Bayesian Network Early Reliability Evaluation Analysis for both permanent and transient faults

A. Vallero*, A. Savino*, S. Tselonis†, N. Foutris†, M. Kaliorakis†, G. Politano*, D. Gizopoulos†, S. Di Carlo*
*Control and Computer Engineering Department, Politecnico di Torino, 10129 Torino, Italy
†Department of Informatics & Telecommunications, University of Athens, 5784, Athens Greece

***Abstract*—Analyzing the impact of software execution on the reliability of a complex digital system is an increasing challenging task. Current approaches mainly rely on time consuming fault injections experiments that prevent their usage in the early stage of the design process, when fast estimations are required in order to take design decisions. To cope with these limitations, this paper proposes a statistical reliability analysis model based on Bayesian Networks. The proposed approach is able to estimate system reliability considering both the hardware and the software layer of a system, in presence of hardware transient and permanent faults. In fact, when digital system reliability is under analysis, hardware resources of the processor and instructions of program traces are employed to build a Bayesian Network. Finally, the probability of input errors to alter both the correct behavior of the system and the output of the program is computed. According to experimental results presented in this paper, it can be stated that Bayesian Network model is able to provide accurate reliability estimations in a very short period of time. As a consequence it can be a valid alternative to fault injection, especially in the early stage of the design.**

I. Introduction

Reliability is an important design aspect for computer systems due to the aggressive technology miniaturization [1], [2], [3]. Unreliable hardware components affect computing systems at several levels. Raw errors can manifest due to several causes such as physical fabrication defects, aging or degradation (e.g., NBTI), process variations, environmental stress (e.g., radiations). Raw hardware errors can propagate through other layers of the system (e.g., architecture, software) up to the output. During the propagation process, raw errors can be masked by each of the affected layers.

A significant effort in the research community has been spent to analyze masking properties at technological and architectural level [4], [5]. Moreover, understanding the effect of software on the reliability of a complex system in which unreliable hardware is present is also gaining increasing importance. The software has intrinsic masking capabilities that can be enhanced by the implementation of software level fault tolerance mechanisms [6], [7] and [8]. However, these mechanisms often incur in a significant performance overhead. Therefore, the role of the software stack coupled with the target hardware architecture must be carefully considered when system reliability is analyzed.

This paper has been fully supported by the 7th Framework Program of the European Union through the CLERECO Project, under Grant Agreement 611404. Direct questions and comments to A. Vallero (alessandro.vallero@polito.it), A. Savino (alessandro.savino@polito.it) and S. Di Carlo (stefano.dicarlo@polito.it).

Several studies focus on understanding and modeling how hardware faults can propagate and manifest through a software application, without considering how these faults can actually propagate and be masked within the software [9], [10], [11], [12]. A very important contribution that examines the impact of software on the architectural vulnerability factor (AVF) of a system is provided in [13]. The paper defines a so called Program Vulnerability Factor (PVF), isolating the software-dependent (architecture-level masking) portion of AVF from the hardware-dependent (microarchitecture-level masking) portion. This metric captures the architecture-level fault masking inherent in a program, allowing software designers to make quantitative statements about programs resilience to soft-errors. PVF can be measured using an architectural simulator, a dynamic binary translator such as Pin [14] or resorting to ACE-like analysis [15]. Moreover, a comprehensive PVF calculation is affected by the software workload that may increase the simulation effort. Another interesting contribution has been proposed in [16] where a statistical model is proposed to estimate the capability of a software application to mask hardware errors. The main contribution of this paper is to introduce a statistical model that simply requires a preliminary characterization of the hardware masking probability and then it is able to analyze software applications executed on this hardware. However this work does not take into account execution time so that it may lead to inaccurate estimations especially for big programs. Overall, one of the main limitations of the publications presented so far is that they are limited to the analysis of the effect of soft-errors: permanent and intermittent faults are not considered at all.

This paper introduces a new statistical approach for the estimation of the reliability of a microprocessor-based system considering both the hardware and the software layer. The software execution on the microprocessor is modeled in the form of a Bayesian Network that describes relations among resources (e.g., registers, memory elements, functional units) involved in the execution of the instructions composing a program. The Bayesian model is then exploited to compute a probability of correct (error-free) execution of the software in the presence of a hardware fault, used to estimate the overall reliability of the system. To construct the model, a preliminary characterization of the Instruction Set Architecture (ISA) of the microprocessor is required. This characterization aims at evaluating the probability of successful execution of each instruction of the target ISA in presence of faults in the microprocessor hardware blocks. Transient, intermittent and permanent faults can be considered in this phase without affecting the way the high-level model is constructed.

978-1-4673-7905-2/15/$31.00

Bayesian Networks represent a model successful employed for reliability estimates in different fields [17]. In the software engineering domain, they have been successfully employed to model software reliability in the distributed domain (Kishore at al. [18], etc.). A few publications consider the application of Bayesian Networks to model system reliability in hardware devices as well [19], [20]. However, they do not consider the interaction of hardware and software in an instruction-based environment.

To validate the proposed statistical model we analyzed 4 different MiBench benchmarks [21] executed targeting permanent and transient faults on top of a x86-64 microprocessor. Experimental results highlight that reliability estimations are accurate when compared to those obtained by time-consuming fault-injection experiments performed using a micro-architectural fault injector [22]. At the same time, the proposed approach enables a significant reduction of the time required to perform the reliability analysis, enabling fast and accurate reliability evaluations.

This paper is organized as follows: Section II illustrates the proposed reliability estimation model. It focuses on the creation and the evaluation of the Bayesian Network. Section III introduces the experimental setup and shows estimation results compared to the fault injection campaigns. Eventually, Section IV summarizes the main contributions of the paper and set future perspectives on statistical estimation.

II. Reliability analysis

Bayesian Networks (BNs) are an efficient statistical model to represent multivariate statistical distribution functions. They can model relationships among random variables and their respective probability density functions by means of conditional probability functions. In particular, conditional dependencies are expressed by a *Direct Acyclic Graph* (DAG). Nodes of the DAG represent conditional probability functions of the random variables, while edges represent conditional dependencies among random variables. If two nodes are connected, it means that the random variables they represent are conditionally dependent.

The proposed statistical reliability analysis methodology focuses on the estimation of the probability of failure of a program running on a specific microprocessor. Bayesian networks are employed to model program traces, i.e., sequences of instructions issued by the microprocessor when the program is executed with a specific workload. In particular, the information required to build the BN is the execution time and the involved hardware resources of instructions. Several traces can be obtained from a single program by profiling its execution with different workloads using a dedicated profiler. We employed [16], but other profilers can be used as well without affecting the estimation model. The collection of analyzed traces represents different program behaviors that may generate different error masking effects during the program execution. Program traces are sequential lists of instructions that can therefore be efficiently modeled in the form of a Bayesian network whose main constraint is the absence of loops in the network. As traces are evaluated, the system failure probability can be estimated by averaging the trace analysis results.

To assess the system reliability, both the hardware and the software layer must be taken into account. In the proposed model the microprocessor ISA represents the direct link between the two layers. In particular, each instruction is considered with regard to the hardware resources required for its execution. To build a BN model of a program trace running in a system, it is essential to take into account the hardware resources, error sources and the instructions of the analyzed trace.

Hardware resources, Res, are divided into two subsets: the storage resources subset (S_{res}), which includes registers and memories, and the computational resources subset (C_{res}), which includes functional units required for computation.

Resources can be affected by errors during program execution. Consequently, when dealing with reliability analysis, it is fundamental to define an error model. For soft-errors the exponential distribution is assumed. The reliability function of a resource ($R_{res}(t)$), i.e., the probability of an error-free resource in a period of time t is given by the following equations:

$$\lambda_{res} = \lambda_{comp} \frac{A_{res}}{A_{comp}} \tag{1}$$

$$R_{res}(t) = e^{-\lambda_{res} t} \tag{2}$$

where λ_{comp} is the raw SER of the electronic component dependent on the target technology, while λ_{res} is the resource SER assuming equal spatial distribution of error occurrence in the component. Since a single resource is part of bigger electronic component, it can be assumed that its reliability is related to the one already defined for the component. More specifically, λ_{res} is a portion of λ_{comp} and we assume it is proportional to the fraction of its silicon area A_{res} over the total area of the component A_{comp} (1). For permanent errors we assume stuck-at-fault model. If a permanent fault occurs in a resource, it means that an internal signal or the output signal of the affected resource has a fixed value.

The occurrence of both transient and permanent errors in hardware resources can be masked at the hardware level due to several masking effects. We therefore consider a masking probability for each instruction of the ISA, representing the probability that an instruction prevents an error occurrence to propagate. In this work, each instruction I is characterized by a set of input storage and computational resources, $R_{I_{in}} \in S_{res} \cup C_{res}$, required for the computation, and a set of output storage resources, $R_{I_{out}} \in S_{res}$, that are updated by I. All resources in $R_{I_{in}}$ can mask errors during the execution of an instruction and are therefore characterized by a given masking probability. Masking of hardware resources can be obtained by error protection mechanism implemented in the design (e.g., ECC for memories, Triple Module Redundancy for registers, etc) or by data transformations they perform (e.g., an AND functional block can mask a bit flip of an operand if the not-faulted operand is a 0).

Computing masking probability can be performed according to two strategies: *operand* analysis or fault injection. The former consists of analyzing the mathematical operations resources are involved. In most of the cases this operation is simple and it is not time consuming. On the contrary, the latter method requires a bigger effort. In fact, a fault injection

campaign is required to analyze the system resources error resiliency. Error can be injected at RTL level as well as at gate level. However these operations can be done just once, as when masking probability of a resource is known, it can be employed in all the system the resource is part of.

To understand how to to build a BN model, let us consider the simple program trace composed of two instructions that is reported in Fig.1. The first step is to identify those resources belonging to $R_{I_{in}}$ and $R_{I_{out}}$ of each instruction of the trace. In fact, they represent the nodes of the BN. Each node N can assume two possible states: error-free (denoted as N) or faulty (denoted as $\overline{\text{N}}$).

In the example of Fig.1, $I1$ has $R_{I_{in}} = \{R1, R2, AND\}$ and $R_{I_{out}} = \{R3\}$, while $I2$ has $R_{I_{in}} = \{R3, R2, ADD\}$ and $R_{I_{out}} = \{R1\}$. We therefore introduce 8 nodes in the network: AND_{I1}, $R1_{I1}$, $R2_{I1}$, $R3_{I1}$, ADD_{I2}, $R3_{I2}$, $R2_{I2}$, $R1_{I2}$. Once resources nodes are defined, resource dependencies are modeled by means of network edges. Resources belonging to $R_{I_{in}}$ are connected to resources belonging to $R_{I_{out}}$ of the instruction. Moreover, output resources of an instruction may be connected to input resources of a following instruction to model instruction dependency. At this point the BN topology is ready and error nodes, representing the input of the network can be therefore included. Error nodes, when faulty, express the raw probability that an error occurs in a resource. According to our model, input errors can only affect resources belonging to $R_{I_{in}}$.

Each node is then associated to a set of conditional probabilities that quantify the probability of correctness of the node depending on the correctness of the input nodes. Four cases must be considered.

a) Input nodes: Input nodes of the network can be either error nodes or nodes associated to storage resources for which the initial error probability is known. A single probability of correctness is associated to these nodes. In case of soft errors, computational resources can be affected by errors while they are employed. Consequently, the probability of a error-free error node connected to a computational resource is computed according to (2) with a proper λ_{res} and t equal to the execution time of the instruction. On the other hand, storage resources can be affected by external errors during the period of time elapsed between a write and a read operation. Therefore, for error nodes connected to storage resources, the probability of a error-free node is calculated by means of (2) considering a proper λ_{res} and t equal to the period of time between the read and the write of the resource. On the opposite, in case of hard errors, the probability of an error-free error node can be one or zero, depending on the presence of the permanent error regardless the kind of resource the node is connected to.

b) Nodes identifying computational resources in $R_{I_{in}}$ of an instruction (e.g., AND_{I1} in Fig.1): These nodes have a single incoming edge connected to an error node. Two conditional probabilities must be defined as reported in Fig.1 for these nodes where $P(M_{AND})$ identifies the masking probability of the hardware resource. This probability can be computed by simulating the behavior of each instruction with different combinations of operands in input [16]. Masking probability is obtained by injecting faults into input operands and comparing the obtained output with the one without faults. This operation only targets operands and it does not involve the gate and the RTL implementation of the circuit.

c) Nodes identifying storage resources in $R_{I_{in}}$ of an instruction (e.g., $R2_{I2}$ in Fig.1): For these nodes there are two possible causes of faults: (1) the resource was already corrupted in a previous instruction or (2) an error corrupts the resource in the time interval between the execution of two instructions. In this case, four conditional probabilities must be defined as reported in Fig.1 where $P(M_{R2})$ identifies the masking probability of the hardware resource. It is worth to highlight that, in our example, we always consider that, when more than one input node is faulty, the probability of correctness of the current node is zero. This is a worst case assumption that however reduces the complexity of the characterization of the masking probability of each resource.

d) Nodes identifying storage resources in $R_{I_{out}}$ of an instruction (e.g., $R1_{I2}$ in Fig.1): These nodes may have several inputs. Therefore, the number of conditional probabilities to set is equal to the number of possible combinations of states the input nodes may assume. Similar to the previous case, we consider that errors can be masked only when computational resources in $R_{I_{in}}$ are error-free and at maximum a single storage resource in $R_{I_{in}}$ is faulty. In the example, we denote with $P(M_{OPADD})$ the masking probability that the ADD operation performed by the computational resource will mask errors in a input storage resource.

Once conditional probabilities are set up properly for every node, the BN network can be solved and the probability of correct execution of a trace can be estimated. However, only a subset of the instructions of a trace directly affect resources that identify the final outcome of the computation. We denote this subset of instructions as active state instructions (A_I). To compute the probability that a program trace is correct, error probability is taken into account only for output resources of the active state instructions. We define such resources as active resources (A^{res}) and we denote with A_i^{res} the subset of active resources modified by instruction I_i . Given these definitions, the probability for the active instruction I_i to be correct ($P(I_i)$) can be computed as the probability that all its active resources are correct:

$$P(I_i) = \prod_{R \in A_i^{res}} P(R) \tag{3}$$

since the probabilities of correctness of output resources of an instruction are statistically independent.

A program trace T is correct (error-free) if all active state instructions are correct. The probability of correctness of a trace ($P(T)$) can therefore be computed as:

$$P(T) = P(\bigcap_{I \in A_I} I) = \prod_{I \in A_I} P(I|J, \forall J < I) \tag{4}$$

When computing $P(T)$, we need to consider that the event that all active instructions are correct is not statistical independent. We therefore need to multiply the probability of correctness of every active instruction given that all its previous active instructions are correct, $P(I|J, \forall J < I)$. This is possible by setting the evidence in the Bayesian network that all its

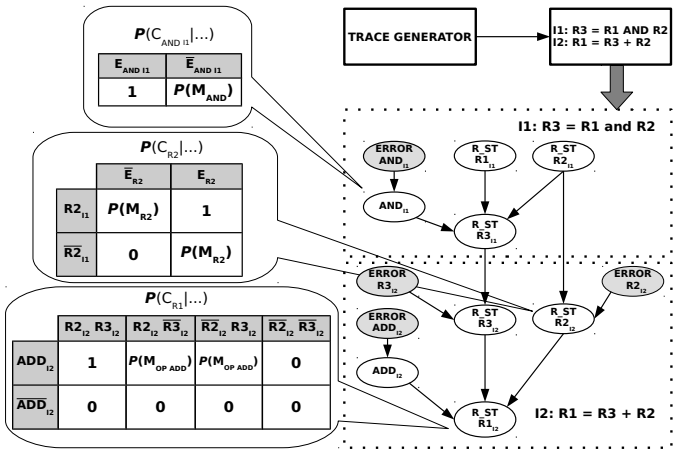


Fig. 1: Example of a bayesian network model for a simple sequence of two instructions.

previous active instructions are correct. The network is then solved and $P(T)$ can be computed.

Once the probability of correctness of a trace is computed, for permanent faults, this value is equal to the final masking probability of the system, $P(M_{System})$. Instead, for transient faults some additional computation is required. In fact, the error rate of the system while executing the trace can be computed by inverting (2) thus obtaining:

$$\lambda^T_{estimated} = -\frac{ln(P(T))}{t_T} \tag{5}$$

where t_T is the execution time of the analyzed program trace. To obtain more accurate estimates of the system SER, λ_{BN}, a simple or a weighted average can be applied to all $\lambda_{estimated}$ of the analyzed traces. Weights of the traces can be set according to the probability that a trace is executed by the analyzed system:

$$\lambda_{BN} = \sum_{T \in analyzed\, traces} \lambda^T_{estimated} \times w_i \tag{6}$$

where $\sum w_i = 1$. Finally, to compute the masking probability of the system:

$$P(M_{System}) = 1 - \frac{\lambda_{BN}}{\lambda_{comp}} \tag{7}$$

III. Experimental results

This section presents the results obtained by implementing the presented Bayesian network model in a reliability analysis tool, and by applying it to a test program executed on a given microprocessor architecture.

A. Framework implementation

We implemented a complete automatic framework able to perform the reliability analysis described in Section II. The framework is composed of two main modules: (i) the *trace generator*, and (ii) the *Bayesian network analyzer*.

The trace generator is built on top of the MARSSx86 [23] full system, cycle-accurate architectural simulator. It simulates the execution of the target program on the x86-64 architecture. During the execution, a detailed trace of the list of executed instructions, and for each instruction the list of resources (e.g., physical register or memory virtual addresses) that are involved in the execution is generated.

Generated traces are analyzed by the Bayesian network analyzer in order to build the Bayesian model of the trace and to estimate the related SER. The Bayesian network analyzer is built on top of SMILE [24], a free C++ framework for the analysis of Bayesian models. It is important to highlight here that, when analyzing the network of a real application composed of hundreds of thousands of instructions, the size of the related network may increase up to a level that saturates the available computational resources. To overcome this limitation, thanks to conditional probability offered by BNs, the networks of the trace is split into several sequential subnetworks each depending on the probabilities computed by the previous network. By applying this iterative approach, scalability of the reliability analysis on complex applications can be achieved with very limited computation time.

B. Experiment setup

To validate the proposed reliability estimation methodology we set up four case studies. They consist of an application programs running on the x86-64 architecture obtained from the MiBench benchmarks [21]. They are *qsort* (32 traces), *aes* (32 traces) and *sha* (16 traces) for transient errors, while *sha* (16 traces) for permanent errors. For each experiment several traces have been analyzed. For a preliminary analysis, hardware faults have been just injected into microprocessor physical registers belonging to the Integer Register File.

Traces have been extracted resorting to the MARSSx86 simulator [23] employed in the Fault Injector tool described in Section III-B2. The tool has the ability to trace various micro-architectural events (such as committed instruction sequence, memory access pattern) which can be reassembled to build the actual execution trace of a program.

To validate the proposed Bayesian Network analyzer, the same benchmarks with the related workloads are analyzed resorting to the proposed Bayesian model and resorting to an extensive architectural fault injection campaign. Computed results are then compared to evaluate the accuracy and the performance of the proposed model.

1) Bayesian model: In order to set conditional probabilities for BN nodes some preliminary operation must be performed. First of all, instructions of the x86-64 architecture must be analyzed. Masking probabilities are evaluated according to the *operands* analysis explained in Section II. Secondly, since the proposed Bayesian model addresses the ISA and faults are injected at micro-architectural level, some precaution must be adopted. This operation requires an analysis of the physical system to be evaluated. In our experiments faults are only injected into the physical register file instead of ISA registers. To overcome this issue we decide to adopt a strategy to tune input error probabilities for ISA registers. In the x86-64 architecture a register rename table keeps the data regarding the renaming process of each physical register into an ISA register. When permanent faults are addressed, input error probability of the faulted resource is set to the probability the register is mapped to the faulted physical register

$$P(Error)\frac{1}{\#\_of\_PHYS\_REGs} \tag{8}$$

instead of being set to one.For transient errors, as register renaming is dynamic, we assume that the number of physical registers that are involved in the computation at the same time is equal to the number of architectural registers. In other words, we assume that the area of the architectural register file, A_{ARCH_RF}, can be evaluated as:

$$A_{ISA\_RF} = A_{PHYS\_RF} \times \frac{\#\_of\_ISA\_REGs}{\#\_of\_PHYS\_REGs} \tag{9}$$

where A_{PHYS_RF} is multiplied by the number of ISA registers over the number of physical registers. Moreover, we assume that all registers have the same size. As a consequence the area of an architectural register, A_{isa_reg}, is the ratio between the number of ISA registers and A_{ISA_RF}.

Active state instructions in a trace are those instructions storing the result of the computation in memory, and they are identified based on the results of the trace generator.

2) Fault Injection: Fault injection experiments have been performed using the MARSSx86-FI [22] fault injector built on top of MARSSx86 [23] full system, cycle-accurate architectural simulator. MARSSx86-FI is capable of injecting single and multiple transient (bit-flip), intermittent (stuck-at-0, stuck-at-1), permanent (stuck-at-0, stuck-at-1) faults, or a mixture of them to the micro-architecture structures of the x86-64 microprocessor architecture. Furthermore, MARSSx86 full system simulation feature provides the capability to monitor the propagation of a hardware fault to the upper levels of system stack: the application output. The extracted output files can be analyzed to classify the injected faults: when no mismatch at the application output is detected, the application execution is labeled as correct.

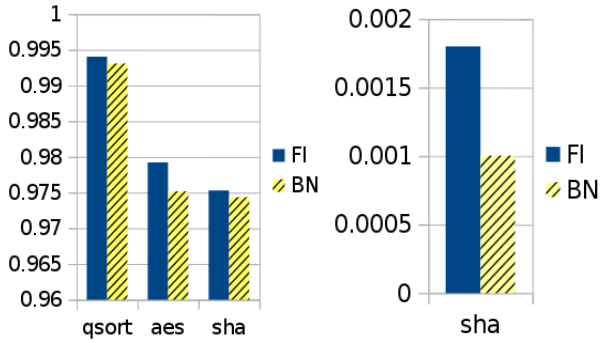
We perform a statistical fault injection campaign, on MARSSx86-FI (see Table I), adopting the statistical sampling of [25]. The methodology computes the number of injection experiments in an array of given size under confidence level[1] and error margin[2] requirements. Using [25], we compute a fault population for 99% confidence and 3% error margin. The calculation leads to a total of 1843 different single-bit transient faults, which are generated for the injections on the integer physical register file. Each transient fault is then modeled as a bit-flip placed in a randomly selected position (i.e., a bit in a physical register) at randomly selected clock cycle.

TABLE I: Marssx86-64 microprocessor model configuration

Parameter	Setting
Fetch/Issue/Commit	4/4/4 instructions per cycle
Combined Predictor	16KB (64K entries, 2 bits/entry, 16 bits BHR)meta pred.: 64K entries
Physical Register File	256 INT; 256 FP; 16 Store; 24 Branch
Reorder Buffer	128 entries
Functional Units	4 clusters (ALUs: 2 INT, 2 FPU; 4 AGUs)
Cache Memories	L1-D (32KB, 4-way, WB) L1-I (32KB, 4-way, WB) L2 (1MB, 16-way, WB)

C. Results

Reliability estimate of the analyzed program traces are compared for the fault injection and the Bayesian model. Figure 2 compares accuracy of the two methods. We can state results show that Bayesian model estimations are very close to the FI ones. In particular, they belong to the uncertainty range of 3% according to [25].



(a) Transient faults (b) Permanent faults

Fig. 2: Estimations of masking probability for both the Bayesian Network and the Fault Injection approaches

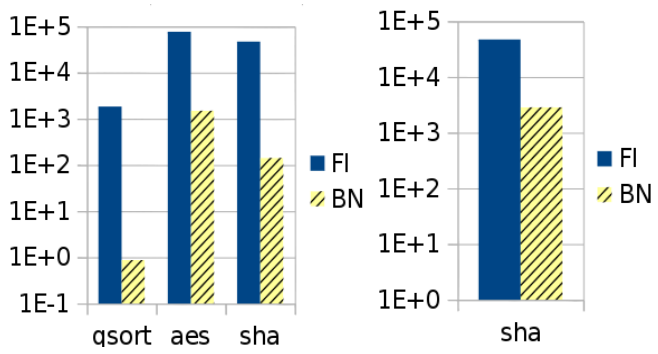
Finally, Figure 3 compares simulation time for each experiment. The figure clearly shows that, resorting to the statistical model, estimation time is reduced by several orders of magnitude thus enabling very fast estimations.

IV. Conclusion

This paper proposes a new statistical approach for the estimation of the reliability of a microprocessor-based system, taking into account the interaction between the hardware and

[1]Probability that the observed sample contains the measured attribute's real mean in the full population

[2]Maximum expected difference between the population's mean value and a sample's mean value of the measured attribute.



(a) Transient errors (b) Permanent faults

Fig. 3: Timing performance comparison of simulation time of a single trace for both the Bayesian Network and the Fault Injection approaches. Time is expressed in seconds

the software layer. Preliminary experimental results performed on the MiBench benchmarks clearly show that the proposed approach is able to provide an accurate and fast estimations when compared to a similar analysis performed using micro-architectural level fault injection. The ability of providing fast reliability evaluations, considering both the hardware and the software layer, is a key feature to enable optimized designs, leading to a progressive avoidance of common practices employed to reach high reliability levels, such as worst case design.

The proposed Bayesian model can be easily adapted to other instruction set architectures as the only requirement to is a profiler able to track the sequence and the execution time of every instruction performed by the target system. In order to reach a comprehensive reliability statistical analysis, further investigations can address the modeling of micro-architectural resources other than the registers file, as the ROB, the LSQ and the BTB.

REFERENCES

[1] R. Baumann, "Soft errors in advanced computer systems," *Design & Test of Computers, IEEE*, vol. 22, no. 3, pp. 258–266, 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*. ACM, 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 IEEE/ACM International Symposium on Microarchitecture*. IEEE Computer Society, 2003, p. 29.

[4] D. Ernst, S. Das, S. Lee, D. Blaauw, T. Austin, T. Mudge, N. S. Kim, and K. Flautner, "Razor: circuit-level correction of timing errors for low-power operation," *IEEE Micro*, vol. 24, no. 6, pp. 10–20, 2004.

[5] R. Vadlamani, J. Zhao, W. Burleson, and R. Tessier, "Multicore soft error rate stabilization using adaptive dual modular redundancy," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*. IEEE, 2010, pp. 27–32.

[6] M. Dimitrov and H. Zhou, "Unified architectural support for soft-error protection or software bug detection," in *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*. IEEE Computer Society, 2007, pp. 73–82.

[7] N. Nakka, G. P. Saggese, Z. Kalbarczyk, and R. K. Iyer, "An architectural framework for detecting process hangs/crashes," in *Dependable Computing-EDCC 5*. Springer, 2005, pp. 103–121.

[8] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, L. Tagliaferri, and C. Tibaldi, "Promon: a profile monitor of software applications," in *8th IEEE International Workshop on Design and Diagnostics of Electronic Circuits and Systems 2005. DDECS 2005.* IEEE, 13-16 April 2005, pp. 81–86.

[9] L. Rashid, K. Pattabiraman, and S. Gopalakrishnan, "Towards understanding the effects of intermittent hardware faults on programs," in *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, June 2010, pp. 101–106.

[10] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou, "Understanding the propagation of hard errors to software and implications for resilient system design," *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 2, pp. 265–276, Mar. 2008.

[11] S. K. S. Hari, S. V. Adve, H. Naeimi, and P. Ramachandran, "Relyzer: Exploiting application-level fault equivalence to analyze application resiliency to transient faults," *SIGPLAN Not.*, vol. 47, no. 4, pp. 123–134, Mar. 2012.

[12] M.-L. Li, P. Ramachandran, U. Karpuzcu, S. K. S. Hari, and S. Adve, "Accurate microarchitecture-level fault modeling for studying hardware faults," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, Feb 2009, pp. 105–116.

[13] V. Sridharan and D. Kaeli, "Eliminating microarchitectural dependency from architectural vulnerability," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, Feb 2009, pp. 117–128.

[14] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," *ACM Sigplan Notices*, vol. 40, no. 6, pp. 190–200, 2005.

[15] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt, "Reducing the soft-error rate of a high-performance microprocessor," *Micro, IEEE*, vol. 24, no. 6, pp. 30–37, Nov 2004.

[16] A. Savino, S. Carlo, G. Politano, A. Benso, A. Bosio, and G. Di Natale, "Statistical reliability estimation of microprocessor-based systems," *Computers, IEEE Transactions on*, vol. 61, no. 11, pp. 1521–1534, Nov 2012.

[17] H. Langseth and L. Portinale, "Bayesian networks in reliability," *Reliability Engineering & System Safety*, vol. 92, no. 1, pp. 92–108, 2007.

[18] L. Yuan-Shun Dai and K. Trivedi, "Performance and Reliability of Tree-Structured Grid Services Considering Data Dependence and Failure Correlation," *Computers, IEEE Transactions on*, vol. 56, no. 7, pp. 925–936, 2007.

[19] S. Zhai and S. Z. Lin, "Bayesian networks application in multi-state system reliability analysis," *Applied Mechanics and Materials*, vol. 347, pp. 2590–2595, 2013.

[20] A. Bobbio, L. Portinale, M. Minichino, and E. Ciancamerla, "Improving the analysis of dependable systems by mapping fault trees into bayesian networks." *Rel. Eng. & Sys. Safety*, vol. 71, no. 3, pp. 249–260, 2001.

[21] University of Michigan at Ann Arbor. Mibench version 1.0. [Online]. Available: http://www.eecs.umich.edu/mibench/

[22] N. Foutris, M. Kaliorakis, S. Tselonis, and D. Gizopoulos, "Versatile architecture-level fault injection framework for reliability evaluation: A first report," in *On-Line Testing Symposium (IOLTS), 2014 IEEE 20th International*. IEEE, 2014, pp. 140–145.

[23] A. Patel, F. Afram, S. Chen, and K. Ghose, "Marss: a full system simulator for multicore x86 cpus," in *Proceedings of the 48th Design Automation Conference*. ACM, 2011, pp. 1050–1055.

[24] M. J. Druzdzel, "Smile: Structural modeling, inference, and learning engine and genie: a development environment for graphical decision-theoretic models," in *AAAI/IAAI*, 1999, pp. 902–903.

[25] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *Design, Automation Test in Europe Conference Exhibition, 2009. DATE '09.*, April 2009, pp. 502–506.