of OMMs. When faults are injected into PE1 and PE2, which execute two similar tasks with an intense memory write access, the occurrence of OMMs reaches 77% and 74% respectively. Nevertheless, that number reduces to 38% in PE3, which executes a single memory intensive task (*IVLC*) and a lightweight function without data dependence (*start*), used for synchronization purpose. In this case, the number of OMM decreased by half and vanished faults increased by 3.5 times.

In turn, the **DTW** application (column I) has higher communication workload and fewer write operations when compared to the MPEG. Accordingly, the number of OMM faults severely reduces to 4%, but the Hang occurrences rise to 13%, for PE1, PE2, and PE3. The reason for this behavior is that the message passing interface is vulnerable to packet modifications, which might cause a PE to wait indefinitely for a packet. Nonetheless, packet corruption is not as frequent as memory accesses in MPEG, and thus not as likely to generate faults.

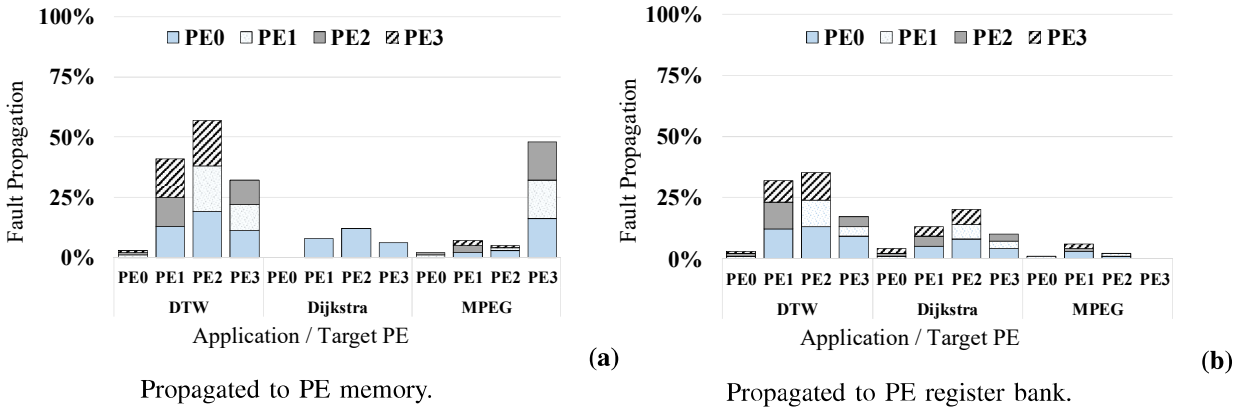

Fig. 6: Fault propagation to other PEs when injecting faults at a target PE.

The **Dijkstra** application (column II), has similar characteristics to the DTW but with different a communication scheme. While the DTW has one task to divide the computation and gather results, the Dijkstra has only one for both functions (Fig. 3). This difference means that PE3 can execute two tasks with different profiles and thus reducing the occurrence of Hang failures by 7%. Similar to the MPEG, results demonstrate that mapping tasks with different functionality to the same PE can attenuate a particular type of soft error.

Finally, under the presence of faults, the manager processor (PE0) exhibits a different behavior than SPs because it performs specific management functions, which demand additional operations (e.g. access to the application repository). Therefore, the behavior of faults injected in PE0 is similar to the remaining PEs since they execute about the same μkernel. Results show a minor variation for the OMM faults in the DTW scenario. This variation is due to the fact that DTW has two initial tasks, while MPEG and Dijkstra have one, and a higher number of interactions with other tasks. Nonetheless, the behavior of fault injects in μkernel is similar to all applications.

This set of experiments demonstrate the effectiveness of the proposed fault injection engine, which enables a detailed evaluation of the effect of faults on different applications mapped onto a NoC-based multiprocessor system. The experiments also revealed the task mapping impact on the system resilience, i.e., it is possible to reduce the soft-error occurrence depending on the mapping strategy. The next section evaluates fault propagation and its effect on the operation of other PEs.

IV. Fault Propagation

In multiprocessor platforms, it is desirable to evaluate not only the impact of a fault on an individual PE, but whether this fault is propagated to other PEs as well. This paper defines an error as propagated when it meets the following definitions.

Definition 1. An error is considered as propagated when a fault injected into a PE affects the memory and/or the register bank of another PE.

Definition 2. At the end of an application execution, a memory or register bank is considered as affected when it stores a different value from the fault-free system. This is only valid when either the application (*i*) terminated spontaneously or (*i*) it executed indefinitely.

The second definition excludes the cases when the simulation terminates unexpectedly and hence has not given enough time for the system to stabilize. Fig. 6 presents the fault propagation (FP) for all applications separating affected entity. Overall, DTW is the application that propagates more faults to other PE memories and register banks. The MPEG presents the lowest FP rate on the register bank (Fig. 6.b) because it contains lesser data exchange, reducing the chance to affect the workload output. However, MPEG reaches almost 50% of FP on memory (Fig. 6.a) for PE3, which is justified by the memory intensive nature of the (*IVLC*) task. This high FP rate is explained because a fault (i.e., bit flipped error), in most of the cases, is propagated whenever the first access to an affected data is a read operation. It is evident that the FP rate increases when faults are injected into PEs that are executing initial tasks. The PE0 presents the lower FP rate because it executes only lightweight management functions responsible for sending and receiving small control packets and application object codes. Further, results also demonstrate that of up to 90% of propagated errors, led to either erroneous computation or a system crash operation.

Conducted experiments provide comprehensive determination of whether faults propagate and impact system operation, demonstrating the importance of providing engines with appropriate means that are able to identify not only the occurrence but also the system characteristics that contribute to the error propagation (e.g., application flow, data dependence, task mapping).

V. Conclusions

This paper developed a portable and non-intrusive fault injection engine for multiprocessor systems. Moreover, it proposed a methodology to evaluate fault propagation that allows a more detailed analysis of MPSoCs. The proposed engine and method demonstrate the importance of early evaluation of soft errors in design time and that 19% of SEUs injected into a single core propagated to others in the worst case for the applications considered herein.

REFERENCES

[1] R. C. Baumann, "Radiation-induced soft errors in advanced semiconductor technologies," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 305–316, Sept 2005.

[2] W. Mansour and R. Velazco, "An Automated SEU Fault-Injection Method and Tool for HDL-Based Designs," *IEEE Transactions on Nuclear Science*, vol. 60, no. 4, pp. 2728–2733, Aug 2013.

[3] F. Rosa, L. Ost, R. Reis, S. Davidmann, and L. Lapides, "Evaluation of multicore systems soft error reliability using virtual platforms," in *NEWCAS*, 2017, pp. 85–88.

[4] M. Kaliorakis, S. Tselonis, A. Chatzidimitriou, N. Foutris, and D. Gizopoulos, "Differential Fault Injection on Microarchitectural Simulators," in *IISWC*, 2015, pp. 172–182.

[5] T. Flenker, J. Malburg, G. Fey, S. Avramenko, M. Violante, and M. S. Reorda, "Towards Making Fault Injection on Abstract Models a More Accurate Tool for Predicting RT-Level Effects," in *ISVLSI*, 2017, pp. 533–538.

[6] J. C. Baraza, J. Gracia, D. Gil, and P. J. Gil, "A prototype of a VHDL-based fault injection tool," in *DFTVS*, 2000, pp. 396–404.

[7] 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 *MICRO*, 2003, pp. 29–40.

[8] M. Solinas, A. Coelho, J. A. Fraire, N. E. Zergainoh, P. A. Ferreyra, and R. Velazco, "Preliminary results of NETFI-2: An automatic method for fault injection on HDL-based designs," in *LATS*, 2017.

[9] F. M. Lins, L. A. Tambara, F. L. Kastensmidt, and P. Rech, "Register File Criticality and Compiler Optimization Effects on Embedded Microprocessor Reliability," *IEEE Transactions on Nuclear Science*, vol. 64, no. 8, pp. 2179–2187, Aug 2017.

[10] S. Feng, S. Gupta, A. Ansari, and S. Mahlke, "Shoestring: Probabilistic Soft Error Reliability on the Cheap," in *ASPLOS*, 2010, pp. 385–396.

[11] H. Cho, S. Mirkhani, C. Y. Cher, J. A. Abraham, and S. Mitra, "Quantitative evaluation of soft error injection techniques for robust system design," in *DAC*, 2013, pp. 1–10.

[12] E. A. Carara, R. P. de Oliveira, N. L. V. Calazans, and F. G. Moraes, "HeMPS - a framework for NoC-based MPSoC generation," in *ISCAS*, 2009, pp. 1345–1348.