Statistical Reliability Estimation of Microprocessor-Based Systems

Authors: Savino A., Di Carlo S., Politano G., Benso A., Di Natale G., Bosio A.,

Published in the IEEE Transactions on Computers Vol. 61 ,No. 11, 2012, pp. 1521-1534.

Statistical reliability estimation of microprocessor-based systems

A. Savino, S. Di Carlo, *Member, IEEE*, G. Politano, A. Benso, *Senior Member, IEEE,* A. Bosio, *Member, IEEE* and G. Di Natale, *Member, IEEE*

Abstract—What is the probability that the execution state of a given microprocessor running a given application is correct, in a certain working environment with a given soft-error rate? Trying to answer this question using fault injection can be very expensive and time consuming. This paper proposes the baseline for a new methodology, based on microprocessor error probability profiling, that aims at estimating fault injection results without the need of a typical fault injection setup. The proposed methodology is based on two main ideas: a one-time fault-injection analysis of the microprocessor architecture to characterize the probability of successful execution of each of its instructions in presence of a soft-error, and a static and very fast analysis of the control and data flow of the target software application to compute its probability of success. The presented work goes beyond the dependability evaluation problem; it also has the potential to become the backbone for new tools able to help engineers to choose the best hardware and software architecture to structurally maximize the probability of a correct execution of the target software.

Index Terms—Index Terms— Microprocessor reliability, safety-critical systems, statistical analysis.

1 Introduction

As microprocessor technology scales down to the very deep sub-micron range, high production variability, voltage scaling and high operating frequency increase the hardware susceptibility to (soft) errors [1], [2], [3], [4], [5], [6], [7], [8]. This has a negative impact on the reliability of a wide range of computer-based applications which are critical to our health, safety and financial security. Since 1996 several studies reported cases of large computer system failures caused by cosmic-ray-induced soft-errors [9], [10].

Several techniques have been proposed to protect digital circuits against soft-errors, e.g., radiation-hardened technologies [11], [12], error detection/correction codes [13] and redundant architectures [14], [15]. Software Implemented Hardware Fault Tolerance (SIHFT) also gained attention in the last decade [16], [17]. These techniques have a negative impact on systems' performance, power consumption, area and design complexity. Their application must therefore be carefully evaluated depending on the soft-error rate of the target system.

Unfortunately, tools and techniques to estimate the susceptibility of a computer system to soft-errors, taking into account both the hardware and the software domain, are not readily available or fully understood. The execution of a program may mask a large amount of soft-errors. In fact, at the system level soft-errors do not matter as long as the final outcome of the program is correct. To efficiently trade-off between fault tolerance cost and system reliability one has to ask: what is the probability of a program P to have a correct execution state given a certain hardware (raw) soft-error rate? Fault injection is a viable solution to answer this question [18], [19], [20]. However, it can be very expensive and time consuming.

This paper proposes the baseline for a new methodology to estimate computer-based systems reliability against soft-errors. The target microprocessor is first characterized to profile the probability of successful execution of each instruction of its Instruction Set Architecture (ISA). A static and very fast analysis of the control and data flow of the executed software is then performed to compute its probability of successful execution in case of soft-errors. The presented method has the potential to help engineers to choose the best hardware and software architecture to minimize the impact of soft-errors on the system's reliability.

This paper is organized as follows. Section 2 shortly overviews the related literature, while Sections 3 and 4 present the proposed model whose experimental validation is given in Section 5. To conclude, Section 6 introduces future improvements and Section 7 summarizes the main contributions of the paper.

2 Related Works

Previous works on the estimation of the Soft-Error Rate (SER) of an IC can be classified into three categories, namely *circuit-level*, *gate-level*, and *architectural-level*.

Circuit-level SER estimation tries to estimate the probability of an error (glitch) at the output of a logic gate hit by a particle. This is mandatory to define technological mitigation techniques to soft-errors [21], [22].

A. Benso, S. Di Carlo, G. Politano, and A. Savino are with the Department of Control and Computer Engineering, Politecnico di Torino, Corso Duca degli Abruzzi 24, I-10129 Torino, Italy. E-mail: {alfredo.benso, stefano.dicarlo, gianfranco.politano, alessandro.savino}@polito.it.

A. Bosio and G. Di Natale are with the Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier, University of Montpellier II/CNRS, 161, rue Ada, 34392 Montpellier Cedex 5, France. E-mail: {alberto.bosio, giorgio.dinatale}@lirmm.fr.

Gate-level SER estimation moves the focus to the nodes of a netlist [23]. Estimating the error susceptibility of a node requires computing the probability of sensitizing the node with an input vector able to propagate the erroneous value to one of the outputs of the circuit [24]. This however requires the simulation of several random vectors whose number significantly increases with the size of the circuit [22], [25], [23], [21], [26], [27].

Only recently the research on SER estimation has moved from circuit and gate-level to the architectural level [28], [29], [30], [31], [32]. The *Architectural Vulnerability Factor* (AVF) expresses the probability of a system error caused by a raw error in a particular hardware structure [29]. In fact, several raw errors occurring at the device/circuit level are masked at the architectural level (e.g., 85 percent, Wang et al. [31]) due to low resource utilization and introduction of computational blocks that affect performance but not correctness (e.g., branch prediction unit). Several publications propose methods to estimate the AVF of a functional block [28], [29], [30], [31], [32], [33], [34]. An interesting solution that includes the software layer is provided by Sridharan and Keli [35]. They propose to compute a Program Vulnerability Factor (PVF) for a set of benchmarks that can be then used to save computation while calculating the AVF of several microprocessors. Differently to what we propose in this paper, the final software workload is not explicitly considered. Only a few publications try to introduce this concept [36] [37]. However, apart from using fault injection, to the best of the authors' knowledge, a very efficient algorithm to estimate the error probability of a computer system taking into account its hardware, architecture, and running software, is still missing, thus motivating the research proposed in this paper.

3 Soft-error and system models

This section shortly introduces the soft-error model considered in this work, together with some basic concepts required to perform the proposed reliability analysis.

3.1 Soft-errors model

Neutron radiations from cosmic rays, alpha particles from packaging materials and environmental/design variations are common causes of perturbations of digital circuit's nodes that manifest as current pulses of very short duration. If this happens in the hold state of a memory cell or in a flip-flop, the content of the storage element is flipped, causing a soft-error. This model is referred to as Single-Event Upset (SEU) and represents the target error model of this work. Perturbations can also cause a glitch in a combinational node of a circuit causing a Single-Event Transient (SET). If a SET is latched into a sequential logic unit, it then manifests as a SEU. SETs have been considered for long time negligible due to different natural masking effects [38], but are becoming a significant source of errors as technology nodes scale down below 100nm [38], [39].

3.2 Soft-error rate and system modelling

The raw soft-error rate (SER) of an electronic component, also denoted with $\lambda_{comp}(t)$, is the rate at which the device encounters or is predicted to encounter soft-errors. Vendors express the SER either as number of failures-in-time (FIT) or as mean-time-between-failures (MTBF). SER can be used together with a probability distribution to define a *reliability function* $\mathcal{R}(t)$ and a *failure function* $\mathcal{F}(t)$ providing respectively the probability of no error (success) and failure of the component before time t [40]. Given a constant raw error rate λ_{comp} the exponential distribution is a good approximation to model the reliability of an electronic device:

$$\mathcal{R}(t) = e^{-\lambda_{comp} t} \quad (1)$$

$$\mathcal{F}(t) = 1 - \mathcal{R}(t) = 1 - e^{-\lambda_{comp} t} \quad (2)$$

Other distributions such as the Weibull distribution or log-normal distribution can be used when raw error rates are not constant. In practice, λ_{comp} is small enough to reasonably allow considering these two probabilities as constants over a period of time as short as the execution of a program P. In this paper, T_M (*mission time*) denotes the time during the life of the component at which its reliability is evaluated, and $\mathcal{R}(T_M)$ and $\mathcal{F}(T_M)$ denote its raw probability of success and failure at that time.

Among the different devices that constitute a computer systems, the microprocessor is by far the most critical and complex component. A microprocessor can be split into two set of resources called storage elements and operators. The set $S = \{s_i \mid i \in [1, \#S]\}$ of storage elements includes registers and memory elements where data processed by instructions are stored. The set $OP = \{op_i \mid i \in [1, \#OP]\}$ of operators contains all remaining microprocessor blocks (e.g., control state machines, arithmetical units, branch prediction units, etc.) that are used during the execution of an instruction to process data contained into storage elements. Assuming an equal spatial distribution of failures in a component, the raw error rate of a resource λ_{res} is a portion of λ_{comp} proportional to the fraction of its silicon area A_{res} over the total area of the component A_{comp}: $\lambda_{res} = \lambda_{comp} {}^{A_{res}}/_{A_{comp}}$

According to the models proposed in this section, by denoting with C_{res} the not-deterministic event "the resource res is correct at time $0 \leq t \leq T_M$", and with NC_{res} its complementary event, then the raw probabilities of success and failure of a resource at T_M are:

$$P\left(C_{res}\right) = \mathcal{R}(T_M)|_{\lambda=\lambda_{res}} = e^{-\lambda_{comp} \cdot \frac{A_{res}}{A_{comp}} \cdot T_M} \quad (3)$$

$$P(NC_{res}) = \mathcal{F}\left(T_M\right)|_{\lambda=\lambda_{res}} = 1 - e^{-\lambda_{comp} \frac{A_{res}}{A_{comp}} T_M} \quad (4)$$

These two probabilities represent the basis of our reliability estimation model. As a first approximation, the set of events $\{C_{res} \mid res \in (OP \cup S)\}$ can be considered independent. Dependencies are in fact introduced by the execution of the program, and they will be taken into account in the model by the introduction of specific heuristics.

4 Software reliability estimation

In the context of this work, estimating the failure probability of a computer system running a program P means estimating the probability of observing an error in the outcome of the program (assumed bugs-free) running on a hardware system affected by soft-errors only. Soft-errors in the hardware may be masked either because they affect idle resources, or because the program's execution somehow overwrites the error. Based on this assumption, this section introduces an analytical model to estimate the probability of success of a program in presence of soft-errors in the hardware.

4.1 Analytical model

Programs are analyzed using the concept of *program traces*.

Definition 1. *A program trace is an ordered sequence of k instructions (k-tuple) executed while running a program P:* $T = \langle I_1, I_2, \ldots, I_k \rangle$ *with* $I_i \in ISA, \forall 1 \leq i \leq k$ *(ISA* identifies the Instruction Set Architecture of the target microprocessor).

Program traces are a general concept used with many variants in software engineering whenever the sequence of instructions executed by a program must be recorded or statically computed to perform further analysis [41].

The probability of success of a trace T can be predicted by computing the probability of a correct program's outcome during the execution of each instruction of the trace. The outcome of the program is usually a portion of the entire state of the system. This is modelled introducing the concept of *active state*.

Definition 2. *The active state of a program P during the execution of an instruction* $I \in T$*, denoted with* $A_I^T \subseteq S$*, is the set of storage elements representing the outcome of the program when executing the instruction I.*

Programs whose output is evaluated only once at the end of the execution have a not-empty active state only for the last instruction of the trace; programs whose output is continuously evaluated have a not-empty active state for each instruction of the trace. The definition of the active state is application dependent and usually defined by the programmer. In the worst case scenario, the complete state of the system can be considered as active.

The execution of each instruction of a trace may propagate or mask errors among resources, thus modifying their raw probability of success defined in (3). This propagation must be evaluated by modelling the way executed instructions react to soft-errors in the hardware. An instruction $I \in T$ can be modeled as a triplet $I = \langle OUT_I, \mathcal{OP}_I, \mathcal{IN}_I \rangle$ where:

- $OUT_I = \{out_1, \ldots, out_z \mid out_i \in S\}$ is the set of storage elements updated by the instruction (outputs),
- $\mathcal{OP}_I : OUT_I \longmapsto O \subseteq OP$ is a function that defines, for each output, the set O of operators required for its computation, and
- $\mathcal{IN}_I : OUT_I \longmapsto J \subseteq S$ is a function that defines, for each output, the set J of storage elements (operands) required for its computation.

All instructions include the program counter into OUT_I since the execution of an instruction always updates this register. Errors in the control-flow of the program can be considered including the program counter into the active state.

Let us denote with ex a stochastic variable indicating the execution of the instruction $I_{ex} \in T$ and with $f \in [1, k]$ (k denotes the number of instructions of the trace) a stochastic variable indicating that a soft-error occurs in the hardware during the execution of the instruction $I_f \in T$. The probability of success of each storage element $s \in S$ given that $ex = f$, i.e., a soft-error manifests in the hardware while the instruction is executed, can be computed as follows:

1) the initial probability of success of each storage element $s \in S$ ($P'(C_s)$), is the probability of an error-free resource (event C_s) or a resource with an error (event NC_s) that is masked by the hardware:

$$P'(C_s) = P(C_s \cup (M_s \cap NC_s)) = \\ = P(C_s) + P(M_s)P(NC_s) \tag{5}$$

M_{res} is the event: "an error in res is masked" and $P(M_{res})$ is the error masking probability of the resource;

2) the final probability of success of each output $s \in OUT_{I_f}$ of the instruction ($P(C_s \mid ex = f)$), is computed considering that an output is correct if: (i) all operators required for its computation are error-free or able to mask the error (this event is denoted with $C_{\mathcal{OP}_{I_f}(s)}$ and its probability defined in (8)) and (ii) all operands required for the computation are error-free (events C_{in}, $\forall in \in \mathcal{IN}_{I_f}(s)$) or able to mask the error. This is formally expressed in the following equation:

$$\begin{aligned} P(C_s \mid ex = f) = P\Bigg(C_{\mathcal{OP}_{I_f}(s)} \bigcap \Bigg\{ \Bigg[\bigcap_{\forall in \in \mathcal{IN}_{I_f}(s)} C_{in} \Bigg] \\ \bigcup \Bigg[DM_{I_f} \bigcap \Bigg(1 - \bigcap_{\forall in \in \mathcal{IN}_{I_f}(s)} C_{in} \Bigg) \Bigg] \Bigg\} \Bigg) = \\ = P\left(C_{\mathcal{OP}_{I_f}(s)}\right) \cdot \Bigg\{ \prod_{\forall in \in \mathcal{IN}_{I_f}(s)} P'(C_{in}) + \\ + P\left(DM_{I_f}\right) \cdot \Bigg(1 - \prod_{\forall in \in \mathcal{IN}_{I_f}(s)} P'(C_{in}) \Bigg) \Bigg\} \end{aligned} \tag{6}$$

DM_{I_f} represents the event: "the execution of I_f masks an error in one of its operands" and $P\left(DM_{I_f}\right)$ is the probability that an instruction masks an error in its operands. This probability can be computed either with fault injection experiments or, as explained later in this paper, by an analytical analysis of each instruction;

3) the final probability of success of all storage elements not in the output set of the instruction ($\forall s \in S - OUT_{If}$) is not affected by the execution and is computed as follows:

$$P\left(C_s \mid ex = f\right) = P'\left(C_s\right) \tag{7}$$

$P\left(C_{\mathcal{OP}_{I_f}(s)}\right)$ used in (6) denotes the probability of success of all operators used to compute the resource s. Similarly to (5), it can be computed as the probability of respecting, for all considered operators, the following conditions: (i) the operator is error-free or, (ii) the operator manifests an error but the error is masked. This can be formalized as follows:

$$\begin{aligned} P\left(C_{\mathcal{OP}_{I_f}(s)}\right) =& P\left(\bigcap_{\forall op \in \mathcal{OP}_{I_f}(s)} \left[C_{op} \cup \left(M_{op} \cap NC_{op}\right)\right]\right) = \\ =& \prod_{\forall op \in \mathcal{OP}_{I_f}(s)} \left[P\left(C_{op}\right) + \left(P(M_{op}) \cdot P(NC_{op})\right)\right] \end{aligned} \tag{8}$$

With this model, the contribution of a fault tolerant operator to (8) is: $P\left(C_{op}\right) + \left(1 \cdot P(NC_{op})\right) = \mathcal{R}_{op}(T_M) + \left(1 - \mathcal{R}_{op}(T_M)\right) = 1$. This correctly models that the fault tolerance mechanism resets the contribution of this operator to the error probability of other resources.

Similarly to the case $ex = f$, the probability of success of each resource at the end of the execution of an instruction $I_j \in T$ following I_f (i.e., $ex = j > f$) is computed taking into account that an error in one of the operands can be propagated to one of the outputs:

1) the probability of success of each storage element not in OUT_{I_j} ($\forall s \in S - OUT_{I_j}$) is constant

$$P\left(C_s \mid ex = j \ \wedge \ j > f\right) = P\left(C_s \mid ex = j - 1\right) \tag{9}$$

2) the probability of success of each storage element $s \in OUT_{I_j}$ is the probability that all operands of I_j are correct or that at least one operand of I_j is not correct but the error is masked by the execution of the instruction:

$$\begin{aligned} P\left(C_s \mid ex = j \ \wedge \ j > f\right) =& P\left(\left[\bigcap_{\forall in \in \mathcal{IN}_{I_j}(s)} C_{in}\right]\right. \\ & \left.\bigcup \left[DM_{I_j} \bigcap \left(1 - \bigcap_{\forall in \in \mathcal{IN}_{I_j}(s)} C_{in}\right)\right]\right) \\ =& \prod_{\forall in \in \mathcal{IN}_{I_j}(s)} P\left(C_s \mid ex = j - 1\right) + \\ & + P\left(DM_{I_j}\right) \cdot \left(1 - \prod_{\forall in \in \mathcal{IN}_{I_j}(s)} P\left(C_s \mid ex = j - 1\right)\right) \end{aligned} \tag{10}$$

When evaluating the execution of a trace T, soft-errors may manifest during any of the k instructions of the trace (i.e., $1 \leq f \leq k$). According to our model, the correctness of a resource $s \in S$ during the execution of an instruction I_j ($ex = j$) depends on the instant the soft-error manifests in the hardware (I_f). A set of j error conditions must therefore be analyzed: 1) the error manifests during the first instruction of the trace ($f = 1$), 2) the error manifests during the second instruction of the trace ($f = 2$) and so on until the case ($f = j$). The probability of success of the resource for each error condition can be computed according to (6), (7), (9) and (10). Since all error conditions have the same probability and represent disjoint events, the probability of success of a resource after the execution of a generic instruction I_j can be computed as follows:

$$\begin{aligned} & P\left(C_s \mid ex = j, \forall 1 \leq f \leq j\right) = \\ & = P\left(\left[\bigcup_{x=1}^{j-1} \left((f = x) \cap \left(C_s \mid ex = j \ \wedge \ j > f\right)\right)\right] \cup \right. \\ & \qquad \left.\left[(f = j) \cap \left(C_s \mid ex = f\right)\right]\right) = \\ & = \sum_{x=1}^{j-1} \left(\frac{1}{j}\right) \cdot P\left(C_s \mid ex = j \ \wedge \ j > f\right) + \\ & \qquad + \left(\frac{1}{j}\right) \cdot P\left(C_s \mid ex = f\right) \end{aligned} \tag{11}$$

The j^{th} instruction I_j of a program trace T is considered correctly executed if all storage elements of its active state $A^T_{I_j}$ are error-free. Denoting C_{I_j} the non-deterministic event "the instruction I_j is correctly executed", the probability of success of the instruction given that $A^T_{I_j} \neq \oslash$ is defined as

$$P(C_{I_j}) = P\left(\bigcap_{\forall s \in A^T_{I_j}} \left(C_s \mid ex = j, \forall 1 \leq f \leq j\right)\right) \tag{12}$$

Computing (12) is not trivial, since the execution of an instruction introduces dependencies among storage elements. Alg. 1 proposes an heuristic to evaluate these dependencies. It produces a subset of the active state containing independent resources that can be used to compute equation (12). In Alg. 1, D is a integer matrix with each row corresponding to an instruction of T and each column to one of the storage elements. The condition D[j][i] $\neq 0$ indicates that, during the execution of the j^{th} instruction, the resource i must be discarded when computing (12) since its contribution has already been taken into account in a different set of resources. On the other hand, D[j][i] $= 0$ denotes that the resource must be considered since its contribution was not considered before. When the program starts (j = 1) all resources are independent (Alg. 1, row 2). For a generic instruction I_j, the status of each resource in D is initially set to those of the previous instruction (Alg. 1, row 5), and then the outputs of the instruction are considered. The overall idea is that each output already includes the contribution of the corresponding operands that can therefore be excluded from the set of resources to consider (Alg. 1, row 14-16). If more than one output is computed based on the same set of operands, only one of these outputs must be considered (Alg. 1, row 18-25). Whenever a storage element is written, the operands used during the last instruction targeting the same resource must be considered again (Alg. 1, row 14-16). The array LW stores, for each storage element, the index of the last instruction of the trace where the resource was written. Alg. 1 is an approximated approach to take into account dependencies among resources; however, the experimental results of Section 5 will show that it is able to provide estimations with a reasonable level of confidence.

Algorithm 1 Algorithm to compute the subset of independent resources for an instruction

```
Require: j: index of the evaluated instruction
 1: if j = 1 then
 2:     D[j]=(0,...,0)
 3:     LW=(0,...,0)
 4: else
 5:     D[j]=D[j − 1]
 6: end if
 7: for i = 1 to count $\left(OUT_{I_j}\right)$ do
 8:     $s = OUT_{I_j}$[i]
 9:     if LW[$s$]<>0 then
10:         for all $r$ in $\mathcal{IN}_{I_{\mathrm{LW}[s]}}(s)$ do
11:             D[j][$r$]=D[j][$r$]-1
12:         end for
13:     end if
14:     for all $r$ in $\mathcal{IN}_{I_j}(s)$ do
15:         D[j][$r$]=D[j][$r$]+1
16:     end for
17:     LW[$s$]=j
18:     D[j][$s$]=0
19:     for k = 1 to i − 1 do
20:         $x = OUT_{I_j}$[k]
21:         if $\mathcal{IN}_{I_j}(x) = \mathcal{IN}_{I_j}(s)$ then
22:             D[j][$s$]=1
23:             break
24:         end if
25:     end for
26: end for
```

Based on Alg. 1, the probability expressed in (12) can be estimated as follows:

$$P(C_{I_j}) \cong \prod_{\forall s \in A^T_{I_j} | \mathrm{D[j][s]}=0} P\left(C_s \mid ex = j, \forall 1 \leq f \leq j\right) \quad (13)$$

Given that equation (13) provides the probability of success of each instruction of a trace, the probability of success of the full trace T ($P\left(C_T\right)$) can be approximate as the average probability of success of those instructions characterized by a not empty active state:

$$P\left(C_T\right) \cong \frac{1}{count(I_i \mid A^T_{I_i} \neq \oslash)} \sum_{\forall I_i | A^T_{I_i} \neq \oslash} P\left(C_{I_i}\right) \quad (14)$$

Several traces can be generated by the execution of a program, depending on the specific workload. Let us denote with $\mathcal{T}_P$ the complete set of possible traces of a program P, with each trace $T \in \mathcal{T}_P$ an independent event characterized by an execution probability $P(T)$ and $\sum_{\forall T \in \mathcal{T}_P} P\left(T\right) = 1$. The probability of success of the program P ($P(C_P)$), can be computed as a weighted average of the probability of success of each trace:

$$P\left(C_P\right) = \sum_{\forall T \in \mathcal{T_P}} P\left(T\right) \cdot P\left(C_T\right) \quad (15)$$

Generating the full set of traces of a real application is obviously often not feasible in a reasonable computational time. A subset of all possible traces (TS), must therefore be sampled in order to statistically represent a significant group of execution alternatives. The more traces are sampled, the better (15) will estimate the reliability of the system as the probability of success of the program in presence of soft-errors in the hardware. In order to take into account the contribution of all traces not included in TS, (15) can be rewritten as follows:

$$\begin{aligned} P\left(C_P\right) \cong & \sum_{\forall T \in TS} \left(P\left(T\right) \cdot P\left(C_T\right)\right) + \\ & + (1 - \sum_{\forall T \in TS} P\left(T\right)) \cdot \mathcal{R}\left(T_M\right)|_{\lambda = \lambda_{comp}} \end{aligned} \quad (16)$$

The first portion of (16) computes (15) on TS. The second portion of the formula considers that in all situations not included in TS the probability of success of the program can be approximated to the worst-case represented by the raw reliability of the component defined in (1).

4.2 Program traces generation

Two approaches can be followed to obtain a relevant set of traces for the proposed reliability estimation model.

Whenever a strong, statistically relevant set of inputs for the target software is available, it can be exploited to derive a corresponding set of traces. Several runs of the program are executed, each with a different input, and run-time information about executed instructions and accessed data are recorded to compose each trace. The probability assigned to each trace ($P(T)$ in (16)) can be uniformly distributed or calculated based on the knowledge of the probability of occurrence of the corresponding inputs. However, in several situations in which very early design exploration is performed, a statistically relevant set of inputs might not be available, or it might be difficult to estimate how much it covers the set of possible executions. For these situations, this paper presents an algorithm that generates a set of traces by performing a static analysis of the program's binary code. The goal of this algorithm is to cover as many parts as possible of the control-flow graph of the application, providing also a metric to measure how many of the possible paths have been covered.

The control-flow graph of a program P, is a labeled directed graph $CFG_P = (Instr, A, L)$ where:

- $Instr = \{I_i \mid I_i \in ISA\}$ is the set of nodes of the graph, with each node representing a single instruction of the program,

- $A = \{(I_i, I_j) \mid I_i, I_j \in ISA\}$ is the set of arcs modelling allowed sequences of instructions, and
- $L : A \rightarrow labels$ is a function that maps each arc to a label.

Each CFG has two special nodes denoted with I_{start} and I_{end} representing the entry point and the exit point of the program. Multiple exit points are connected to a single node. In our model, the label of an arc (I_i, I_j) is the probability $p_{i,j}$ of crossing the arc during the execution of the program. $p_{i,j}$ can be assigned applying the following policies:

1) If I_j is the only direct successor of I_i, and therefore it is not a branch instruction, the probability of crossing the arc (I_i, I_j) is equal to 1;
2) If I_i has m direct successors on the graph, i.e., there are m arcs directed from I_i, and no run-time information about the probabilities of crossing each arc is available, then each arc is assigned a probability equal to $\frac{1}{m}$;
3) If I_i has m direct successors, and from the knowledge of the program or from run-time information it is possible to conclude that some of the arcs are less probable than others (e.g., arcs that terminate the program in case of errors), custom probabilities can be assigned given that the sum of the probabilities of the arcs directed from the node must be equal to 1. Variable probabilities can be also assigned, modelling for instance loops that start with a high probability that decreases when the number of iterations increases.

The control-flow graph of a program can be automatically generated by statically analyzing its binary code with tools such as Diablo [42].

A modified depth-first search algorithm on the CFG of the program named Traces Generation Algorithm (TGA) is used to statically compute a set of execution traces (Alg. 2). The main problem of this approach is that, in the case of loops, the number of traces that can be generated is theoretically infinite. A set of terminating conditions is therefore introduced to stop the generation either when the computed traces provide the desired coverage of the CFG, or when a maximum number of traces has been generated.

TGA is a recursive algorithm that requires the following set of global variables:

- TS: the set of generated traces. It is an empty set when the algorithm starts;
- TARGET_CEP: according to (15), each trace is associated with an execution probability $P(T)$. If all possible traces of a program can be generated, their cumulative execution probability (CEP) is equal to 1, thus guaranteeing the full coverage of all execution paths. When instead, the number of possible traces is theoretically infinite, TARGET_CEP is the minimum cumulative execution probability that has to be reached before stopping the trace generation. This value is also used as a metric of the completeness of the generated set of traces;
- MAX_T: is an upper bound on the number of generated traces. It forces the algorithm to stop even if TARGET_CEP has not been reached;
- STOP_IF_ALL_ARCS_COVERED: if set to true, this flag allows to stop the generation when all arcs of the CFG have been traversed at least once. This represents the minimum set of traces that must be considered to analyze a program. It can be used for early and very fast evaluations.

Algorithm 2 Traces Generation Algorithm

```
Require: TS ← ∅,
    TARGET_CEP ← [0,1],
    MAX_T,
    CEP ← 0,
    STOP_IF_ALL_ARCS_COVERED ← {true,false}
 1: TGA (node = $I_{start}$ , T = ∅, prob=1)
 2: T ← T ∪ node
 3: if node = $v_{end}$ then
 4:     TS ← TS ∪ T
 5:     CEP ← CEP + prob
 6:     mark all arcs traversed by T as visited
 7:     if all_arc_visited AND
        STOP_IF_ALL_ARCS_COVERED=true then
 8:         exit
 9:     end if
10:     if |TS| = MAX_T then
11:         exit
12:     end if
13:     if CEP >= TARGET_CEP then
14:         exit
15:     end if
16:     return
17: else
18:     for all I in directed_successors(node) do
19:         newprob ← newprob * $p_{nodo,v}$
20:         TGA (v,T,newprob)
21:     end for
22: end if
```

TGA begins the generation considering the starting node I_s and an empty trace T with execution probability equal to 1 (Alg. 2, row 1). It adds the current node to the trace (Alg. 2, row 2) and then checks if the current node corresponds to I_{end} to detect whether the end of a trace has been reached (Alg. 2, row 3).

In case the current trace is not complete (Alg. 2, rows 18-21), the algorithm selects iteratively each direct successor of the current node and, for each corresponding arc, it generates a new trace by recursively calling itself (Alg. 2, row 20). The probability of the new trace is the product of the current probability by the probability of execution of the arc (Alg. 2, row 19).

In case the current trace is complete (Alg. 2, rows 4-16), it is added to the set TS (Alg. 2, row 4). CEP (Alg. 2, row 5) and the set of arcs traversed at least once (Alg. 2, row 6) is updated. The different terminating conditions are then evaluated. Rows 7-9 terminate the generation if all arcs have been traversed at least once and STOP_IF_ALL_ARCS_COVERED is set to true. Rows 10-12 stop the generation if MAX_T traces have been generated and, finally, rows 13-15 stop the generation if target TARGET_CEP has been reached. If none of these conditions are true, the generation continues exploring additional paths on the graph.

Listing 1 shows a simple example of a program, coded for the Intel 8088™ microprocessor, counting the number of elements of an array. Items are stored in memory at address 0100h and range boundaries are passed through the stack. The program loops until all items are evaluated (CX is used to count the number of items in the array passed in the stack). In order to simplify the example, the program omits any context saving operation .

```
pop cx             ;counter
pop ax             ;get upper & lower
pop bx             ;limits in ax & bx
mov si, 0          ;number counter
mov di, 0100h      ;get initial location
lp: mov dx, word ptr [di] ;get the content
cmp dx, ax         ;check the upper
jle lw             ;if number is lower
jmp nxt            ;if number is larger
lw: cmp dx, bx     ;check lower limit
jge lim            ;if number is larger
jmp nxt            ;if number is lower
lim: inc si        ;increment counter
nxt: add dx, 2     ;get next location
loop lp            ;repeat until items
```

Listing 1. Intel 8088 example program

The CFG of the program is summarized in Fig. 1. No run-time information about the probability of traversing each arc is available. In case of branches, all arcs directed from the node have been assigned with the same execution probability (policies 1 and 2). The CFG shows different paths and a loop.

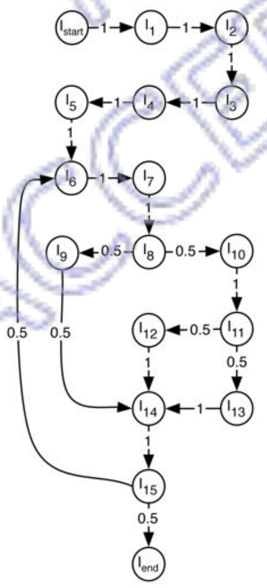


Figure 1. Control-Flow graph statically computed from the binary code of Listing 1

By executing Alg. 2 with STOP_IF_ALL_ARCS_COVERED set to true, the following set of traces is generated:

- $T1 = < I_{start}, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{14}, I_{15}, I_{end} > (P_{T1} = 0.25)$
- $T2 = < I_{start}, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_{10}, I_{11}, I_{12}, I_{14}, I_{15}, I_{end} > (P_{T2} = 0.125)$
- $T3 = < I_{start}, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_{10}, I_{11}, I_{13}, I_{14}, I_{15}, I_{end} > (P_{T3} = 0.125)$
- $T4 = < I_{start}, I_1, I_2, I_3, I_4, I_5, I_6, I_7, I_8, I_9, I_{14}, I_{15}, I_6, I_7, I_8, I_9, I_{14}, I_{15}, I_{end} > (P_{T4} = 0.0625)$

This set allows to reach a CEP equal to 0.5625. Fig. 2 plots how CEP increases by increasing the number of generated traces. By traversing the loop multiple times , i.e., arc (I_{15}, I_6), additional execution alternatives can be evaluated reaching, with about 40 traces, a CEP almost equal to 1.

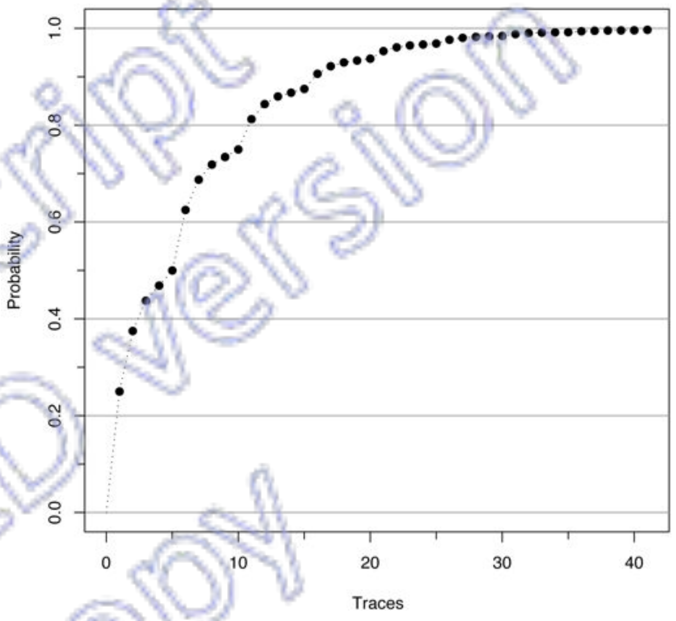


Figure 2. Plot of the cumulative trace execution probability vs. the number of generated traces

5 Experimental model validation

This section presents the experimental setup used to validate the model proposed in the previous sections. It covers three main aspects: the microprocessor characterization, the statistical reliability estimation, and the validation and discussion of the results.

5.1 Microprocessor characterization

The microprocessor characterization is a key operation that must be performed only once, independently from the program that will be executed in the system. Two microprocessor cores have been characterized in this paper: the Intel 8088™ and the OpenRISC1200.

The Intel 8088™ (hereinafter referred to as 8088) has the same architecture of the more famous Intel 8086™ with the only difference being that the external data bus width is reduced from 16-bit to 8-bit. It is a CISC microprocessor with a very simple two-stage pipeline.

It is equipped with 16-bit registers grouped as follows: four general purpose registers (AX, BX, CX, DX) also accessible as eight 8-bit registers; four memory indexing registers (stack-pointer SP, base-pointer BP, source-index SI, destination-index DI); four segment registers (code segment CS, data segment DS, stack segment SS, extra segment ES) and two registers for controlling the execution flow (program counter PC, status flags SF). The 8088 ISA contains 111 instructions without floating-point support. The microprocessor model is provided by the HT-LAB toolkit [43]. This toolkit, distributed under the GNU license, includes the VHDL code of a complete 8088 based system: the processor, the ROM and the RAM, some peripheral devices, and a set of facilities to convert assembly code in a format that can be directly included and executed in the VHDL code.

The OpenRISC1200 (hereinafter referred to as OR1200) is a 32-bit scalar RISC microprocessor with Harvard architecture and 5 stage integer pipeline. It has 32 general purpose 32-bit registers, caches, virtual memory support and basic DSP functions. It supports the ORBIS32 instruction set for a total of 215 instructions. The instruction set includes 32-bit integer instructions, basic DSP instructions, 32-bit Load and Store instructions, program control flow instructions and some special instructions. The VHDL model of the OR1200 is freely available on the OpenCores website (http://www.opencores.org) .

Both processors have been synthesized using Synopsis Design Compiler with the AMS 350nm technology library. The choice of the target library could lead to small fluctuations in the reliability results, but this issue is beyond the scope of this paper. Fig. 3 provides a summary of the area occupation of the two cores that gives an idea of the complexity of the two microprocessors. The 8088 accounts for a total of 652 flip-flops while the OR1200 accounts for a total of 1891 flip-flops, all of them considered as potential target SEU locations.

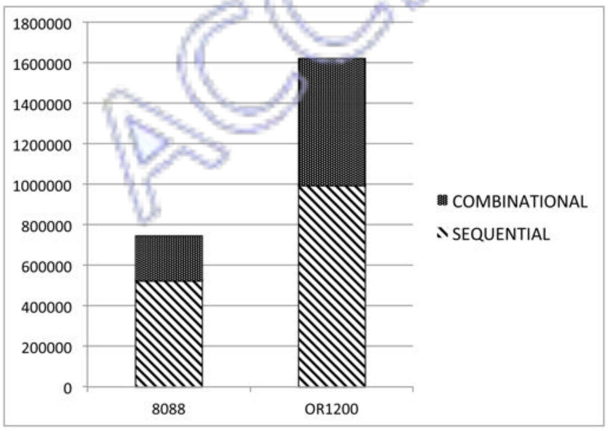


Figure 3. Microprocessors area summary provided in equivalent gates

The microprocessor characterization process estimates the masking probabilities $P(C_{\mathcal{OP}_I})$ and $P(DM_I)$ required to compute equations (6) and (10).

The most efficient way to compute $P(C_{\mathcal{OP}_I})$ is to setup a fault injection campaign. For each instruction `<INSTR>` of the ISA, and for each possible combination of operands, fault injection has been performed with the microprocessor executing a simple program composed of the target instruction preceded and followed by a set of `NOP` instructions. This solution guarantees that `<INSTR>` traverses all stages of the pipeline (2 for the 8088 and 5 for the OR1200) without other instructions interfering with its operations. The same instruction is simulated several times with different operands in order to explore different execution conditions. An average of 10,000 SEUs for the 8088 and 30,000 SEUs for the OR1200 has been injected for each variant of each instruction. At this stage, fault injection only targets operators and it does not include flip-flops associated with operands and output registers that will be considered instead when estimating $P(DM_I)$. Obviously, the size of the fault list, and therefore the length of the fault injection experiment, heavily depends on the number of registers and instructions of the microprocessor.

The effort required to perform this part of the characterization can be extremely variable depending on several factors. The most important ones are the available computational resources, which impact the time required to perform the experiments and elaborate the results, the level of detail of the microprocessor model, which directly affects both the confidence in the generated the fault list and the reliability of its injection, and the chosen Fault Injection mechanism (hardware, software, or simulation-based), which determines the cost and precision of the injection results. Nevertheless, it is worth reminding that having to consider only the microprocessor without any workload but the individual instructions of its ISA, the fault list generation is faster, easier, and more complete because it can be exhaustively generated with a simple software program.

Differently from $P(C_{\mathcal{OP}_I})$, $P(DM_I)$ can be analytically computed by analyzing the behaviour of each instruction without the need of performing VHDL simulations. Instead, a set of C programs exhaustively performs this analysis simulating the behavior of each instruction in presence of faults in its operands. Let's take as an example two instructions, ADD and CMP (compare) for the 8088:

- ADD computes the sum of two 16-bit operands and stores it into a new 16-bit word. Regardless of their value, any error in one of the operands will generate an error in the output result. $P(DM_{ADD})$ is therefore equal to 0.
- CMP compares two 16-bit operands. The result in this case heavily depends on the value of the compared data. By analyzing all possible combinations and errors, a $P(DM_{CMP}) = 0.95$ is obtained. This means that 5% of the errors in the operands will be masked by the instruction itself.

Fig. 4 reports an example of the characterization of a subset of instructions for the two considered processors. For each instruction, the figure reports: 1) the operators area ratio (i.e., the number of flip-flops of the used operators over the total number of flip-flops) required to compute equations (3) and (4) for the operators; 2) the overall operators masking probability $P\left(C_{\mathcal{OP}_I}\right)$ and 3) the data masking probability $P\left(DM_I\right)$. The figure highlights that the 8088 has a lower capability of masking errors in the hardware compared to the OR1200. As shown in the following sections, this will negatively reflect on the reliability at the system level.

5.2 Experiments and validation

Experiments have been conducted on three application programs, two of them (QSORT and AES) obtained from the MiBench Ver. 1.0 benchmarks [44]:

1) *HUFFMAN*: performs Huffman encoding applied to a list of 16 symbols. The result is the Huffman code associated to each symbol.
2) *QSORT*: sorts a given array of integer numbers stored in the main memory using the quick sort algorithm.
3) *AES*: performs AES encryption of a 138-Bytes message.

The reliability of the two microprocessors while running these three benchmarks has been assessed both by applying the proposed estimation model, and by executing a very extensive fault injection campaign aimed at confirming the estimated results. In order to reduce the complexity of the fault injection experiments, the three benchmarks do not contain I/O instructions and all input data are predefined and stored in the RAM along with the program's binary code.

All experiments have been performed on a workstation equipped with a dual Intel Xeon@3.16GHz quad core processor and 32GB of RAM.

The proposed estimation model has been coded in a C program. The tool includes a library of parsers for the assembly language of the two considered microprocessors, and it implements a multi-thread architecture to fully exploit the parallelism offered by the available workstation. Experiments only focused on faults in the microprocessor. For this reason, each instruction that writes data outside the microprocessor (e.g., any instruction storing data in the memory) has a not empty active state. When generating program traces using Alg. 2, the knowledge of the source code is used to assign custom variable probabilities to the different branch instructions. Several golden runs of the program with random data have been performed to obtain an estimation of the most probable branches of the program. This makes it possible to reduce the number of traces required to reach the desired CEP level.

The fault injection campaign has been performed resorting to a custom fault simulation environment developed at LIRMM [45]. To fairly compare performances, the fault injector proposed in [45] has been extended to allow multiprocess simulations. Fault injection experiments have been setup as follows:

1) The overall system, including the microprocessor, RAM, ROM, etc., is simulated and all activities over the primary inputs and outputs of the processors (control signals, data and address buses) are logged in an external file. A table mapping each instruction of the target program to its execution time expressed in terms of clock cycle is also generated.
2) Fault simulation of SEUs in the microprocessor's flip-flops is performed while applying the inputs stored in step 1. An exhaustive fault injection campaign is performed. All possible clock cycles as well as all possible flip-flops have been considered as target fault locations. This makes it possible to reach 100% of confidence in the simulated results.
3) A report is generated starting from the results of the fault simulations. This report summarizes, for each instruction, how many faults are detected and how many are masked.

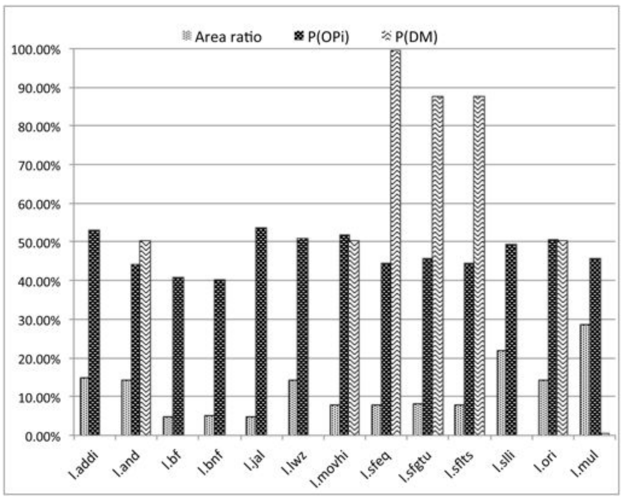
Fig. 5 compares the performance of the proposed method compared to fault injection in terms of CPU time. Results are provided in hours of CPU time using a logarithmic scale. It is evident how the proposed model outperforms fault injection, reducing the computation time by several orders of magnitude. When considering the fault injection campaign, the 8088 is the most critical core in terms of computation time. This is due to the fact that instructions of the 8088 ISA usually require multiple clock cycles to be executed, strongly increasing the simulation time of the synthesized core. For the proposed method the situation is instead inverted. The computation time is mainly affected by the number of instructions composing the program. The OR1200, which like all RISC processors only implements simple instructions, requires more instructions to code a program (see Table 1), therefore making the reliability analysis more time consuming.

Fig. 6 compares the storage requirement for the two methods. Storage requirement is reduced compared with fault injection, especially considering the 8088 whose fault injection requires saving long simulations in terms of clock cycles. Overall, the amount of data to store is quite small and does not represent a critical issue for the analysis.

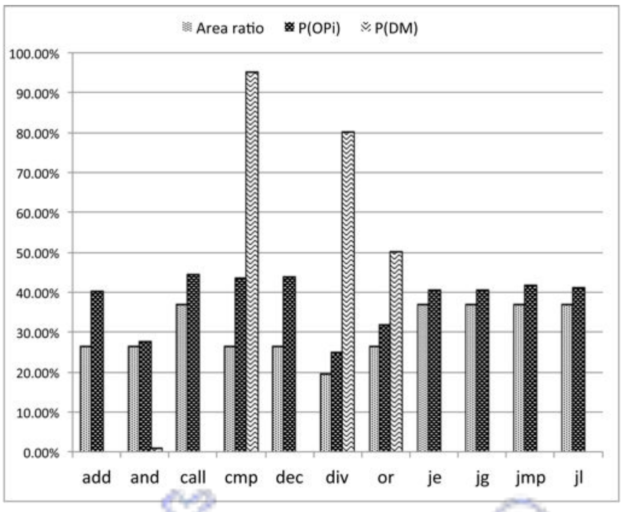
To conclude the description of the experimental setup, Table 1 summarizes the information about the complexity of the different benchmarks in terms of number of instructions and average number of clock cycles for an execution (CC). It also reports the number of traces that have been simulated along with the reached CEP. For all experiments the trace generation algorithm has been executed with a TARGET_CEP=0.95

5.3 Results

Fig. 7 proposes six plots that summarize the results of the reliability analysis performed on the six case studies. Each plot reports the following three curves:



(a) OR1200



(b) 8088

Figure 4. Characterization of a subset of instructions of the (a) OR1200 and (b) 8088 microprocessors.

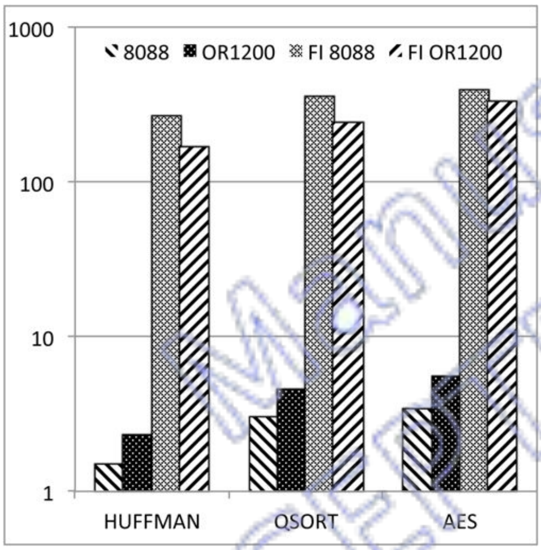


Figure 5. Comparison of the computation time between the proposed model and the fault injection analysis. Time is expressed in hours of CPU and reported using a logarithmic scale.

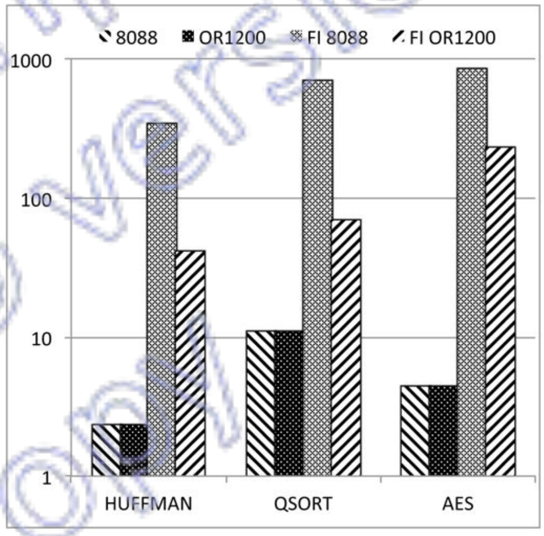


Figure 6. Comparison of the storage requirement between the proposed model and the fault injection analysis. Results are expressed in MBytes using a logarithmic scale.

Table 1
Summary of the experimental setup.

Processor	Benchmark	Instructions	CC	Traces	CEP
8088	HUFFMAN	285	13,273	~1,000	0,95
	QSORT	112	26,891	~5,000	0,95
	AES	3,163	32,652	~2,000	0,95
OR1200	HUFFMAN	453	1,405	~1,000	0,95
	QSORT	180	2,301	~5,000	0,95
	AES	5,758	7,606	~2,000	0,95

- *Raw rel. fun.*: it is the raw reliability function of the microprocessor computed according to (1) for a mission time T_M of 6 years and an error rate $\lambda = 0.019 \cdot 10^{-6}$ for both the 8088 and the OR1200 (the specific value of λ characterizes the technology and does not influence the accuracy of the prediction).
- *FI based rel. fun.*: it is the reliability function estimated considering the results of the fault injection as: $\mathcal{R}(T_M) = e^{-\lambda T_M} + (1 - e^{-\lambda_{8088} T_M}) \cdot P_{mask}$. This function takes into account the probability of having a fault free device at time T_M, and the probability of having a faulty device whose error is masked with a given probability. The masking probability for the given benchmark, reported in Table 2, has been computed according to the fault injection results as the number of masked faults over the total number of injected faults.
- *Estimated rel. fun.*: it is the reliability function estimated with the proposed model considering differ-

ent values of T_M.

Table 2
Summary of fault injection experiments in terms of injected and masked SEUs.

Benchmark	Benchmark	Injected	Masked	Mask Prob
8088	HUFFMAN	8,627,450	4,296,779	~0.49
	QSORT	17,479,150	8,835,476	~0.51
	AES	21,289,114	7,876,973	~0.37
OR1200	HUFFMAN	2,656,855	2,098,916	~0.79
	QSORT	3,560,753	2,892,176	~0.81
	AES	14,382,946	11,362,528	~0.79

Fig. 7 clearly shows that the estimated reliability function is in general able to approximate the fault injection based reliability function, thus confirming the capability of the proposed method to efficiently estimate the reliability of the target microprocessor considering the running program. This can be better appreciated looking at Fig. 8 that reports the error between the estimated reliability function and the fault injection curve. With a reasonable number of traces this error is always lower than 7% guaranteeing a good confidence in the prediction.

Looking at the experimental results, one can notice that the estimation error increases with the increment of the mission time. A portion of this error can be accounted to the approximated characterization of the microprocessor, and to the impossibility of exploring the complete set of possible traces and reaching CEP=1. However, by analyzing the way our model works, the majority of the error is probably introduced by the heuristic used to take into account resources dependencies. This introduces a certain error in the estimation that becomes evident when the the missione time T_M, and consequantly the fault probability of the single resources, increases.

To conclude, Fig. 9 shows how the proposed method can be used to perform very fast early design exploration. It reports the estimated reliability function of the 8088 running HUFFMAN with four different fault tolerant configurations of the microprocessor: (i) all ALU's internal flip-flops are fault-tolerant (ALU-FT), (ii) all microprocessor's user registers are fault-tolerant (REG-FT), (ii) both the ALU and the registers are fault-tolerant (ALU+REG-FT), (iv) all resources of the microprocessor are fault-tolerant (ALL-FT). The four configurations can be easily analyzed by changing the masking probabilities of the different resources. Even if working with a simple microprocessor, Fig. 9 clearly demonstrates the potential of the proposed tool. In this specific case study, introducing a fault tolerant ALU has a minimal impact on the overall reliability of the system (8088 vs. ALU-FT), while protecting the registers provides a major improvement. Although this is somehow expected, it is interesting to note that protecting the whole processor provides a minimal improvement in the overall reliability with respect to protecting only the registers. It is worth remembering here that, according to (16), even if all resources of the processor are fault tolerant the estimated probability decreases with T_M every time the CEP of the generated traces is not equal to one, confirming the estimation of Fig. 9.

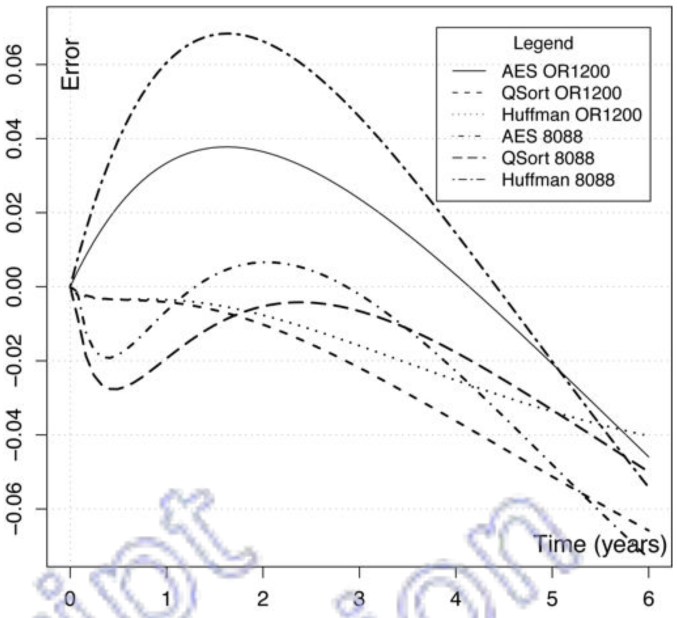


Figure 8. Plot of the error between the fault injection based reliability function and the estimated reliability function for the considered benchmarks and microprocessors

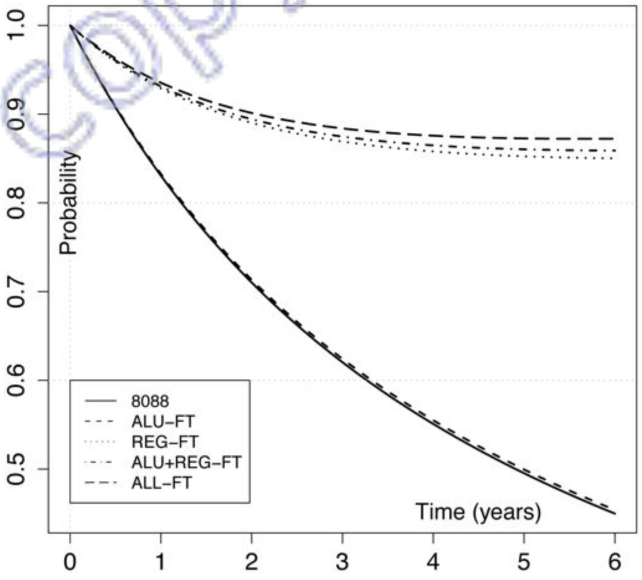


Figure 9. Early design exploration for the implementation of fault-tolerant resources. The graphs shows the estimated reliability function of the 8088 running HUFFMAN with four different fault tolerant configurations of the microprocessor

6 Future improvements

The reliability analysis proposed in Sections 4.1 and 4.2 is clearly limited by the complexity of the analyzed software, and in particular by the complexity of the corresponding CFG. This section discusses how this complexity could be managed by exploiting the intrinsic hierarchy of a program. Considering the simple CFG including a call to a function F reported in Fig. 10-a, the CFG can be clearly partitioned into two portions: (i) the main program, and (ii) the function F (gray part of Fig. 10-a).

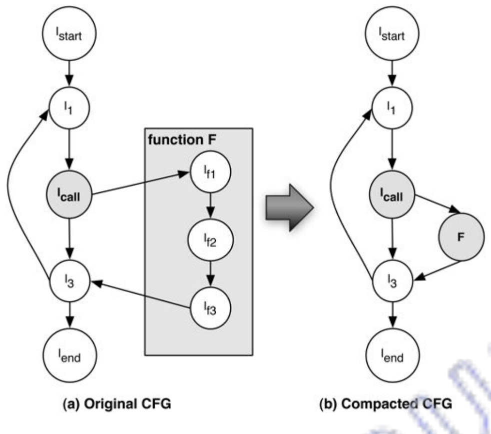


Figure 10. Control Flow Graph Reduction

The full portion of the graph modelling the function F can be collapsed into a single node (Node F of Fig. 10-b), defining a new high level instruction. The new instruction will be characterized by a set of input operands including all function parameters, a set of output values corresponding to the return values of the function, and finally, a single operator corresponding to the actual computation performed by the function with a probability of success computed using equation (15) by considering the CFG of the function in isolation. The state of this instruction will contain both local and global variables of the program, but in general, the active state of the function will include the function return values only. This collapsing technique has the potential to reduce the complexity of the CFG by analyzing portions of it in isolation. This will allow the reduction of the amount and complexity of the traces to analyze, thus allowing the management of very complex applications.

7 Conclusion

This paper proposed a new reliability evaluation methodology targeting microprocessors running a software application. Compared to fault injection, the proposed approach makes it possible to save a considerable amount of time: fault injection is used only once for a one-time, reusable, characterization of the microprocessor in terms of probability of success of each of its instructions in the presence of a soft-error in the hardware. The overall reliability of the microporocessor running a given workload is then computed with a purely probabilistic approach. The same characterization can then be reused every time the same CPU is used to build a new system or a new application software needs to be evaluated. The proposed method makes it possible to perform early exploration of design alternatives giving the possibility of comparing the system reliability using different processor architectures, even before the actual system's design is available. In the long run, the diffusion of this approach could lead to the availability of libraries of microprocessor characterizations (freely available or proprietary) that would allow users to evaluate the reliability of microprocessor-based systems without the need of neither a single fault-injection campaign, nor a deep knowledge of the microprocessor architecture (usually proprietary).

There is still room for several improvements. Simulation and computational constraints do not allow to manage more than one program at a time, or to consider the introduction of operating system code. In order to manage very complex applications, further optimizations such as the one proposed in Section 6 must be implemented to brakedown the complexity into more managable subproblems. Since the execution time is not part of the model, in its current form the proposed approach does not allow the targeting of real-time constraints. In order to obtain an even more precise reliability estimation, the proposed heuristic for the computation of the dependency among resources, which represents one of the most critical elements of the model, can be further refined.

Experimental results performed on the Intel 8088™ and the OpenRISC1200 microprocessors are very promising. All the presented experiments show very small differences in the reliability estimation between this approach and a traditional fault injection experiment, but with a huge saving in computation time. The complexity of the microprocessors used for the experiments is not very high, but they nevertheless include several of the most critical functionalities of state-of-the-art devices (e.g. pipelines, floating-point units, etc...). The results suggest that there is no reason to believe that the proposed methodology would not be applicable to more complex microprocessors, provided that the resources to characterize them are available. It should also be considered that if the complexity of a modern microprocessor does not allow its characterization as proposed in this paper, it would neither allow a reliable fault injection campaign.

References

[1] S. Kumar and A. Aggarwal, "Self-checking instructions: reducing instruction redundancy for concurrent error detection," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, 2006, pp. 64–73.

[2] A. Shye, J. Blomstedt, T. Moseley, V. Reddi, and D. Connors, "Plr: A software approach to transient fault tolerance for multicore architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 135–148, April-June 2009.

[3] R. Baumann, “Technology scaling trends and accelerated testing for soft errors in commercial silicon devices,” in *Proceedings of the IEEE International On-Line Testing Symposium*, 2003, p. 4.

[4] B. R., “Soft errors in advanced computer systems,” *IEEE Design and Test of Computers*, vol. 22, no. 3, pp. 258–266, 2005.

[5] S. Borkar, “Tackling variability and reliability challenges,” *IEEE Design and Test of Computers*, vol. 23, no. 6, p. 520, 2006.

[6] ——, “Thousand core chips: a technology perspective,” in *Proceedings of the 44th annual Design Automation Conference*, 2007, pp. 746–749.

[7] P. Dodd, “Physics-based simulation of single-event effects,” *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 343–357, Sept. 2005.

[8] S. Mitra, M. Zhang, T. Mak, N. Seifert, V. Zia, and K. S. Kim, “Logic soft errors: a major barrier to robust platform design,” in *IEEE International Test Conference, 2005. Proceedings*, Nov. 2005, pp. 10 pp.–696.

[9] E. Normand, “Single event upset at ground level,” *IEEE Transactions on Nuclear Science*, vol. 43, no. 6, pp. 2742–2750, Dec 1996.

[10] R. Baumann, “Soft errors in commercial semiconductor technology: Overview and scaling trends,” in *IEEE Reliability Physics Tutorial Notes, Reliability Fundamentals*, Apr. 2002, pp. 121_01.1–121_01.14.

[11] S. Krishnamohan and N. R. Mahapatra, “Analysis and design of soft-error hardened latches,” in *Proceedings of the 15th ACM Great Lakes symposium on VLSI*, 2005, pp. 328–331.

[12] M. Hosseinabady, P. Lotfi-Kamran, G. Di Natale, S. Di Carlo, A. Benso, and P. Prinetto, “Single-event upset analysis and protection in high speed circuits,” in *Eleventh IEEE European Test Symposium, 2006. ETS ’06.* IEEE, 21-24 May 2006, pp. 29–34.

[13] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama, “A 1.3ghz fifth generation sparc64 microprocessor,” in *Proceedings of the 40th annual Design Automation Conference*, June 2003, pp. 702–705.

[14] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, “A watchdog processor to detect data and control flow errors,” in *9th IEEE On-Line Testing Symposium, 2003. IOLTS 2003.* IEEE, 7-9 July 2003, pp. 144–148.

[15] S. D. Carlo, G. D. Natale, and R. Mariani, “On-line instruction-checking in pipelined microprocessors,” in *Asian Test Symposium, 2008. ATS ’08. 17th*, Nov. 2008, pp. 377–382.

[16] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and L. Tagliaferri, “Control-flow checking via regular expressions,” in *10th Asian Test Symposium, 2001. Proceedings.* IEEE, 12-21 Nov. 2001, pp. 299–303.

[17] 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.

[18] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, “Seu effect analysis in a open-source router via a distributed fault injection environment,” in *Proceedings Design, Automation and Test in Europe, 2001. Conference and Exhibition 2001.* IEEE, 13-16 March 2001, pp. 219–223.

[19] A. Benso, S. Di Carlo, G. Di Natale, L. Tagliaferri, and P. Prinetto, “Validation of a software dependability tool via fault injection experiments,” in *Proceedings Seventh International On-Line Testing Workshop, 2001.* IEEE, 9-11 July 2001, pp. 3–8.

[20] A. Benso, S. Di Carlo, G. Di Natale, P. Prinetto, and L. Tagliaferri, “Software dependability techniques validated via fault injection experiments,” in *6th European Conference on Radiation and Its Effects on Components and Systems, 2001.* IEEE, 10-14 Sept. 2001, pp. 269–274.

[21] M. Omana, G. Papasso, D. Rossi, and C. Metra, “A model for transient fault propagation in combinatorial logic,” in *Proceedings of the 9th IEEE On-Line Testing Symposium*, 2003, pp. 111–115.

[22] A. Maheshwari, I. Koren, and W. Burleson, “Techniques for transient fault sensitivity analysis and reduction in vlsi circuits,” *IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*, p. 597, 2003.

[23] H. Nguyen and Y. Yagil, “A systematic approach to ser estimation and solutions,” in *IEEE International Reliability Physics Symposium Proceedings*, March-April 2003, pp. 60–70.

[24] K. Mohanram and N. Touba, “Partial error masking to reduce soft error failure rate in logic circuits,” in *IEEE International Symposium on Defect and Fault-Tolerance in VLSI Systems*, 2003, p. 433.

[25] ——, “Cost-effective approach for reducing soft error failure rate in logic circuits,” in *Proceedings of the IEEE International Test Conference*, vol. 1, 2003, pp. 893–901.

[26] M. Sonza Reorda and M. Violante, “Accurate and efficient analysis of single event transients in vlsi circuits,” in *Proceedings of the IEEE International On-Line Testing Symposium*, 2003, p. 101.

[27] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi, “Modeling the effect of technology trends on the soft error rate of combinational logic,” in *Proceedings of the International Conference on Dependable Systems and Networks*, 2002, pp. 389–398.

[28] S. Kim and A. K. Somani, “Soft error sensitivity characterization for microprocessor dependability enhancement strategy,” in *Proceedings of the International Conference on Dependable Systems and Networks*, 2002, pp. 416–428.

[29] 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*, 2003, p. 29.

[30] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin, “Measuring architectural vulnerability factors,” *IEEE Micro*, vol. 23, no. 6, pp. 70–75, Nov.-Dec. 2003.

[31] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. patel, “Characterizing the effects of transient faults on a high-performance processor pipeline,” in *Proceedings of the International Conference on Dependable Systems and Networks*, 2004, p. 61.

[32] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, “Techniques to reduce the soft error rate of a high-performance microprocessor,” in *Proceedings of the 31st annual international symposium on Computer architecture*, 2004, p. 264.

[33] X. Li, S. V. Adve, P. Bose, and J. A. Rivers, “Online estimation of architectural vulnerability factor for soft errors,” in *Proceedings of the 35th International Symposium on Computer Architecture*, 2008, pp. 341–352.

[34] X. Li, S. Adve, P. Bose, and J. Rivers, “Softarch: an architecture-level tool for modeling and analyzing soft errors,” in *Proceedings. International Conference on Dependable Systems and Networks*, 2005, pp. 496–505.

[35] V. Sridharan and D. R. Kaeli, “Using pvf traces to accelerate avf modeling,” in *Proceedings of the IEEE Workshop on Silicon Errors in Logic - System Effects*, Stanford, California, March 23-24 2010. [Online]. Available: http://web.me.com/vilas.sridharan/Vilas_Sridharan/Publications_files/3_Sridharan_P.pdf

[36] T. M. Jones and M. F.P., “Evaluating the effects of compiler optimization on avf,” in *Workshop on the Interaction between Compilers and Computer Architecture (INTERACT)*, 2008.

[37] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, “Static analysis of seu effects on software applications,” in *Proceedings of the International Test Conference*, 2002, pp. 500–508.

[38] T. Karnik and P. Hazucha, “Characterization of soft errors caused by single event upsets in cmos processes,” *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 2, pp. 128–143, April-June 2004.

[39] S. Mitra, T. Karnik, N. Seifert, and M. Zhang, “Logic soft errors in sub-65nm technologies design and cad challenges,” in *Proceedings. 42nd Design Automation Conference*, June 2005, pp. 2–4.

[40] NIST/SEMATECH. e-handbook of statistical methods. [Online]. Available: http://www.itl.nist.gov/div898/handbook/

[41] J. Larus, “Efficient program tracing,” *Computer*, vol. 26, no. 5, pp. 52 –61, may 1993.

[42] J. Maebe and B. De Sutter. Diablo. [Online]. Available: http://diablo.elis.ugent.be/

[43] HT-LAB. Cpu86 cpu86 8088 fpga ip core. [Online]. Available: http://www.ht-lab.com/freecores/cpu8086/cpu86.html

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

[45] A. Bosio and G. Di Natale, “Lifting: A flexible open-source fault simulator,” in *Proceedings of the 17th IEEE Asian Test Symposium*, 2008, pp. 35–40.

Alessandro Savino received the MS degree in computer engineering and the PhD degree in information technologies from the Politecnico di Torino, Italy, where he has been a postdoc in the Department of Control and Computer Engineering since 2009. His main research topics are microprocessor test and software-based self-test.

Stefano Di Carlo received the MS degree in computer engineering and the PhD degree in information technologies from the Politecnico di Torino, Italy, where he has been an assistant professor in the Department of Control and Computer Engineering since 2008. His research interests include DFT, BIST, and dependability. He is a golden core member of the IEEE Computer Society and a member of the IEEE.

Alfredo Benso received the MS degree in computer engineering and the PhD degree in information technologies, both from Politecnico di Torino, Italy, where he is working as a tenured associate professor of computer engineering. His research interests include DFT, BIST, and dependability. He is also actively involved in the Computer Society, where he has been a leading volunteer for several projects. He is a Computer Society Golden Core Member, and a senior member of the IEEE.

Gianfranco Politano received the MS degree in computer engineering and the PhD degree in information technologies from the Politecnico di Torino, Italy, where he has been a postdoc in the Department of Control and Computer Engineering since 2011. His main research topics are system reliability and machine learning techniques. He is a student member of the IEEE and the IEEE Computer Society.

Giorgio Di Natale received the PhD in Computer Engineering from Politecnico di Torino in Italy in 2003. Currently he is a researcher for the National Research Center of France at the LIRMM laboratory in Montpellier. He has published articles in publications spanning diverse disciplines, including memory testing, fault tolerance, and secure chips design and test. He is a Golden Core member of the IEEE Computer Society and he serves the European Test Technology Technical Council (eTTTC) of IEEE Computer Society as vice-chair.

Alberto Bosio received the MS degree in computer engineering and the PhD degree in information technologies from the Politecnico di Torino, Italy. He is currently an associate professor in the Laboratoire d'Informatique, de Robotique et de Microelectronique de Montpellier, University of Montpellier II/CNRS, Montpellier, France. His main research activity are methodologies and tools to improve the development of highly dependable systems, at different levels: for basic digital components, for systems on chip, up to microprocessor-based systems. He is a member of the IEEE.

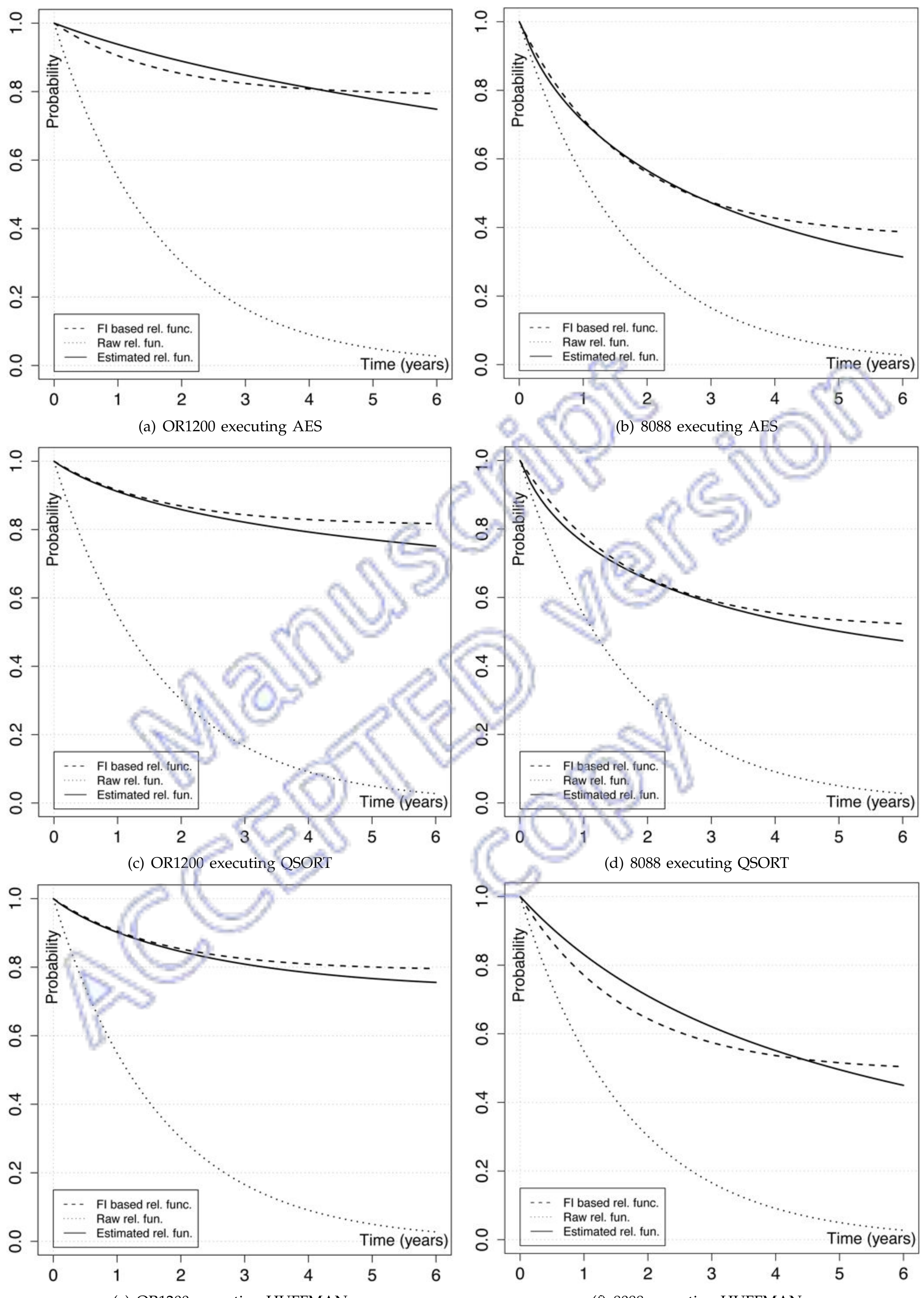


Figure 7. Result of the reliability analysis for the two microprocessors running the three considered benchmarks. Each plot shows the raw reliability function of the microprocessor (raw rel. fun.), the reliability function estimated through fault injection (FI based rel. fun.) and the reliability function estimated with the proposed model (Estimated rel. fun.).