System-level Reliability Evaluation through Cache-aware Software-based Fault Injection

Firas Kaddachi[1], Maha Kooli[1], Giorgio Di Natale[1], Alberto Bosio[1], Mojtaba Ebrahimi[2], Mehdi Tahoori[2]
[1]Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier (LIRMM), France
name.surname@lirmm.fr
[2]Karlsruhe Institute of Technology, Karlsruhe, Germany
name.surname@kit.edu

***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 the product time-to-market. In this paper, we propose a novel fault injection technique to evaluate the reliability of a computing system running a software at early design stage where the hardware architecture is not completely defined yet.**

The proposed approach efficiently operates on the original source code of the software in order to inject transient faults in the data or the instructions. To be accurate and to achieve a better characterization of the system, we simulate faults occurring in the system memory units such as the data cache and the RAM by developing a system emulator. To validate our approach, we compare the simulation results to those obtained with an FPGA-based fault injector. The similarity of the results proves the accuracy of our approach to evaluate system reliability with a gain in the execution time and without requiring a fully defined hardware system.

***Index Terms*—Reliability, Soft Errors, Fault Injection, Software, Cache, RAM, Memory**

I. Introduction

Reliability is a major concern for the critical avionic, aerospace, military, and transportation systems. The manufacturing defects, the environmental perturbations and the aging-related phenomena present different fault sources that can lead to failure. Fig. 1 shows how the generated faults can propagate from the hardware components to the software layer. During this propagation, the faults are masked at the technological or at the architectural level [1] [2] , or they reach the software layer of the system and can corrupt the final outputs.

The fault injection is a well-known method, that consists of a deliberate corruption of the system under evaluation to observe its behavior in the presence of faults. While the hardware-based fault-injection techniques directly inject faults in internal processor components, the software-based fault injection techniques model faults at more abstract level. The former techniques perform fault-injection campaigns in more realistic conditions, and hence provide more accurate results. The latter techniques are nevertheless less dependent to the hardware and provide cheaper solutions. However these software-based techniques do not focus on simulating realistic faults occurring in the system memory units such as RAM, data/instruction caches or register files.

This paper proposes a new software-based fault injection technique bridging the gap between software and hardware fault injection: being fast and flexible in early design phase like software-based fault injection, but as accurate as hardware-based fault injection. To reach these objectives, we consider the cache organization which has a significant influence on the results of the fault injection. As we are working on the software level where the concept of RAM and caches is not defined, we build a system memory emulator to model these memory units. In order to be accurate and in the same time as independent as possible from the target hardware architecture, these emulators require minimum information on hardware configurations. The considered fault models are the effect of hardware soft errors (i.e. single event upset) on the software application of the target system. We do not consider errors in code resulting from the implementation or the design of the software.

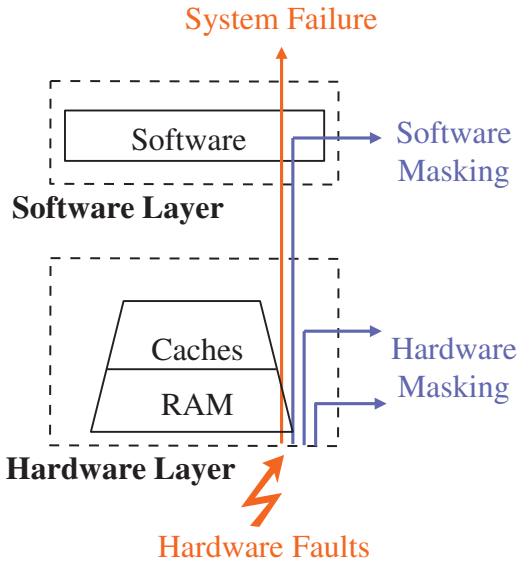

Fig. 1: System Layers and Fault Propagation

The main advantages of this approach are:

- Developing a fast, low-cost and accurate platform to evaluate the overall system reliability in early design stage, when the hardware is not fully designed yet.

- Targeting faults occurring in the system memory units, such as the RAM and the cache hierarchy.
- Offering a better observability to software components and a better controllability of the injection mechanism in terms of the fault location in space and time. This can enhance the development of reliability improvement methods.

The rest of the paper is organized as follows. Section II summarizes related works. Section III introduces the proposed approach. Section IV presents the experimental results, and a comparison with a hardware-dependent fault injector. Finally, section V concludes the paper.

II. Related Works

To evaluate the system reliability, different techniques exist in the literature. The analytical techniques can quickly estimate the failure probability, however, they are highly inaccurate [3] [4]. The fault injection techniques provide different levels of accuracy. In this section, we present the later techniques, and we discuss their shortcomings compared to the proposed approach.

A. Hardware-based Fault-Injection Techniques

The hardware-based fault injection techniques [5] perform fault injections in more realistic conditions. Thus they offer more accurate results. These techniques can use a manufactured processor prototype [6], a simulation of the processor architecture [7] , or an implementation of the processor on an FPGA board [5].

The main difference to the proposed approach is the presence of the hardware under test which makes the reliability evaluation more expensive in terms of hardware cost and execution time. In addition, while the hardware-based techniques are dependent to the target hardware architecture, our approach can be easily applied for different hardware architectures. Finally, our approach offers a better observability of the software components, and thus of the fault location in terms of space (*i.e.* which variable or instruction is affected by the fault). This allows a better understanding of the fault effects at software level and enhances the development of reliability improvement techniques.

B. Software-based Fault-Injection Techniques

The software-based techniques [8] [9] provide cheaper solutions to evaluate the reliability. These techniques model fault injections at different abstract levels: Java-source-code level [10], operating-system level [9] [8], byte-code level [11] or low level virtual machine [12].

In terms of accuracy, the software-based fault-injection techniques are considered in the literature less accurate because they do not perform fault injections in realistic conditions. In addition, they do not target the simulation of faults in the system memory units such as the RAM or the caches. Compared to that, our approach targets an accurate simulation of the faults occurring in the RAM and the data cache. Furthermore, the proposed approach offers a better visibility of the system memory units, which allows a design-space exploration of the target system by easily changing the type and the size of these components to observe the impact on the reliability.

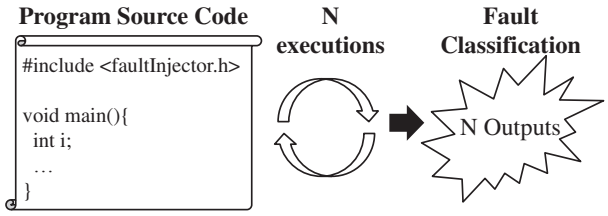

Fig. 2: Overview of the Proposed Approach.

III. Proposed Approach

The proposed fault injection technique allows to inject bit flips in both data and instructions of the target application, in order to simulate realistic faults occurring in the system memory units. It operates on the original source code of the software to simulate single fault injections and does not require a hardware platform. Fig. 2 presents an overview of the approach. The program is executed N times (N is the number of the fault injections). Each program execution simulates a single fault and generates the faulty outputs that are compared with the golden outputs in order to classify the faults. This process can be easily extended to multiple faults.

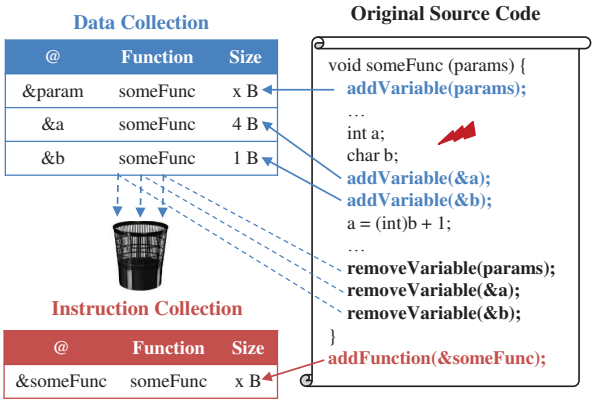

Fig. 3: Data and Instruction Collector.

A. Fault Injection Mechanism

The fault injection mechanism consists of two threads running in parallel.

1) ***Data/Instruction Collector****:* During the program execution, we dynamically collect all the data and the instructions. Fig. 3 shows how we embed additional function calls in the original source code of the program. The proposed tool collects only active software components during program execution. When they are no longer active, they are removed from the dynamic active collection. For fault injection in data, we collect the program variables. These variables represent data

stored in the different system memory units, such as RAM, data cache or register files. For fault injection in instructions, we collect the function codes of the program. The function codes represent instructions that are stored in the RAM or the instruction cache.

*2) **Fault Injector**:* An additional thread is running in parallel with the main thread of the program. It performs a single bit flip per program execution. First, it selects a random time when the fault will be injected. Then it waits until this time is reached to select a random bit to be the target of the fault injection.

B. Fault Classification

The proposed technique presents the advantage of reducing the fault classification to a simple comparison task of the golden and the faulty outputs at software level. Based on this comparison, we consider the following outcomes:

- *Masked:* The faulty program properly terminates its execution and delivers correct outputs. The injected fault does not propagate to the program outputs.
- *Fail Silent Violation (FSV):* The faulty program properly terminates its execution, but provides incorrect outputs.
- *Crash:* The faulty program does not properly terminate. It either suddenly stops working or never stops (e.g. infinite loop).

C. System Memory Emulator

In modern microprocessors, the concept of data cache is introduced to store data for future utilization by the processor. Therefore, the data can occupy different memory units during program execution. The data residence has an influence on the propagation of injected faults to final program outputs. In order to model this aspect, and thus provide more realistic fault injection and more accurate fault classification, we build RAM and data-cache emulators during program execution [13]. The user has to provide some hardware configurations to the proposed tool, in order to model the target processor. Those emulators are easily implemented inside the original source code of the program.

*1) **Data Cache Emulator**:* The data cache emulator collects the most recently used variables during the program execution. It is implemented as a priority queue that follows each read/write operation performed on all variables, in order to model the realistic faults occurring in the data cache. The user has to provide the following data-cache configurations:

- Size: The proposed tool classifies the faults occurring in some unused memory area in the data cache as masked, because they do not have any influence on program execution, which permits to save simulation time.
- Write-Hit Policy: For a write-through data cache, the variable rejoins the head of the most recently used variables. For a write-back data cache, we additionally mark the variable as dirty.
- Write-Miss Policy: For a no-write-allocate data cache, the variable is not added to the data cache. For a write-allocate data cache, the variable joins the head of the most recently used variables.
- Replacement Strategy: It can be Least Recently Used (LRU), Least Recently Replaced (LRR) or random. Depending of the replacement strategy, we remove the variables from the data cache, in order to make more free space.

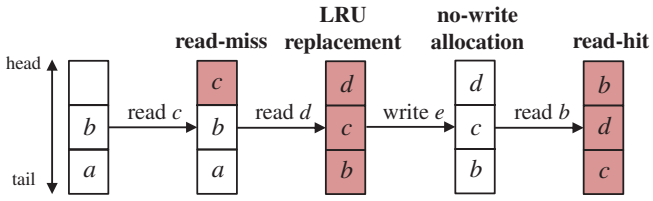

Fig. 4: Example of Data-Cache Emulator

Fig. 4 presents an example of a simplified data-cache emulator with user-specified data-cache configurations. The variable *c* joins the head of the most recently used variables, because it is read and is not present in the data cache. The variable *d* represents a read-miss just like the variable *c*. In addition, the data-cache emulator removes the variable *a* to make more free space, because it is the least recently used variable. The write operation performed on the variable *e* has no influence on the data cache organization, because we have a no-write-allocation policy for write misses. The read operation performed on the variable *b* makes it rejoin the head of the most recently used variables.

*2) **RAM Emulator**:* The RAM emulator is implemented as a collection of all function codes and all active variables during program execution. The proposed tool targets only the active memory area in the RAM, which allows to save simulation time. The user has to provide the total RAM size in order to determine the unused memory area. The faults occurring in this area are considered as masked because they do not have any influence on program execution.

*3) **Fault Injection in Data Cache**:* Fig. 5 illustrates the algorithm we use to simulate faults occurring in the data cache. After selecting a random injection instant and a random byte in the data cache to be the target of fault injection, the algorithm checks if the selected byte resides in some unused memory space in the data cache. In this case, the fault is considered as masked. If the selected byte resides in occupied memory area in the data cache, the algorithm selects a random bit and performs the fault injection. Furthermore, we take into consideration that the variable may leave the data cache and the original variable content is then reloaded from the RAM in later utilization of this variable. The algorithm passes over the fault injection in this variable. In case a write operation is performed on the faulty variable after the injection instant and before leaving the data cache, then the corrupted content of this variable is written back to the RAM, either immediately for a write-through data cache or after leaving the data cache for a write-back data cache. Thus we do not have to undo the fault injection.

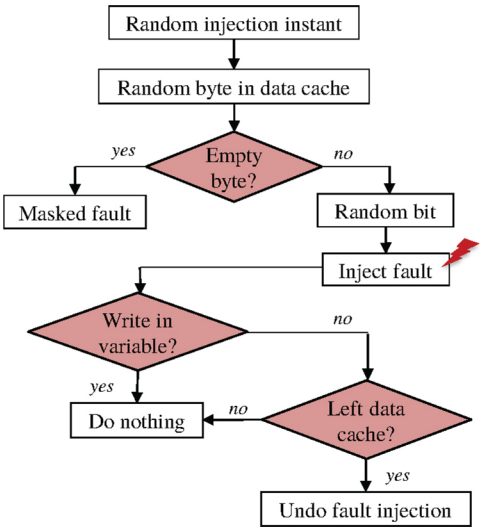

Fig. 5: Algorithm of Fault Injection in Data Cache

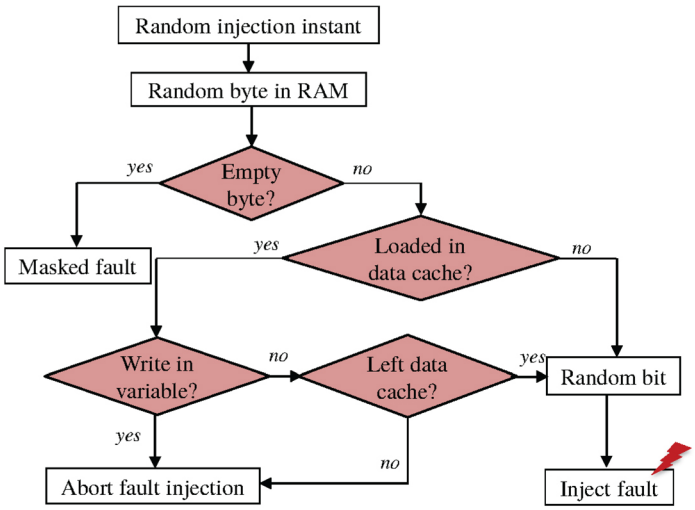

Fig. 6: Algorithm of Fault Injection in RAM

*4) **Fault Injection in RAM**:* Fig. 6 illustrates the flow of simulating the fault injection in the RAM. First, the algorithm checks if the randomly selected byte resides in some unused memory area in the RAM. Then, it classifies the fault as masked. Otherwise, it verifies whether the corresponding variable resides in the data cache at the injection instant, by considering the data cache emulator. The fault injected in a variable residing in the data cache at this instant does not immediately propagate to the final output of the program. Therefore, we do not immediately perform the selected fault injection but we wait until the variable leaves the data cache. In case a write operation is performed on the selected variable before leaving the data cache, the new variable content is written back to the RAM, either immediately for a write-through data cache or after leaving the data cache for a write-back data cache. Then our algorithm aborts the fault injection and considers a fault injected in such a case as masked. We perform fault injection when the variable (i) does not reside in the data cache at the injection instant, or (ii) resides in the data cache at the injection instant, but leaves the data cache, without being affected by a write operation after the selected injection instant and before leaving the data cache.

IV. Experiments and Results

The proposed approach is used to conduct experiments on different benchmarks. The results are compared with an accurate FPGA-based fault injection technique applied on the LEON3 processor [14].

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 MiBench benchmark suite [15] (bit count, quick sort, string search, fft, crc 32). All the test-benches are written in C-programming language.

B. FPGA-based Fault Injector

To validate our approach, we run the same benchmarks using a hardware-based fault injector. Such tools are considered in the literature as accurate techniques to evaluate system reliability. We use SCFIT, which is an FPGA-based fault injector proposed in [5]. It allows to inject faults in flip flops and memory units. We apply this technique on the LEON3 processor [14].

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 inputs to our tool:

- RAM size: 256KB
- Data cache size: 4KB / 8KB
- Data cache policy: write-through for the write miss, no-write allocate for the write hit and LRU for the replacement strategy

In order to obtain statistically significant results with an error margin of 1% and a confidence level of 95%, we simulate 10K fault injections for each program as proposed in [16].

*1) **Simulations on the Data Cache:*** We simulate the effect of faults occurring in the data cache. Fig. 7 presents the masking probabilities of the faults injected with the proposed approach compared with those obtained using the FPGA-based fault injector. Fig. 8 provides a comparison of the execution time between the experiments run with the proposed approach and the FPGA-based fault injector.

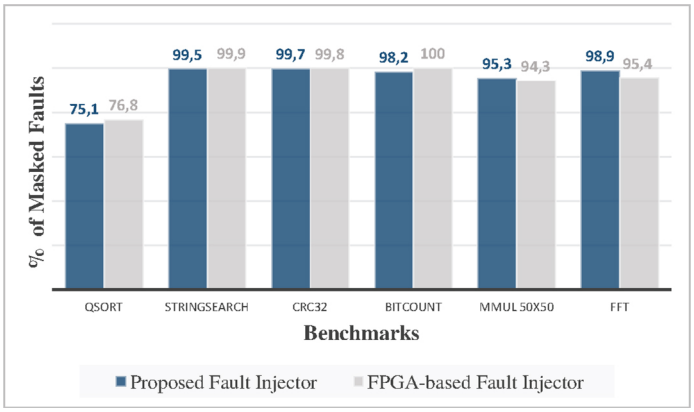

Fig. 7: Masking probabilities of Proposed Tool and FPGA-based Fault Injector for Faults Occurring in Data Cache.

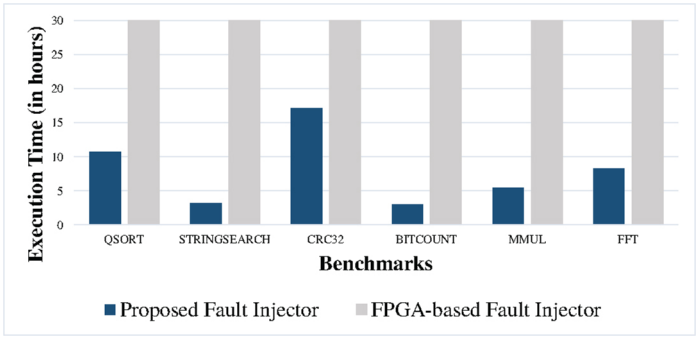

Fig. 8: Execution Time (in hours) of Proposed Tool and FPGA-based Fault Injector for Faults occurring in Data Cache

*2) **Simulations on the RAM:*** We also simulate the effect of faults occurring in the RAM. For these experiments, the fault injections in the instructions of the RAM are not considered. For the employed benchmarks, the size of the instructions is too small compared to the size of the data. Fig. 9 presents the masking probabilities of the faults injected with the proposed approach compared with those obtained using the FPGA-based fault injector. Fig. 10 provides a comparison of the execution time between the experiments run with the proposed approach and the FPGA-based fault injector.

The results of the proposed approach are very close to those of the FPGA-based fault injector. They show a significant accuracy gain by modeling the system memory units. On average, the integration of the system memory emulators reduces the absolute difference from more than 10% to 2.3%

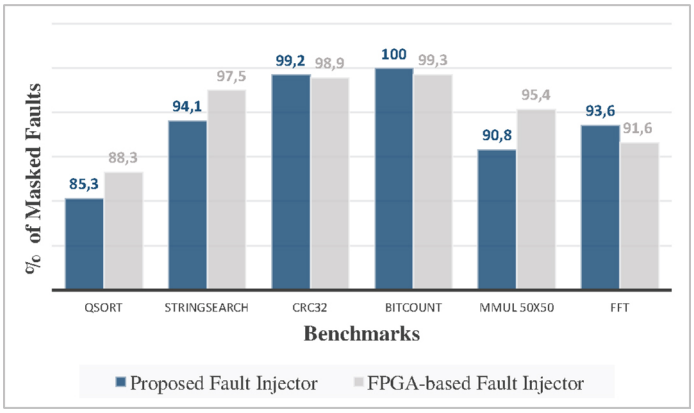

Fig. 9: Masking probabilities of Proposed Tool and FPGA-based Fault Injector for Faults Occurring in RAM.

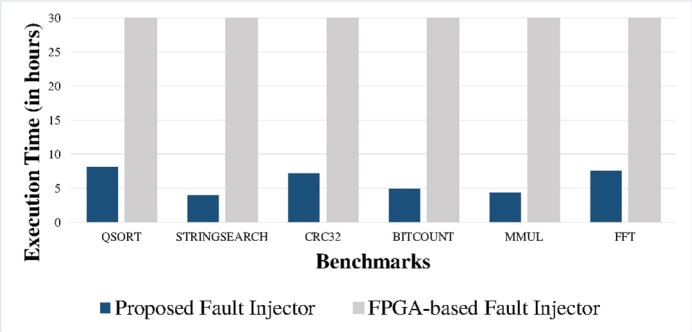

Fig. 10: Execution Time (in hours) of Proposed Tool and FPGA-based Fault Injector for Faults occurring in RAM.

for the RAM, and 1.4% for the data cache. This proves that our approach allows to accurately evaluate the effect of faults occurring in different memory components of the system, such as the data cache and the RAM.

Furthermore, Fig. 8 and Fig. 10 show that the proposed approach offers a significant gain in the execution time. On average, the speed-up is respectively 6x and 5x for the faults occurring in data cache and RAM, compared to the FPGA-based fault injection technique. It is important to mention that the used FPGA-based fault injection technique is 3 to 4 orders of magnitude faster than a simulation-based fault injection at hardware.

*3) **Design Space Exploration:*** In order to show that the proposed approach can be easily applied on different hardware architecture, we change, in an additional set of experiments, the size of the data cache. Fig. 11 presents the results of the fault injection using the proposed tool for both 4KB and 8KB data cache.

The proposed tool allows to easily change the hardware configurations in order to observe their impact on the overall system reliability. The making probabilities for 8KB data cache increase on average compared to 4KB data cache. In fact, when the data cache size increases, there is more

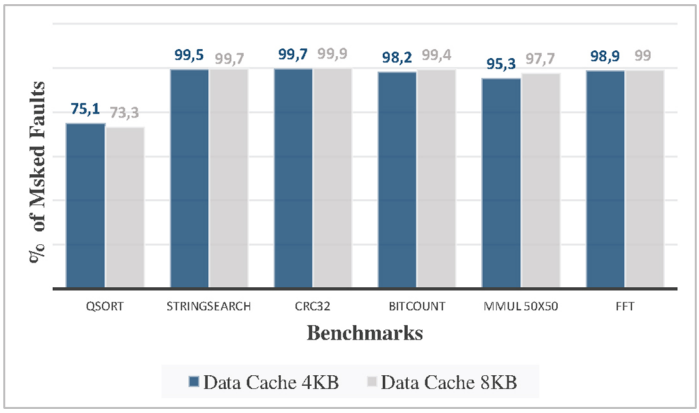

Fig. 11: Masking probabilities of Proposed Tool for 4KB and 8KB Data Cache.

probability to inject faults in variables that do not have any influence on further program execution. However this also depends on the workload, which is the case for the Qsort program. This workload sorts the elements of a given array and re-reads these elements in each iteration. Thus all variables residing in the data cache have influence on final program outputs.

D. Discussion

The proposed fault injection technique allows to evaluate the effect of transient faults affecting the data cache and the RAM. It can also target the permanent faults. As a future work, we propose to target more system components by simulating faults occurring in the registers, and the instruction cache. Furthermore, in this paper we only consider one-level data cache. Our approach is generic enough to be applied on multiple-level data cache.

To enhance the performance of the tool, we propose to optimize the fault injection process. We can automate the collection of the program data and instructions. Furthermore, we can improve the implementation of the integrated functions in order to reduce the computational overhead.

V. Conclusion

In this paper, we propose a fast, low-cost and accurate software-based approach to evaluate the reliability of critical systems in early design stage. The proposed tool efficiently operates on the source code of the workloads running on the target system. It allows to inject faults in both program variables and function codes that respectively represent data and instructions stored in the system memory units. To be accurate, we build a RAM emulator and a data-cache emulator, which require minimum information on hardware configurations.

To validate our approach, we compare our results to an FPGA-based fault injector. The results show that we reach our objectives. In terms of execution time, our approach offers a significant gain compared to the existing fault-injection techniques. In addition our approach does not require the presence of a fully defined hardware. The system emulators allow an efficient design-space exploration of the target system by changing the type and the size of the system memory units to observe the impact on the reliability. Finally, in terms of accuracy, the proposed tool provides an accurate evaluation of the system reliability that is very close to a hardware-based evaluation.

Acknowledgment

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

References

[1] 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.

[2] 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.

[3] N. J. George, C. R. Elks, B. W. Johnson, and J. Lach, "Transient fault models and AVF estimation revisited," in *Proceedings of the 2010 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2010, Chicago, IL, USA, June 28 - July 1 2010*, 2010, pp. 477–486.

[4] M. Ebrahimi, L. Chen, H. Asadi, and M. B. Tahoori, "CLASS: combined logic and architectural soft error sensitivity analysis," in *18th Asia and South Pacific Design Automation Conference, ASP-DAC 2013, Yokohama, Japan, January 22-25, 2013*, 2013, pp. 601–607.

[5] 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.

[6] M. Nicolaidis, *Soft errors in modern electronic systems*. Springer Science & Business Media, 2010, vol. 41.

[7] J. J. H. Pontes, N. Calazans, and P. Vivet, "An accurate single event effect digital design flow for reliable system level design." in *DATE*, W. Rosenstiel and L. Thiele, Eds. IEEE, 2012, pp. 224–229.

[8] 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.

[9] 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.

[10] R. de Oliveira Moraes and E. Martins, "Jaca - a software fault injection tool," *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, p. 667, June 2003.

[11] B. P. Sanches, T. Basso, and R. Moraes, "J-swfit: A java software fault injection tool," in *Dependable Computing (LADC), 2011 5th Latin-American Symposium on*. IEEE, 2011, pp. 106–115.

[12] 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*
2014, Montpellier, France, May 26-28, 2014, 2014, pp. 1–6.

[13] M. Kooli, F. Kaddachi, G. D. Natale, and A. Bosio, "Cache- and register-aware system reliability evaluation based on data lifetime analysis," in *Proceedings of the 34th IEEE VLSI Test Symposium, VTS 2016, Las Vegas, USA, April 25-27, 2016*.

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

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

[16] 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.