Cache- and Register-aware System Reliability Evaluation based on Data Lifetime Analysis

Maha Kooli, Firas Kaddachi, Giorgio Di Natale, Alberto Bosio
Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier (LIRMM), France
name.surname@lirmm.fr

***Abstract*—Developing new methods to evaluate the software reliability in an early design stage of the system can save the design costs and efforts, and will positively impact product time-to-market. This paper introduces a new approach to evaluate, at early design phase, the reliability of a computing system running a software. The approach can be used when the hardware architecture is not completely defined yet.**

In order to be independent of the hardware architecture and at the same time accurate, we propose to use the Low-Level Virtual Machine (LLVM) framework. In addition, to reduce the reliability evaluation time, our approach consists in analyzing the variable lifetimes to compute the probability of masked faults. Finally, to achieve a better characterization we propose to consider also the presence of caches and register files. For this purpose, a cache emulator as well as a register file emulator are developed. Simulations run with our approach produce very similar results to those run with a hardware-based fault injector. This proves the accuracy of our approach to evaluate system reliability with a gain in the simulation time and without requiring a hardware platform.

***Index Terms*—Reliability, Hardware Faults, Lifetime, LLVM, Data Cache, RAM**

I. Introduction

System reliability has become an important design aspect for computer systems due to the aggressive technology miniaturization, which introduces a large set of different failure sources for hardware components [1] [2] [3]. The hardware system can be affected by faults caused by physical manufacturing defects, environmental perturbations (*e.g.*, radiations, electromagnetic interference), or aging-related phenomena [4]. Faults propagate through the different hardware structures composing the full system, as shown in Fig. 1. However, they can be masked during this propagation either at the technological or at architectural level [5] [3]. When a fault reaches the software layer of the system, it can corrupt data, instructions or the control flow. These errors may impact the correct software execution by producing erroneous results or prevent the execution of the application leading to abnormal termination or application hang. The software stack can play an important role in masking errors, which enables the improvement of the system reliability. In this work we investigate the role of the software and its impact on the overall system reliability in a very early design stage of the system, *i.e.*, when the hardware architecture is possibly not yet fully defined.

In order to evaluate the reliability in such early design stage, we need to investigate methods and tools to describe the software independently from the target hardware architecture.

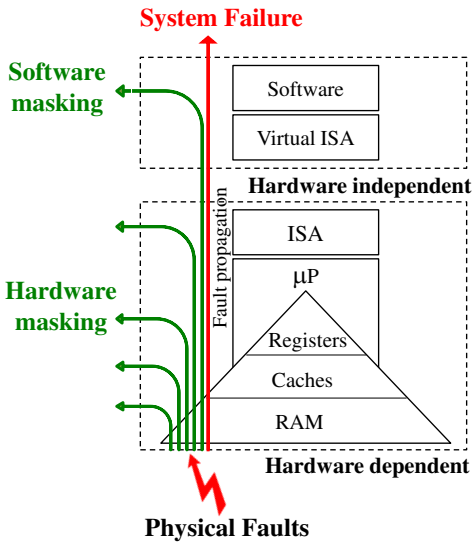

Fig. 1: System Layers and Fault Propagation

At this stage, the Instruction Set Architecture (ISA) might be unknown and therefore cannot be used to perform simulations to analyze how faults propagate through the software modules. A possible solution is to use virtualization techniques to abstract the ISA. The virtualization concept ensures the possibility to make analysis without previous knowledge of the ISA. Different alternatives of virtual environment implementing virtual ISA are available in the literature [6] [7] [8]. LLVM [8] is a compiler framework that uses virtualization with virtual instruction sets to perform complex analysis of software applications on different architectures. LLVM uses the intermediate representation as a form to represent code in the compiler. This representation is similar to an assembly code and independent from the source language and the target machine.

In this paper, we introduce a new approach to evaluate the system reliability without performing a long fault injection campaign, as usually used by existing approaches in the literature [9]. The method we propose allows evaluating the outcome of the software when a single fault affects its data. We use the concept of the variable lifetime and the variable residence to compute the percentage of masked faults. To be accurate, we propose to use the LLVM virtual ISA. Further-

978-1-4673-8454-4/16/$31.00 ©2016 IEEE

more, to achieve a better characterization, we must follow a holistic approach when targeting the software layer. Thus we take into account the presence of the cache and the register files. Since we are working at a high level, where the concept of data cache is not modeled, we build a system emulator to represent a simplified model of a memory subsystem. We emulate the behavior of different system components: the RAM, the data cache, and the register files.

The main advantages of the proposed approach are: (i) it does not require a fully predefined hardware, and (ii) it significantly reduces the simulation time compared to the standard hardware reliability evaluation techniques. To validate our approach, we compare the results to those of an FPGA-based fault injection tool using the microprocessor LEON3 [10]. The similarity of the results proves the effectiveness and the efficiency of our approach.

The rest of the paper is organized as follows. Section II summarizes related works. Section III introduces the proposed approach. Section IV presents the experiments and the results. Section V concludes the paper.

II. Related Work

In this section, we report related work existing in the literature. In subsection II-A, we introduce the fault injection techniques. In subsection II-B, we present the reliability evaluation tools based on LLVM. In subsection II-C, we present existing methods that use the concept of lifetime analysis.

A. Fault Injection

Fault injection is a widely used technique to evaluate the system reliability [9]. It is based on performing controlled experiments in order to observe the system behavior in the presence of faults. Two main classifications of fault injection techniques exist in the literature: hardware-based techniques that directly inject faults in the target hardware, and software-based techniques that model the hardware fault at an abstract level. The hardware-based techniques [11] perform fault injection campaigns in more realistic conditions and provide therefore more accurate results. The software-based techniques [12] [13] provide cheaper solutions to evaluate the reliability of the whole system with sufficient accuracy levels. In general, the fault injection techniques are expensive in term of simulation time and energy consumption because they require computing a big number of simulations (up to 10K) for the same application. Our approach provides a better solution to evaluate the fault effect by performing only one program execution.

B. LLVM-based Evaluation Tools

The reliability evaluation tools based on LLVM use the fault injection method to simulate hardware faults [14]. Thomas et al. [15] develop LLFI, an LLVM-based fault injector permitting to inject transient faults into the processor's computation units. The tool is used to map fault outcomes back to the source code, and understand the relationship between program characteristics and the various types of fault outcomes. Sharma et al. [16] develop KULFI, another LLVM-based fault injector permitting to inject single bit flips into the instructions as well as in the data/address registers. KULFI simulates faults occurring within the CPU and provides a finer control over the fault-injection process.

The limitation of the previous tools is that they are still time-consuming and they do not target memory components such as caches or RAM. We target in our approach the data cache and the data in the RAM.

C. Lifetime Analysis

Mukherjee et al. [3] use lifetime analysis to compute the Architectural Vulnerability Factor (AVF) of the instruction queue and execution units. Biswas et al. [17] apply the lifetime concept to compute the AVF of the data cache, the data translation buffer, and the store buffer. Montesinos et al. [18] use the register lifetime to propose a technique that protects register files against soft errors.

While the previous techniques compute lifetime analysis on a physical processor, our approach uses a virtual ISA. Besides, the proposed techniques in [3] target faults in instructions, in [17] target faults in only some address-based structures, and in [18] target only register files. In our approach, we target the data in different components of the system (RAM, data cache and register files).

III. Proposed Approach

In this section, we present our approach. We explain the concept of lifetime analysis and variable residence. Then we introduce our system emulator and the considered fault classifications.

A. Lifetime Analysis

During the program execution, a variable can be read or written. The variable is alive from the first write (followed by a read) to the last read (before the next write or the ending of the program), otherwise it is dead. When a variable is dead, its content is irrelevant to the correct program execution since it will be either re-written or never used again. Thus any fault affecting this variable will be masked.

In Fig. 2, we present the steps of the lifetime computation. Starting from the original source code written in any programming language (or possibly the binary code), we generate the corresponding LLVM code using the LLVM compiler. Since LLVM does not provide information about the exact timing when an instruction is executed, we instrument the original code by adding the information of the current clock cycle of the executed instruction (we consider that each LLVM instruction is executed in one clock cycle). In particular, we use a counter that is incremented after each instruction. Then we execute the program and we record a trace that contains information about each *write* (*'store'* or *'alloca'* instruction) and *read* (*'load'* instruction) operation computed by the program. We record, for each read/write operation, the corresponding clock cycle, the physical address of the variable, the operation type (read or write), and the variable size. Once

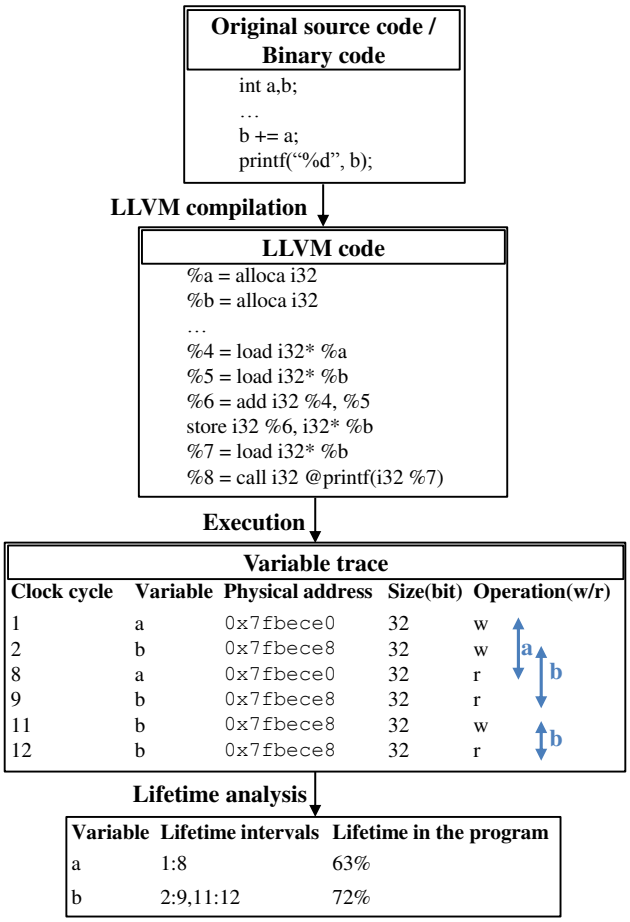

Fig. 2: Lifetime Computation

we have the trace, we calculate the cumulative number of clock cycles in which the variable was alive. Divided by the total program clock cycles, this corresponds to the lifetime of the variable in the program. Clearly, the lifetime of the program variables is depending from the used workload.

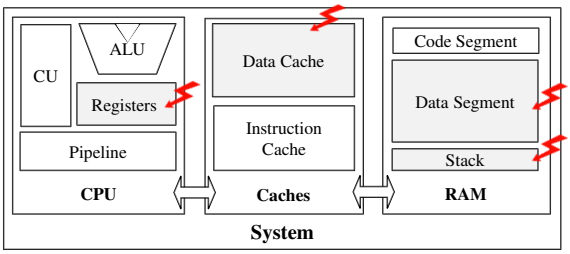

Fig. 3: Data Location in System

B. Variable Residence

In modern microprocessors, the concept of data caches is introduced to store data in order to accelerate their future requests by the processor. To be cost-effective and to enable efficient use of data, the caches are relatively small compared to the RAM. This cache-based architecture introduces a sort of hardware redundancy for increasing performances. This means that the same variable can have, at the same time, several copies in different locations. It can reside in the RAM (the data segment or the stack), the data cache and/or the CPU (the registers), as shown in Fig. 3. While at the software level it is the same variable, from the hardware point of view only one copy is active in the program and influences the execution. Therefore, a fault affecting one of the non active copies might be masked. Thus to compute the percentage of masked faults, we take into account the variable residence. In addition to that, since a fault in the data can affect any of the memory units containing data as shown in Fig. 3, we also consider in our computation the target component (*i.e.* where the fault occurs).

To determine the variable residence, we require information about the program execution and the memory units. In our analysis, we want to target a hardware-independent level. At this level, the concepts of data cache and register files are not defined. We developed a memory subsystem emulator representing a simplified model of the actual system. The concept of system emulator is explained in the next subsection.

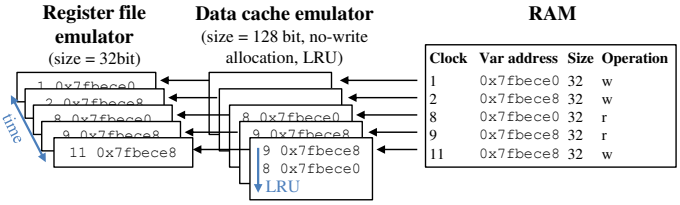

Fig. 4: System Emulator

C. System Emulator

As presented in Fig. 4, the system emulator considers three main units: the RAM, the data cache and the register files. While in this paper we focus on a single cache layer, the proposed method can be straightforwardly scaled to multiple levels of cache. The structure of each unit is designed in a way to be: (i) as close as possible to a real system behavior, and (i) as generic as possible to support different characteristics of different microprocessors. For each component, we require some hardware configurations to be given by the user as input to the tool. These parameters are the only link of our approach with the hardware characteristics. In this paper, we consider the following:

- The RAM contains all the active variables used during the program execution. As input, we require the RAM size.
- The data cache contains, for each clock cycle, a set of the recently used variables for future request. As input, we need the data-cache size, its write-miss policy (write allocation or no write allocation), its write-hit policy (write through or write back) and its replacement strategy (Least Recently Used (LRU), Least Recently Replaced (LRR), random).
- The register files contain, for each clock cycle, a small set of the recently used variables by the program. As

input, we need the size of the register files where data are stored.

During the program execution, we build for each clock cycle the content of each component, as shown in Fig. 4. The data-cache emulator and the register-file emulator are updated for each clock cycle when a variable has either a write or a read operation.

The data-cache emulator is implemented as a priority queue. We present in Fig. 5 the algorithm we follow for its construction. Whenever a variable is written or read, if write-allocation (or only read if no-write allocation), either we update its position to the head, if already in the cache, or we add it, if not. The process of adding variables depends on the state of the cache at the current clock cycle and the replacement policy. If the cache has enough space for the new variable, it is added to the head. Otherwise we keep deleting existing variables with respect to the replacement mechanism, till we find enough space for the new variable.

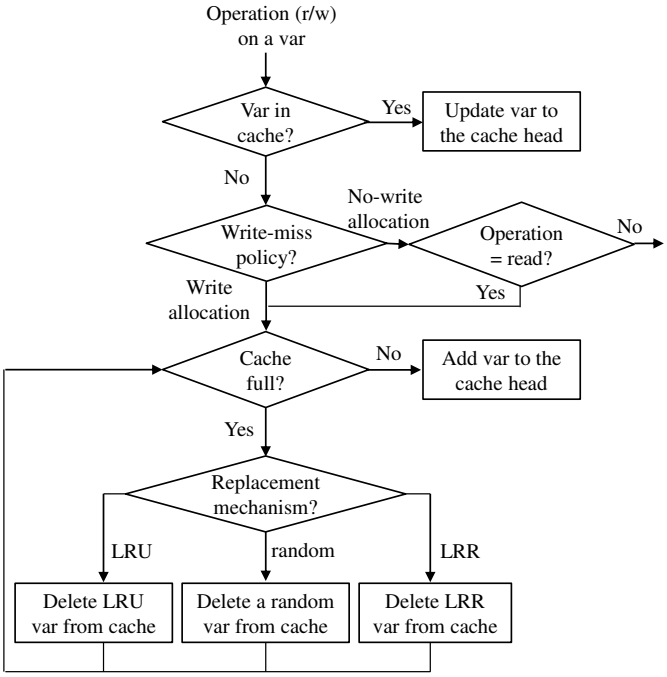

Fig. 5: Cache Building

The register-file emulator is also implemented as a priority queue. The process of adding variables is simpler. It follows the algorithm presented in Fig. 6. If the variable is already in the registers, we simply update it to the head. In the other case, the variable is added based on the LRU replacement mechanism.

D. Fault Classification

Once we have the variable residence and the variable lifetime in the program, we classify the fault outcomes into masked, i.e. the program terminates correctly, or failure, i.e. the fault affects either the program outputs (erroneous results) or the program execution (crash or hang).

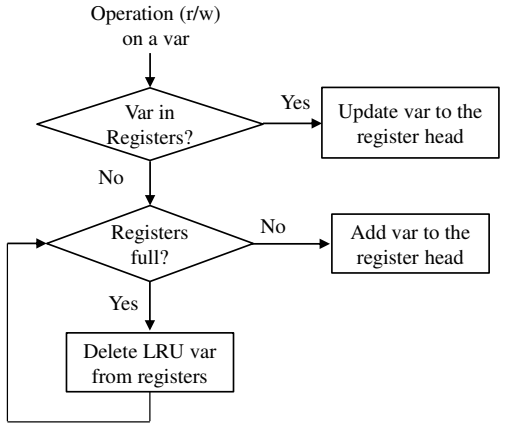

Fig. 6: Register-File Building

1) Fault Classification in the RAM: In order to classify the faults occurring on data in the RAM, we consider, for each clock cycle, the variable lifetime in the program and its residence in the data cache. We use the following assumptions:

- A fault affecting a variable that (i) is living in the program and (ii) will be reloaded to the data cache before its death, results to a failure, as shown in Fig. 7.
- A fault affecting a variable that does not satisfy the previous condition is masked.

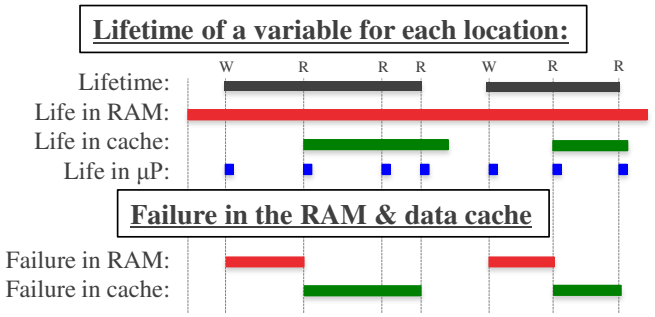

Fig. 7: Lifetime Analysis based on the Variable Residence and the Variable Lifetime in the Program (cache policy: write though with no-write allocation).

2) Fault Classification in the data cache: In order to classify the faults occurring in the data cache, for each clock cycle, we consider the variable lifetime in the program and its residence in both the data cache and the register files. We use the following assumptions:

- A fault affecting a variable that (i) is living in the program, (ii) resides in the data cache and not in the register, and (iii) will be used (read/written) before leaving the data cache, results to a failure, as shown in Fig. 7.
- A fault affecting a variable that is (i) living in the program, (ii) residing in the data cache and in the registers, and (iii) will be reloaded to the register files before

leaving the data cache, results also to a failure, as shown in Fig. 7.
- A fault affecting a variable that does not satisfy the previous conditions is masked.

The computed percentage of masked faults corresponds to the minimum of masked faults in the program. In fact, the program can contain techniques of software fault tolerance, such as software masking (*e.g.*, C=AxB, if B=0 then any fault in A will be masked) or redundancy (*e.g.*, data duplication or triplication). Any faults occurring in a variable protected by such techniques is masked, while the lifetime analysis would consider them as failures. Thus, we enhance our tool by adding a configuration file where the user can state, for each variable, if there is a fault tolerance mechanism implemented at software level. In particular, it is possible to provide the list of variables either protected or detected by the software itself. Then in our analysis we compute that any fault occurring in one of these variables is masked in case of protection, or detected in case of detection.

IV. Experiments and Results

In this section, we present the results obtained by applying our approach on different benchmarks and we compare them to the results of an FPGA-based fault injector.

A. Target Benchmarks

In order to evaluate our approach, we set up a list of benchmarks on which we run simulations. The target benchmarks have different execution times and memory utilization, and cover both data-intensive and control-intensive algorithms. We use a matrix-multiplication program with a 50x50 integer array. We also select a set of workloads from the open-source benchmark suite MiBench [19] (bit count, quick sort, string search, fft, crc 32).

B. FPGA-based Fault Injector

To validate our approach, we use a hardware-based fault injector. Such tools are considered in the literature as accurate techniques to evaluate system reliability.

We use SCFIT, an FPGA-based fault injector proposed by Ebrahimi et al. in [11]. It permits to inject single bit-flips in flip flops and memory units. We apply this technique on the LEON3 processor [10].

The SCFIT platform manages the fault-injection process and the communication between the host computer and the FPGA board. After implementing the target processor on the FPGA board, the host computer sends the program to be executed. A fault is injected in the target processor component during the execution of the program. When the faulty execution completes, snapshots of the RAM are sent back to the host computer. We compare the faulty RAM to the golden RAM in order to classify the fault.

C. Results and Comparison

For the simulations, we set up the following configurations for the LEON3 processor, and we provide them as input to our analysis:

- RAM size: 256 KB
- Data cache size: 4 KB
- Cache policy: write-through for the write hit, no-write allocate for the write miss and LRU for the replacement mechanism
- Register file size: 512 B

1) Simulations on the RAM: First we simulate the effect of faults occurring in the data of the RAM. Fig. 8 presents the masking probabilities of the faults analyzed by the proposed approach compared with those obtained using the FPGA-based fault injector.

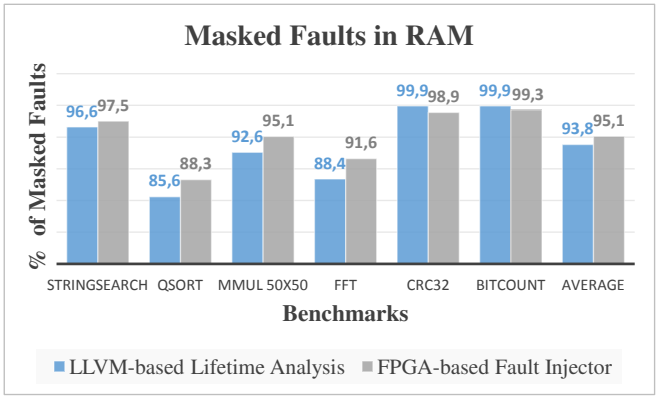

Fig. 8: Results of Fault Classification in the RAM for the LLVM-based Lifetime Analysis and the FPGA-based Fault Injector.

2) Simulations on the Data Cache: We also simulate the effect of faults occurring in the data cache. Fig. 8 presents the masking probabilities of the faults analyzed with the proposed approach compared with those obtained using the FPGA-based fault injector.

Our results are very close to those of the FPGA-based fault injector. On average, the absolute difference is 1.8% for the RAM, and 1.2% for the data cache. This proves that our approach permits to accurately evaluate the effect of faults occurring in different memory components of the system, such as the data cache and the RAM.

To obtain statistically significant results with an error margin of 1% and a confidence level of 95%, 10K fault injections have to be simulated as proposed in [20]. Thus, in term of time and energy consumption, for 10K injections, the program is executed 10K times and the outcomes are analyzed 10K times. However, our approach requires only one program execution and one fault analysis, which greatly saves the simulation time. In all tested cases, our tool concludes the analysis in few seconds.

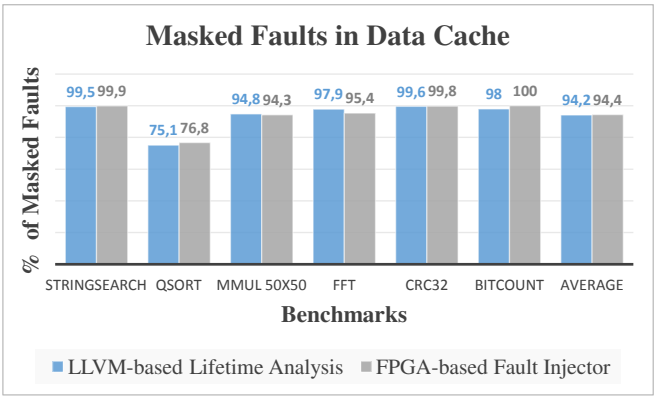

Fig. 9: Results of Fault Classification in the Data Cache for the LLVM-based Lifetime Analysis and the FPGA-based Fault Injector.

D. Discussion

Based on lifetime analysis, our approach permits to evaluate the effect of faults affecting either the data cache or the data of the RAM. As a future work, we propose to target more system components by analyzing both faults occurring in the data and the instructions. We can consider faults in the registers, in the instructions of the RAM and in the instruction cache. Furthermore, in this paper we consider only one level of data cache. Our approach is general enough to be applied on multiple cache levels.

V. Conclusion

In this paper, we presented a new approach to evaluate system reliability without performing a long fault-injection campaign and without requiring a hardware platform. The approach consists in analyzing the variable lifetime and the variable residence in different components of the system, which permits to reduce the reliability evaluation time. In order to be independent from the hardware architecture, we use the LLVM virtual instruction set. Moreover, to achieve a better characterization of the system reliability, we consider the presence of the RAM, the data cache and the register files. For that we build a system emulator that models the behavior of these components.

To validate our approach, we compare the results to an FPGA-based fault injector. The results show that we reach our objectives. Compared to the fault injection technique, our approach permits to save the simulation time without losing accuracy. In addition our approach does not require the presence of a fully defined hardware architecture.

Acknowledgment

We are grateful to Mojtaba Ebrahimi and Prof. Mehdi Tahoori from the Chair of Dependable Nano Computing (CDNC) at KIT in Germany, for their great help to run the FPGA-based fault simulations.

This work has been supported by the joint FP7 Collaboration Project CLERECO (Grant No. 611404).

References

[1] R. Baumann, "Soft errors in advanced computer systems," *IEEE Des. Test*, vol. 22, no. 3, pp. 258–266, May 2005.

[2] S. Borkar, T. Karnik, and V. De, "Design and reliability challenges in nanometer technologies," in *Proceedings of the 41st Annual Design Automation Conference*, ser. DAC '04, 2004, pp. 75–75.

[3] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the 36th Annual International Symposium on Microarchitecture, San Diego, CA, USA, December 3-5, 2003*, ser. MICRO 36, pp. 29–42.

[4] M. Kooli, A. Bosio, P. Benoit, and L. Torres, "Software testing and software fault injection," in *10th International Conference on Design & Technology of Integrated Systems in Nanoscale Era, DTIS 2015, Napoli, Italy, April 21-23, 2015*, 2015, pp. 1–6.

[5] R. Vadlamani, J. Zhao, W. Burleson, and R. Tessier, "Multicore Soft Error Rate Stabilization Using Adaptive Dual Modular Redundancy," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '10, 2010, pp. 27–32.

[6] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. (2013, 02) The Java Virtual Machine Specification. [Online]. Available: http://docs.oracle.com/javase/specs/jvms/se7/html/

[7] Microsoft Corporation, .Net Framework 4. [Online]. Available: http://msdn.microsoft.com/en-us/library/vstudio/w0x726c2%28v=vs.100%29.aspx

[8] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, ser. CGO '04, 2004, pp. 75–86.

[9] M. Kooli and G. Di Natale, "A survey on simulation-based fault injection tools for complex systems," in *Proceedings of the 9th International Conference on Design & Technology of Integrated Systems in Nanoscale Era, DTIS 2014, Santorini, Greece, May 6-8, 2014*, 2014, pp. 1–6.

[10] Leon3. [Online]. Available: www.gaisler.com/index.php/products/processors/leon3

[11] M. Ebrahimi, A. Mohammadi, A. Ejlali, and S. G. Miremadi, "A fast, flexible, and easy-to-develop fpga-based fault injection technique," *Microelectronics Reliability*, vol. 54, no. 5, pp. 1000–1008, 2014.

[12] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Trans. Softw. Eng.*, vol. 24, no. 2, pp. 125–136, Feb. 1998.

[13] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham, "Ferrari: A flexible software-based fault and error injection system," *IEEE Trans. Comput.*, vol. 44, no. 2, pp. 248–260, Feb. 1995.

[14] M. Kooli, P. Benoit, G. Di Natale, L. Torres, and V. Sieh, "Fault injection tools based on virtual machines," in *9th International Symposium on Reconfigurable and Communication-Centric Systems-on-Chip, ReCoSoC 2014, Montpellier, France, May 26-28, 2014*, 2014, pp. 1–6.

[15] A. Thomas and K. Pattabiraman, "LLFI: An intermediate code level fault injector for soft computing applications," 2013.

[16] V. C. Sharma, A. Haran, Z. Rakamaric, and G. Gopalakrishnan, "Towards formal approaches to system resilience," in *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), Vancouver, BC, Canada, December 2-4, 201*, 2013, pp. 41–50.

[17] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, "Computing architectural vulnerability factors for address-based structures," *SIGARCH Comput. Archit. News*, vol. 33, no. 2, pp. 532–543, May 2005.

[18] P. Montesinos, W. Liu, and J. Torrellas, "Using register lifetime predictions to protect register files against soft errors," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 286–296.

[19] Mibench. [Online]. Available: wwweb.eecs.umich.edu/mibench

[20] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09, 2009, pp. 502–506.