HAL
open science

Cache-aware reliability evaluation through LLVM-based analysis and fault injection

Maha Kooli, Giorgio Di Natale, Alberto Bosio

To cite this version:

Maha Kooli, Giorgio Di Natale, Alberto Bosio. Cache-aware reliability evaluation through LLVM-based analysis and fault injection. IOLTS: International On-Line Testing Symposium, Jul 2016, Sant Feliu de Guixols, Spain. pp.19-22, 10.1109/IOLTS.2016.7604663 . lirmm-01444619

HAL Id: lirmm-01444619

https://hal-lirmm.ccsd.cnrs.fr/lirmm-01444619v1

Submitted on 25 Jul 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Cache-aware Reliability Evaluation through LLVM-based Analysis and Fault Injection

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

***Abstract*[1]—Reliability evaluation is a high costly process that is mainly carried out through fault injection or by means of analytical techniques. While the analytical techniques are fast but inaccurate, the fault injection is more accurate but extremely time consuming. This paper presents an hybrid approach combining analytical and fault injection techniques in order to evaluate the reliability of a computing system, by considering errors that affect both the data and the instruction cache. Compared to existing techniques, instead of targeting the hardware model of the cache (*e.g.*, VHDL description), we only consider the running application (*i.e.*, the software layer). The proposed approach is based on the Low-Level Virtual Machine (LLVM) framework coupled with a cache emulator. As input, the tool requires the application source code, the cache size and policy, and the target microprocessor instruction set. The main advantage of the proposed approach is the achieved speed up quantified in magnitude orders compared to existing fault injection techniques. For the validation, we compare the simulation results to those obtained with an FPGA-based fault injector. The similarity of the results proves the accuracy of the approach.**

***Index Terms*—Reliability, Soft Errors, Fault Injection, Analysis, LLVM, Cache, Data, Instruction**

I. Introduction

A constant trend in the computer architecture is the continuous reduction of the distance between the microprocessor and the memory. From a single external RAM, we use now up to four cache levels in order to reduce the data access time. The main consequence is that up to 60% of the microprocessors die area is nowadays devoted to cache memory blocks [1]. The large portion of occupied die area make the caches to be critical components for the whole microprocessor reliability especially for safety critical fields such as avionic, aerospace, and transportation. Thus, any fault in the cache can cause catastrophic failure of the system. To avoid such events, designers investigate methods to evaluate the reliability. The most used are fault injection and analytical techniques. While the former are accurate but very time consuming since they require performing a big number of tests, the latter are fast but inaccurate.

This paper proposes a new approach to evaluate the reliability of a computing system based on the analytical technique and the fault injection. One goal of this work is to study the outcomes of a computing system running a software when affected by faults. The system outcomes are significantly influenced by faults occurring in the cache, since they are very close to the main processing unit and can easily propagate to other components through the CPU and register files. Thus, we target in this paper faults affecting the data and the instruction cache.

The proposed approach is based on LLVM [2], a compiler framework that uses virtualization with Virtual ISA (VISA) to perform complex analysis of software applications on different architectures. The use of the Intermediate Representation (IR) as a form to represent code in the compiler, makes the framework independent from the source language and the target machine. Using LLVM, we aim to have an abstraction, at software level, of the caches and the hardware faults. We implement a cache emulator that models the cache behavior, and we define fault models that represent the effect of soft errors at the software layer. This abstraction permits to save the time and the cost of using a fully designed hardware system.

The advantages of our approach are: (i) achieving speed up of the simulation time compared to existing fault injection techniques, (ii) targeting faults in the data and instruction cache without requiring a fully designed hardware architecture, (iii) having accurate results and good observability of the fault location in space and time, which could help in the design 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 Section V concludes the paper.

II. Related Works

A. Fault-Injection Techniques

LLFI [3] and KULFI [4] are LLVM-based fault injectors permitting to inject fault into the IR code. As fault model, LLFI considers transient faults into the processor's computation units, and KULFI injects single bit flips into the instructions as well as in the data/address registers. The major difference compared to the proposed approach is the considered fault location. Thanks to the integration of the cache emulator, we target faults occurring in the data and the instruction cache, which is not considered by these approaches. In addition, we inject the fault directly in the readable IR (not the binary), which offers more observability of the fault location in space and time.

More generally, several fault injection techniques exist in the literature to evaluate the soft errors [5]. For software-based techniques, the fault injection mechanism is modeled

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

at different abstract levels such as operating-system level [6] [7] or source-code level [8]. These techniques are cheap but not enough accurate. The hardware-based techniques can use a manufactured processor prototype [9], a simulation of the processor architecture [10], or an implementation of the processor on an FPGA board [11]. These techniques are accurate, but more expensive in terms of hardware cost and evaluation time. The proposed approach accurately evaluates the effect of soft errors without requiring a fully definition of the hardware architecture, and thus significantly reduces the evaluation time and cost.

B. Analytical Reliability Evaluation Techniques

Analytical techniques aim to develop a low cost reliability evaluation tool in term of execution time and hardware dependency. These techniques permit to quickly estimate the failure probability but they are highly inaccurate [12].

Mukherjee et al. [13] use lifetime analysis to compute the Architectural Vulnerability Factor (AVF) of the instruction queue and execution units. Biswas et al. [14] apply the concept of lifetime to compute the AVF of the data cache, the data translation buffer, and the store buffer. While the previous techniques perform the analysis on a real processor, our approach uses the VISA of LLVM, which makes the evaluation less dependent from the hardware architecture. Using the same VISA, Kooli et al. [15] use the lifetime analysis to compute the minimum percentage of failure probability of faults occurring in the data (RAM, data cache). Compared to this technique, the proposed approach computes the exact failure probability, and is able to target faults occurring in both data and instructions.

III. Proposed Approach

A. Software Fault Model

In this work, we aim at detaching the analysis of the software level from the hardware level. We therefore need to model how hardware faults manifest at the software level. When a fault reaches the software level, it can corrupt data, instructions or control flows. To represent these behaviors at software level, we define fault models as mutations [16] in the LLVM source code. Each mutation represents the effect of a real fault occurring in the hardware.

- *Wrong Data (WDat):* An operand of the VISA changes its value due to a bitflip, or is used in place of another (*e.g.*, a pointer refers to a wrong address). This corresponds to the result of a Single Event Upset (SEU) affecting either the instruction operand or the memory segment storing the data of the program. This also models the control flow error when the operand corresponds to a branch condition.
- *Instruction Replacement (InsR):* An opcode in the VISA is misused (*e.g.*, the opcode 'add' is interpreted as 'sub' or as invalid operand). This corresponds to the result of a SEU affecting the opcode of the instruction. This also models the control flow error when the target opcode is jump or branch.

B. Fault Analysis Process

The fault occurs randomly at any memory location and instant. During its execution, the program can use the whole available memory or a part of it. It is clear that the faults occurring in the memory not used by the program do not have any impact on the program execution. Therefore, to save simulation time, we consider these faults as masked, without running fault injection simulation. As presented in Fig. 1, the tool starts by a first analysis of the program. It randomly selects an instant and a memory location. If the location corresponds to non used memory, the fault is classified as masked, otherwise the tool proceeds by the fault injection as it will be explained in the next section.

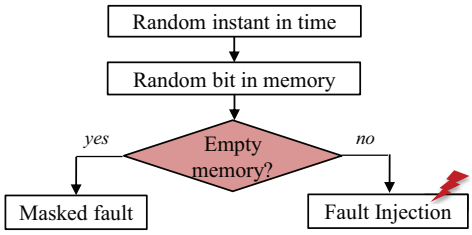

Fig. 1: Fault Analysis Process.

C. Fault Injection Process

The fault injection mechanism, presented in Fig. 2, permits to inject faults into the readable LLVM code of the program. For each analyzed fault (*i.e.*, non masked in the analysis process), the tool generates a faulty program with mutation [16] in the original program, where the mutant corresponds either to WDat or InsR.

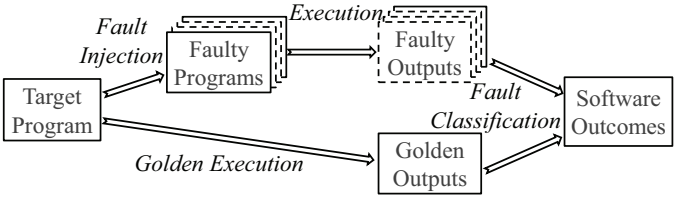

Fig. 2: Fault Injection Process.

1) Fault Injection in Data: In LLVM, the data is represented by the program variables (operands). The fault injection in the data corresponds to the WDat fault model. We model the bit flip by a bitwise xor between the correct value or address and a random bit. Since the fault injection is embedded into the source code, we have to dynamically control the fault instant. Thus to make the fault transient, we add an index that permits to inject the fault only in the selected instant. In Fig. 3, we provide an example of permanent and transient fault injected respectively in the address and the value of a variable.

2) Fault Injection in Instruction: Faults in the instruction can affect either the opcode or the operands. If the selected bit corresponds to one of the operands, the fault becomes a WDat in that operand, and we follow the process explained in the previous sub-subsection. However, if the selected bit corresponds to an opcode, the fault is modeled as an InsR.

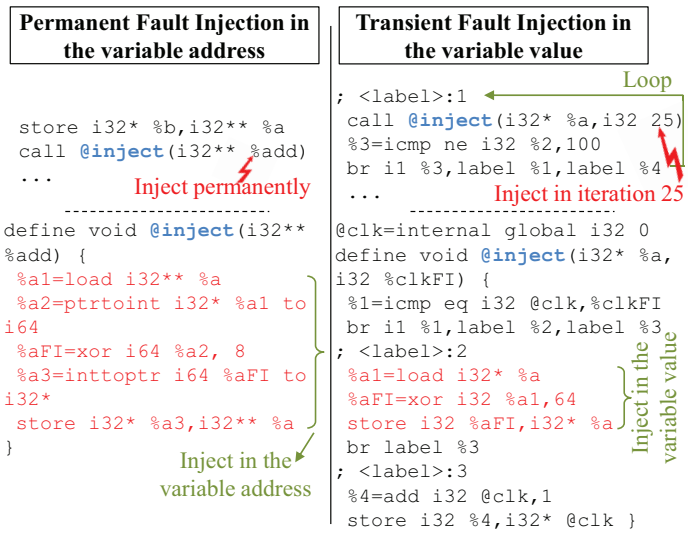

Fig. 3: Example of Fault Injection in the Data.

The InsR fault corresponds to the transformation of an opcode into a different one. This transformation is microprocessor dependent. For a given ISA, we developed a tool that analyzes, for each instruction, the probability that an opcode i becomes the opcode j under a bit flip. For this work, we consider a single fault affecting the SPARC ISA (this ISA is used for the approach validation in section IV). The tool considers the binary representation of each opcode, injects a bit flip and checks if the transformed code corresponds to a valid or invalid opcode. Having the statistics for the ISA, we map them to the LLVM VISA to obtain the statistics on the VISA. These statistics provide, for each opcode, the probability to switch to a valid or invalid opcode.

D. Cache Emulator

Modern microprocessor-based systems extensively use caches to reduce the access time to data and instructions. At software level, the concept of caches is not defined. In order to be able to target faults in caches, we implement a cache emulator as a queue that collects the used variables or instructions during the program execution. To build the cache, some hardware configurations are required such as the cache size, the write hit and miss policy and the replacement strategy.

The data cache emulator accumulates the used variables during the program execution. It follows each read and write operations. A first execution of the program permits to generate a trace of its variables containing information about each operation, the corresponding clock, and the variable size and physical address. Using this trace together with the cache configurations, we build the state of the cache for each clock cycle of the program execution.

Fig. 4 shows an example of a small data cache emulator, with a size of 96 bits, a write through and no-write allocation policy and a Least Recently Used (LRU) replacement strategy. For no-write allocation, the variable enters or updates its position to the head, when there is a read operation (both read and write operation in case of write allocation). If the cache is full at the moment of adding a new variable, the algorithm keeps removing existing variables following the replacement strategy, till it finds enough space for the new variable.

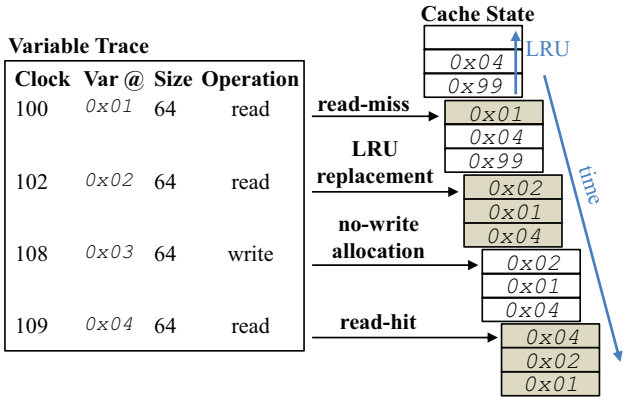

Fig. 4: Data Cache State for each Clock Cycle.

The instruction cache emulator collects the instructions executed by the program. It is easier to implement compared to the data cache emulator since we do not follow the read and write operations. We simply provide a unique ID to each instruction in the code and consider its size. Then, for each clock cycle, the instruction either enters or updates its position to the head of the cache. If the cache is full at the moment of adding new instruction, the algorithm keeps removing existing instructions following the replacement strategy, till we find enough space for the new instruction.

E. Simulating Faults occurring in Caches

Once the cache emulator is built, we are able to simulate faults occurring in the data and the instruction cache. In this paper, we consider only transient faults. We apply the algorithm presented in Fig. 1. The faulty instant and bit are randomly selected from respectively the total clock cycles and the cache size. For the analysis process, we consider the active memory as the size of the variables/instructions residing in cache at the injection instant. For the fault injection process, we identify the faulty variable or instruction corresponding to the selected bit and we create the mutant. After the activation of the mutant, we follow the behavior of the faulty variable. When it leaves the cache without being written, and in case of write through policy, the variable will recuperate the correct value from the RAM (or from the higher-level cache). Thus we undo the fault injection at that instant.

IV. Experiments and Results

To evaluate our approach, we select a set of workloads from the open-source MiBench benchmark suite [17] (bit count, quick sort, string search, fft, crc 32). Then for validation, we compare the results to those provided by a hardware-based fault injector, which is considered in the literature as accurate techniques to evaluate system reliability. The used tool is SCFIT, an FPGA-based fault injector applied on the LEON3 processor, proposed in [11].

For the simulations, we set up the following hardware configurations for both the FPGA-based and the proposed method: 256KB as RAM size, 4KB as cache size, 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 faults for each program as proposed in [18].

1) Accuracy Evaluation: Fig. 5 and Fig. 6 presents the masking probabilities of the faults respectively injected in the data cache and the instruction cache, with the proposed approach compared with those obtained using the FPGA-based fault injector. The results of the proposed approach are very close to those of the FPGA-based fault injector. This proves that our approach allows to accurately evaluate the effect of faults occurring in the cache.

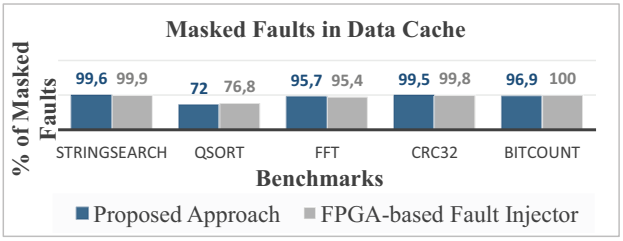

Fig. 5: Evaluation of Faults Occurring in Data Cache.

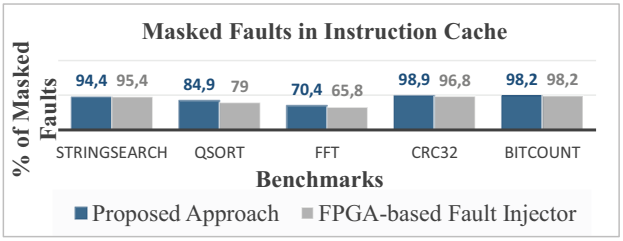

Fig. 6: Evaluation of Faults Occurring in Instruction Cache.

2) Simulation Time: Table I presents the percentage of faults evaluated only with the analytical process and without going through the fault injection process. For some workloads, this percentage is high (>80%) depending on the size of occupied space in the cache by the data/instructions. Thanks to the analytical process, our approach reduces the number of performed fault injections. It permits to minimize the cost of executing the same workload several time, which represents the main disadvantage of a full fault injection method.

TABLE I: Faults Evaluated with the Analytical Process.

Workload	StrSearch	Qsort	FFT	CRC32	BitCount
Faults in Data	9.5%	1.1%	1.6%	2.9%	95.9%
Faults in Instruction	83.2%	32.9%	41.1%	97.5%	96.5

In addition to the time saved from the analytical process, the fault injection process offers as well a significant gain. In fact, the faults are injected inside the source code of the program. Thus the simulation time of the faulty program is the simulation time of the golden program because the embedded code to simulate the fault does not have influence on the program execution. For all the used benchmarks, the time of the golden execution is about 1s. Using the FPGA-based fault injector, it requires 10.8s to inject a fault. However, using our method, it needs the same time of the golden execution. This means that the speed-up is about 10x for one simulation. 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.

V. Conclusion

This paper proposes a reliability evaluation approach that uses the analytical technique and the fault injection. It is based on the LLVM framework coupled with a cache emulator to consider faults in the data and the instruction cache. To proves the accuracy of the evaluation, we compare our approach with an FPGA-based fault injector. In term of execution time, the approach is very fast thanks to the integration of the analytical process, and the software abstraction of the hardware system.

References

[1] S. D. Carlo *et al.*, "Software-based self-test of set-associative cache memories," *IEEE Transactions on Computers*, 2011.

[2] 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*, 2004, pp. 75–86.

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

[4] V. C. Sharma *et al.*, "Towards formal approaches to system resilience," in *Proceedings of the 19th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC*, 2013, pp. 41–50.

[5] M. Kooli *et al.*, "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*, 2014.

[6] G. A. Kanawati *et al.*, "FERRARI: A flexible software-based fault and error injection system," *IEEE Transactions on Computers*, 1995.

[7] J. Carreira *et al.*, "Xception: A technique for the experimental evaluation of dependability in modern computers," *IEEE Trans. Softw. Eng.*, 1998.

[8] F. Kaddachi *et al.*, "System-level reliability evaluation through cache-aware software-based fault injection," in *Proceedings of the 19th IEEE International Symposium on Design and Diagnostics of Electronic Circuits and Systems, DDECS*, 2016.

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

[10] J. J. H. Pontes and otherl, "An accurate single event effect digital design flow for reliable system level design." in *Design, Automation and Test in Europe, DATE*, 2012.

[11] M. Ebrahimi *et al.*, "A fast, flexible, and easy-to-develop FPGA-based fault injection technique," *Microelectronics Reliability*, 2014.

[12] M. Ebrahimi *et al.*, "CLASS: combined logic and architectural soft error sensitivity analysis," in *Proceedings of 18th Asia and South Pacific Design Automation Conference, ASP-DAC*, 2013.

[13] S. S. Mukherjee *et al.*, "A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor," in *Proceedings of the 36th Annual International Symposium on Microarchitecture, MICRO*, 2003.

[14] A. Biswas *et al.*, "Computing architectural vulnerability factors for address-based structures," *SIGARCH Comput. Archit. News*, 2005.

[15] M. Kooli *et al.*, "Cache- and register-aware system reliability evaluation based on data lifetime analysis," in *Proceedings of the 34th IEEE VLSI Test Symposium, VTS 2016*.

[16] M. Kooli *et al.*, "Software testing and software fault injection," in *Proceedings of 10th International Conference on Design & Technology of Integrated Systems in Nanoscale Era, DTIS*, 2015.

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

[18] R. Leveugle *et al.*, "Statistical fault injection: Quantified error and confidence," in *Design, Automation and Test in Europe, DATE*, 2009.