A fault-injection methodology for the system-level dependability analysis of multiprocessor embedded systems

Antonio Miele

Politecnico di Milano, Dipartimento di Elettronica, Informazione e Bioingegneria, Piazza Leonardo 32, 20133 Milano, Italy

Article history:
Received 4 November 2013
Revised 27 May 2014
Accepted 28 May 2014
Available online 14 June 2014

1. Introduction

Dependability is increasingly becoming a key aspect in the design of embedded systems, given their widespread adoption in mission- and safety–critical applications such as industrial automation, transportation, network infrastructures and military facilities. This need has been even more exacerbated as the aggressive technological advances in transistor size scaling, increased frequencies and power densities have negatively affected the reliability of the components constituting such systems, also increasing their susceptibility to transient faults, also known as *soft errors* [1].

In this context, it is mandatory to perform an assessment of the dependability of the system under design in order to identify its criticalities and, possibly, the vulnerabilities of the adopted fault-tolerant mechanisms. Moreover, when considering the complexity of modern embedded systems (in terms of both the architectural platform and the number of executed applications and functionalities) and the stringent time-to-market, it becomes paramount to perform this kind of analysis from the early phases of the design flow – a common practice for the other design parameters, such as performance or power consumption – to provide a prompt feedback for the system refinement and to reduce the risk of late discovery of safety-related design pitfalls.

Fault injection plays a relevant role in the fulfillment of dependability-related analysis [2]; it allows to evaluate how the system reacts to a controlled corruption of its internal state. However, it is possible to state that classical fault injection approaches do not suffice, especially in the early phases of the design flow, when the designer is evaluating how to harden the system. In fact, most approaches (e.g., [3–5]) are usually based on monitoring the final outputs and comparing them against a golden counterpart for a specified time window, reasonable to obtain the systems' results. This strategy suffices only in reporting statistics on the final effects of the injected faults (e.g., how many times a system's failure occurred, or whether the hardening mechanisms were able to mitigate the fault effects on the system's outputs), but it is not able to state the effects of the fault in the system and, in particular, on the various functionalities within the application. This aspect is especially relevant when considering that the executed applications are composed of a large set of interconnected functionalities. In conclusion, this strategy does not provide any relevant information to the designer who aims at performing a local selective hardening as a cost-effective countermeasure to the identified problematic situations.

Aiming at investigating the presence of errors inside the architecture, other approaches (e.g., [6–8]) monitor not only the output of the system but also its internal memory elements; in this way it is possible to classify as *latent* the scenario in which, at the end of the experiment, the output of the system is correct, but the

E-mail address: antonio.miele@polimi.it

internal state is corrupted, thus potentially producing an error in the future. Unfortunately, this approach is still limited to the identification of a corrupted state in system architectures, but it hardly provides additional information on the actual status of the application executed on the embedded system. This represents again a limitation when considering the complexity of both the embedded system's architectures and the executed applications; indeed, it becomes mandatory to analyze how the executed application failed because of the occurrence of a fault, and how the errors were activated and propagated in the various parts of the application. Nevertheless, the trace analysis may be also misleading by identifying "false positive" situations: sometimes, faults may generate errors in the control flow (e.g., a jump of few instructions backwards causing their re-execution) whose final effect is a neglectable timing error while the result is still correct. In fact, in these situations, the considered approach would identify a considerable difference between the status of the corrupted system and the golden one. Moving one step further, when considering multiprocessor architectures the situation is even more critical, as the effects of one failure may corrupt more cores, or the expected benefits of hardening mechanisms using redundant computations may not hold due to unexpected effects/interactions.

Given theses motivations, the goal of this work is to propose a new methodology based on fault injection for the *system-level dependability assessment* of heterogeneous multiprocessor systems in the early stages of the design flow. The methodology aims at providing the means to investigate the effects of the faults not only on the produced outputs but also in the internals of the system during the overall execution and from a *system-level perspective*, i.e., combining the architectural abstraction level and the application one. The purpose is to provide the support for an accurately study of the system's behavior after the faults happened, to provide a detailed report on which are the critical portions of the architecture and (especially) of the application, also with respect to the system's properties and functionalities and to the fault-tolerant strategies employed for hardening the system. Therefore, the methodology provides:

- the support for an accurate planning of the fault campaign on the basis of not only the architectural information but also the application one, by integrating also some state-of-the-art approaches as a support (e.g., for the fault modeling or the pruning of the fault list);
- the capability to perform error monitoring also at application level with the granularity of function call/return events, and considering custom functionality-aware conditions for an accurate analysis of the relationship between the architectural faults and the error in the executed applications;
- a joint error propagation and classification approach, in particular offering the capability to perform a custom ad hoc classification of the effects of the fault, not only on the final output at the end of the simulation, but internally to trace the error evolution both in the application (in particular in its functions) and in the architecture.

Aiming at working in the early stages of the design flow, the framework supporting the methodology has been implemented within a state-of-the-art simulation framework [9] for architectural specifications modeled with SystemC and Transaction Level Modeling (TLM, [10]), and that has a fault injection infrastructure [11]. The methodology will mainly focus on transient faults, due to the peculiarities of the adopted fault injector; however the analysis mechanism can be easily extended to support permanent faults. The effectiveness of the methodology will be demonstrated in two different case studies considering a hardened image processing system and a control system from the automotive scenario. It will be shown how the proposed approach is able to produce an accurate dependability report considering aspects of both the architecture and the application of the system under consideration.

The rest of the paper is organized as follows. The following section presents a review of the literature on the use of simulated fault injection for dependability analysis, highlighting the limitations of the previous approaches in the execution of accurate dependability analysis in complex scenarios. Section 3 describes the proposed system-level dependability analysis methodology for SystemC/TLM specifications of heterogeneous multiprocessor architectures; first an overview of the analysis flow will be given and then the various steps will be described in details. A description of the methodology at work in two case studies is presented in Section 4, while the last section draws the conclusions on the presented approach.

2. Related work

A large number of fault injectors for digital systems has been proposed in the past years, as shown in the survey presented in [2]. Based on the goal we pursue, the most relevant aspect for their classification is the abstraction level they work at: physical fault injection (e.g., [12,13]), software implemented fault injection (SWIFI, e.g., [14,15]), fault emulation (e.g., [5,8,6,4,16]), or fault simulation (e.g., [11,3,7,17–21]). Since they act at different abstraction levels, the various approaches are complementary, and, therefore, they support the reliability analysis in the various phases of the system design flow, from the preliminary evaluation towards the final system validation. The methodology proposed in this paper is integrated in the early phases of the design flow and requires an high controllability and observability of the fault; thus, we will consider a fault simulation framework.

The strategy for the analysis usually adopted by these injection frameworks is based on the monitoring of the effects the injected faults have on the system's final results. In case of microprocessor systems, these frameworks classify software result as silent or failure [3,4], and at most they will state whether an exception has been thrown or a timeout expired [15,6]. Some approaches perform an evaluation of the sensitivity of the various architectural components of the microprocessor [5,3]. Others perform an analysis of the error propagation by comparing the traces of registers' content with the golden counterparts [11,8,22], thus providing a straightforward architectural-level evaluation. Such comparison becomes unfeasible when considering complex systems due to the large amount of values to be managed; nevertheless, in most situations, it is difficult to give an application-level semantic to such architectural raw data. In conclusion, the adopted strategies are usually very simple and consider a classification based on a reduced and generic set of classes of effects; moreover, when analyzing the internal state, these strategies work only at an architectural abstraction level.

Other approaches for a more accurate analysis of the fault effects have been proposed in the past years. One interesting approach consists in the employment of Failure Models and Effects Analysis (FMEA) on the embedded system; however, it has been only adopted at the digital logic level [23] or the architectural level [24] thus not considering the application executed on the system under test. In [25] the authors present an analysis of the propagation of the fault effects to critical registers in a specific microcontroller. In [26], error propagation and fault classification are combined; in particular, the status of the system is analyzed during the overall simulation according to a set of user-custom classes of effects exploiting the error propagation. Again, the approach acts at architectural level. Other kinds of analysis strategy that focus on a higher level of abstraction are the Architecture Vulnerability Factor

(AVF, [27]), which aims at estimating the probability that a fault affecting the various components of a processor leads to an error on the primary output of the system, and the derived Program Vulnerability Factor (PVF, [28]) and Thread Vulnerability Factor (TVF, [29]). Unfortunately, these approaches work at assembly instruction level, thus too low to analyze a complex system/application; nevertheless, another common limitation is that they never consider complex heterogeneous architectures composed by several processors and with hardware accelerators.

When considering the aim of anticipating the dependability analysis at the early stages of the design flow, the usage of SystemC and TLM [10] is particularly appealing due to their support for the modeling of the system at different levels of abstraction, and in particular at the functional one. A large set of fault injection frameworks for SystemC/TLM specifications has been proposed in the literature (e.g., [9,17–19,30]). However, most of these works mainly investigate fault injection strategies specific for the SystemC models [11,18,30] or define functional fault models for high-level specifications modeled with the TLM approach [31,17], neglecting the issues of the dependability analysis. Very few works proposed approaches for the system-level dependability evaluation in complex systems specified with SystemC/TLM [19,32,33,30]. For instance, the approach in [19] supports FMEA by specifying custom monitoring conditions on the primary output of the systems, thus taking into account the system functionality. However, among them, only in [32] the authors highlight the necessity to define internal watch points for the application-level monitoring of the occurred errors and their propagation in order to identify criticalities of specific parts of the system; no systematic approach is proposed in the paper for achieving this goal. Furthermore, since these approaches work at system level, they adopt virtual platforms consisting of behavioral models; for our purpose, this represents a limitation since at this high abstraction level there is no distinction between the architecture and the executed application. Therefore, it is not possible to capture many relevant aspects for the dependability analysis. On the other hand, other approaches have considered more detailed architectural models based on instruction set simulators or cycle-accurate processor models in SystemC [3,9,17]. However, similarly to the approaches discussed above, they support only the analysis of the final results of the executed application.

In conclusion, there is a lack of a methodology supporting a system-level dependability analysis able to focus at the same time on the architecture and on the application of the system under design, and supporting an accurate analysis of the effects of the faults in the various parts of the system. In [34] we defined a preliminary simulation framework for performing an application-level dependability analysis in SystemC/TLM multiprocessor models; however, it did not propose a complete methodological approach. Here, we adopt that framework and we define the overall methodology achieving the pursued goal; moreover, we propose two case studies accurately highlighting the effectiveness of the methodology in the identification of the dependability issues of the system under design.

3. The dependability-analysis methodology

The proposed methodology for the system-level dependability analysis is depicted in Fig. 1. It takes as input a specification of the embedded system in terms of the architectural platform and the executed application, and produces a dependability report showing the vulnerabilities and the critical aspects of that system.

The embedded system under analysis is composed of a multiprocessor architecture running a parallel application. The architecture is composed by a set of heterogeneous processing units,

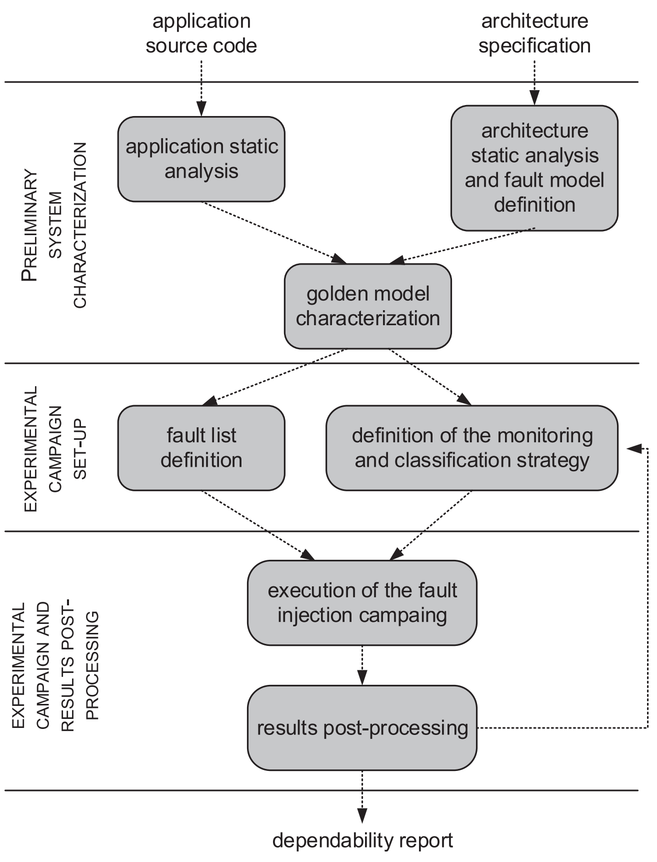
application source code
architecture specification
PRELIMINARY SYSTEM CHARACTERIZATION
application static analysis
architecture static analysis and fault model definition
golden model characterization
EXPERIMENTAL CAMPAIGN SET-UP
fault list definition
definition of the monitoring and classification strategy
EXPERIMENTAL CAMPAIGN AND RESULTS POST-PROCESSING
execution of the fault injection campaing
results post-processing
dependability report

Fig. 1. Workflow of the proposed methodology.

general purpose processors or hardware accelerators; the architecture is described in SystemC/TLM to provide a model to be simulated that can be used for the fault injection campaigns. Coherently with the other design and analysis activities performed during the early stages of the design flow, the application is modeled as a set of tasks, each one representing a specific functionality. The source code is implemented in C: tasks are implemented in terms of functions and they are parallelized by means of the thread paradigm. Moreover, each task is mapped on a specific processing unit, in charge of its execution. Fig. 2 presents an example of a

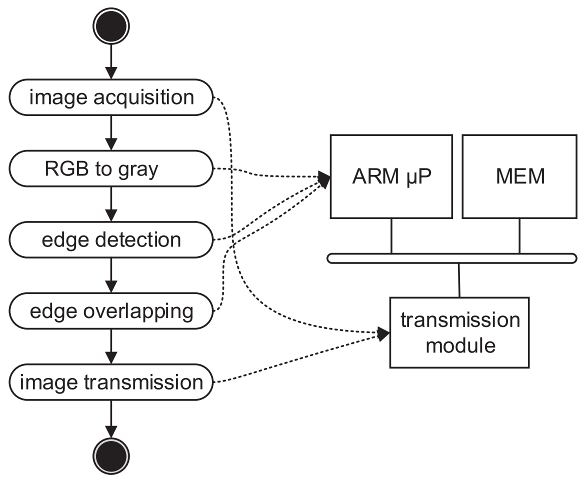

Fig. 2. An example of image processing application mapped on a processor-based architecture.

simple architecture running an image processing application, organized as a chain of tasks. Ffor sake of simplicity the `main` function orchestrating the execution of the various functions is omitted; for the same reason, a simple-thread application has been considered even if during the presentation it will be shown how the methodology works in multi-threaded scenarios.

The output of the methodology is an accurate report on the dependability characteristics of the system under analysis. It will contain statistical information on the response of the system to the occurrence of faults taking also into account the possibly-available fault management mechanisms, and a detailed report on the error propagation, both at application and architecture levels for a selected subset of fault scenarios, which will accurately illustrate the fault/error relationship. The granularity level of the application-level monitoring and classification activities supported by the methodology are intended for single task execution, in particular capturing the function call/return events. Thus, this report will represent a characterization of the sensible zones in the architecture and the critical functionalities in the application, information that can be used by the designer as a feedback for refining the applied hardening solution, or for guiding a selective hardening of the system.

The methodology workflow is organized in several steps divided into three main phases:

1. preliminary characterization of the system,
2. set-up of the experimental campaign, and
3. execution experimental campaign and results' post-processing.

Preliminary characterization of the system. This phase is devoted to an analysis of the internal structure of the model of the considered system and of its nominal behavior, i.e., when not affected by any fault. First, the architecture specification is analyzed and suitable fault models are defined; due to the adoption of SystemC/TLM architecture specifications, functional fault models corrupting the internal status of the architecture's components are defined. This activity is performed also by leveraging possibly available lower level information on the hardware implementation of the architecture's components. The second step is devoted to the extraction of specific information from the application source code and the executable; this characterization is used in the subsequent phases for interpreting architecture raw data at an application-execution level, similarly to a software debugger. The last step of the preliminary phase is the golden model characterization of the system execution at both architectural and application level: the result is a graph representing the sequence of executed tasks annotated with actual input/output values, timing and mapping information, and the traces of the relevant observation points in the architecture.

Set-up of the experimental campaign. The fault list is defined according the system characterization performed in the previous phase and a liveness analysis of the memory locations; in this way, it is possible to accurately stimulate the system under analysis to obtain a precise dependability report. Moreover, the monitoring and classification strategies to be used during the experiments are defined by the designer, with the aim of accurately studying the dependability properties exposed by the system under design. The methodology offers the possibility to define ad hoc conditions to monitor the system behavior at both application and architecture level, also with respect to the properties and the functionalities the system exposes. Such conditions allow to analyze at system level the evolution of the error in both the system's internal points and outputs.

Experimental campaign and results post-processing. Planned experiments are executed and results are collected and analyzed to generate a dependability report. Experiments will produce statistics on the response of the system to the occurrence of faults from both the architectural and application points of view. Moreover, the most problematic situation can be further investigated to determine step-by-step the actual evolution and propagation of the errors in the system. The experimental sessions may be executed several times according to the analysis of the results' post-processing, to better investigate specific situations; to this purpose, it is also possible to go back to the set-up phase to refine the monitoring and classification strategy or to generate a more specific fault list.

The various steps composing the three phases of the methodology are discussed in details in the following sections. As a conclusive discussion necessary for presenting the details of the various steps we here introduce the fault simulation environment that supports the proposed methodology.

The framework automating the methodology has been implemented in ReSP[1] [9], a simulation platform for multiprocessor systems with hardware accelerators running parallel multi-threaded applications. ReSP is a flexible simulation framework for SystemC/TLM models, implemented in Python and C++, providing many facilities for an agile and non-intrusive analysis of the systems during the simulation (e.g., performance evaluations, processor's software debugging and power estimation and fault injection). Among them, two are particularly relevant for the current work: the fault injection infrastructure and a set of monitoring interfaces. ReSP implements the fault injection infrastructure described in [11], particularly targeted for transient faults. It supports the injection of faults in the internal status of the architecture's components (represented by the attributes of the SystemC modules) by means of *simulation commands*, and in the interconnections between TLM ports by means of *saboteurs* automatically instantiated and interconnected. Moreover, to support the various analysis, the following specific monitoring interfaces are provided:

- processor modules offer a sort of debugging interface for continuously monitoring issued instructions and for accessing the registers and the memory locations,
- probes can be instantiated on the TLM interconnections similarly to the saboteurs, thus implementing traffic monitors, and
- architecture-level monitors can be implemented by means of *SystemC simulation delta-cycle callbacks*, that are user-defined functions executed by ReSP simulation kernel at the end of each simulation delta-cycle.

Fig. 3 summarizes the mentioned injection and monitoring facilities.

3.1. Static characterization of the application

The static characterization of the application is a necessary step to enable the debug-like application-level interpretation of the architectural raw data. In particular, during this step, the source code of the application is analyzed to retrieve all the information necessary to know how each function will be invoked and executed during the system's execution. For instance, it is of interest to know the required parameters, their types, and in which register of memory location they are stored. The extracted information will be organized in a data structure, called *functions table*, that is returned as output by the current step. Such table will allow during the simulation to interpret the content of the register bank and the issued instruction of each processor to detect when a function is invoked or terminates and which are the transmitted actual parameters.

[1] The re-engineered version of the tool can be downloaded from [35].

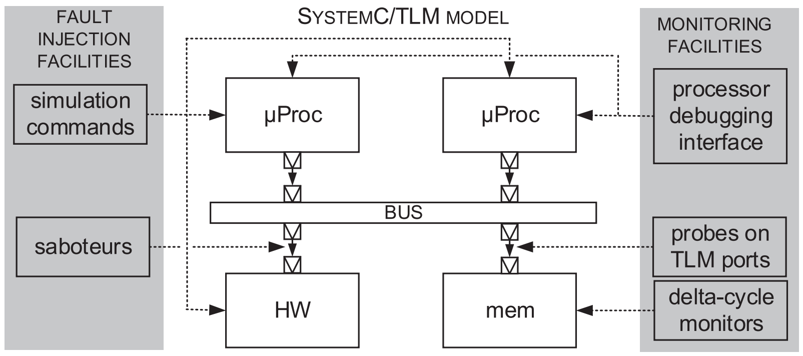

Fig. 3. Injection and monitoring facilities available in ReSP.

Table 1
Functions table for the image processing application in Fig. 2.

Function	Memory addresses	Parameters			
		Name	Position	Type	Location
main	0x1498 – 0x1640				
rgb2gray	0xea0 – 0xf64	inputImg	0	char* (dim = 17,280)	reg0
		outputImg	1	char* (dim = 5760)	reg1
		width	2	unsigned	reg2
		height	3	unsigned	reg3
edgeDetector	0xf68 – 0x12fc	inputImg	0	char* (dim = 5760)	reg0
		outputImg	1	char* (dim = 5760)	reg1
		width	2	unsigned	reg2
		height	3	unsigned	reg3
edgeOverlapping	0x1300 – 0x1480	inputImg	0	char* (dim = 17,280)	reg0
		edgeImg	1	char* (dim = 5760)	reg1
		outputImg	2	char* (dim = 17,280)	reg2
		width	3	unsigned	reg3
		height	4	unsigned	SP + 0x0
readBitmap	HW	inputImg	0	char* (dim = 17,280)	reg0
		width	1	unsigned	reg1
		height	2	unsigned	reg2
writeBitmap	HW	outputImg	0	char* (dim = 17,280)	reg0
		width	1	unsigned	reg1
		height	2	unsigned	reg2

Table 1 shows the functions table for the example in Fig. 2. The table stores an entry for each function representing an application task. The entry specifies the name of the function and the range of addresses in the instruction memory occupied by the function assembler code. Moreover, the entry contains the list of parameters and the related data, in particular describing: (i) the position in the prototype, (ii) its type, and (iii) the register or memory location that will be used for its transmission. The return value (if any) is represented similarly to the parameters. The parameter type is a particularly relevant item since it represent not only the actual data type of the parameter but also the modality of the parameter transmission, that is by value or by address (i.e., by using a pointer). In fact, the modality specifies if the parameters represent an input or an output for the function, and therefore if a change of its value will affect the function callee. Moreover, since we consider C source code, arrays are passed only by address; for this reason, the designer has to manually specify the size of each array. It is worth noting that the size of the array may change according to the input data specified during the execution; therefore, if necessary, these parameters can be also specified in the subsequent step dealing with the golden model characterization. Finally, we assume that tasks executed by hardware accelerators are invoked in the application source code by means of a function call mechanism (as in [36]); therefore, also these functions will have an entry in the functions table. It is worth noting that this assumption does not represent a limitation; in fact, hardware accelerators are usually configured as slave components piloted by means of a set of data and control registers mapped on the memory address space. Thus, if these operations are not wrapped in a single function, the entry in the functions table will be filled manually by the designer, and will contain the addresses of the registers to be accessed for starting the accelerator.

The activities executed to build the described table are shown in Fig. 4. First, the function prototypes are extracted from the source code and are parsed to retrieve the function names and the list of parameters (position, name and type). If any, the designer will specify the size of the arrays; since such information cannot be automatically computed. Then, each function prototype is analyzed according to the function-calling convention defined in the Application Binary Interface (ABI) of the processor that will execute the application (an example of ABI for the ARM processor is available in [37]). The function-calling convention will determine which register or memory address will contain parameters passed to the function. For instance, when considering the ARM processor, the first four actual parameters will be passed through the register bank and the subsequent will be pushed on the memory stack; for this reason, in Fig. 1, the location of the last parameter of the `edgeOverlapping` function is a memory address specified rela-

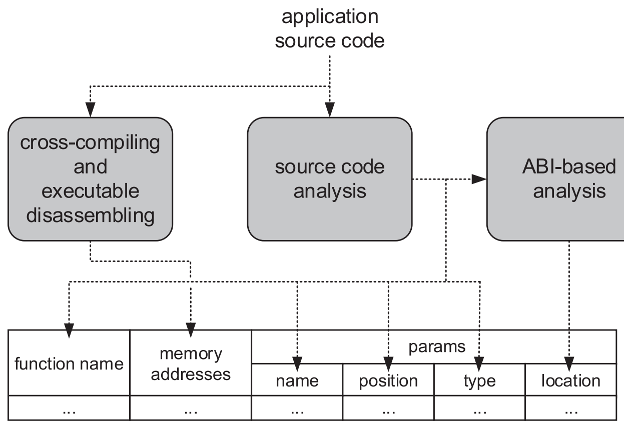

Fig. 4. Generation of the functions table.

tively to the stack pointer value. Moreover, it is important to notice that during the execution, the location specified for the parameters passed by address will contain a pointer; thus, in order to access the actual data, it will be necessary to follow that pointer and to read the referenced memory location. Finally, the executable is disassembled to analyze its organization in the memory; in particular, the boundary addresses of the memory space containing the code of each function are retrieved and added to the functions table.

The described activities have been automated by means of a simple parser tool that has been integrated in ReSP; in particular the executable disassembling has been implemented by means of the *GNU Binary File Descriptor* (BFD) library for the manipulation of executable files.

As a final consideration, the retrieved information supports the analysis of the system execution at the application level with a granularity of single function call; however, it does not allow to further delve into the execution of the single instructions in the source code. From a methodological point of view, this approach is consistent with our interest in performing an analysis at system level; indeed we consider only the executed functionalities by making the reasonable assumption that the tasks executed by the application are opportunely structured in functions in the source code. In order to enable an analysis of the execution of each single instruction, it is necessary to compile the source code with the debugging symbols; however, this represents an intrusive action since a different executable is generated.

3.2. Analysis of the architecture and definition of the fault model

A preliminary analysis on the architecture is performed with the goal of (i) identifying the observation points that can be probed to analyze the status of the system during the simulation, and (ii) defining the fault models used for soliciting the system during the experimental campaigns.

As discussed above, ReSP supports a non-intrusive read and write access (i) to the internal state of the architectural components (registers and memories), and (ii) to the TLM ports. Therefore, these two elements will represent the selectable observation points and injection ones; they are listed to be used in the subsequent steps.

Given ReSP fault injector's capabilities, this work is mainly focused on soft errors, that are usually represented by means of transient corruptions of the memory content. However, since we are not working at Register-Transfer Level (RTL) but at transaction level, only a subset of the memory locations are actually represented in the considered model in terms of a set of internal state's elements, while the other ones are "encapsulated" in the algorithmic descriptions of the various functionalities. In this scenario, it is necessary to define functional fault models for SystemC transaction-level specifications; to this end, we integrated in the proposed methodology the approach presented in [31], here briefly presented. Each component in the architecture is analyzed separately. If a full RTL specification of the component or partial information on its implementation are available, the effects of faults affecting the RTL memory locations are abstracted and represented in terms of corruptions of the internal state of the high level model. Moreover, possible available information of the final implementation of the component at logic level and device level can be considered and abstracted to further characterize the fault model. If no RTL specification is available, functional failures are defined by mutating the model functionalities and by capturing the effects on the component's internal state and output TLM ports. As a final note, the described analysis is performed manually by the designer.

When considering the example in Fig. 2, the processor within the architecture is modeled with an Instruction Set Simulator (ISS); therefore, this functional model will contain only the registers specified in the processor Instruction Set Architecture, or ISA (e.g., the program counter, the register bank, the status register). We can thus directly represent faults in these registers as bit-flips, while we represent faults in the other RTL memory elements (e.g., the pipeline registers) as the effects of the propagation of the errors on the ISA registers (as done also in [38]). Furthermore, we can also inject faults directly in each memory cell, while faults in the bus and in the memory controller can be modeled by corrupting the transactions between connected TLM ports (we neglect the fault modeling in the transmission peripheral since we are not interested in injecting in that component). Finally, according to the facilities provided by ReSP, the set of available observation points includes the ISA registers, the memory cells and the components' TLM ports.

3.3. Golden model characterization

Once the static analysis of the system has been carried out, the subsequent step consists in the characterization of a plain execution of the system to be used as a golden model during the fault injection experiments. During this phase, two different types of data are extracted: (i) the golden architecture-level characterization, and (ii) the golden application-level characterization.

In the literature, the golden model is defined by collecting the final outputs of the system or the traces of the internal memory elements considered as relevant of the comparison in a faulty scenario. We call this activity *golden architecture-level characterization*; in particular, the methodology collects the traces of a subset of the observation points specified by the designer. This characterization is carried out by means of a specific tool within ReSP, that collects all the values assumed by the internal status of the components and the TLM ports during a single simulation; these observation points are accessed by means of the interfaces shown in Fig. 3.

In addition to this classical golden model, we introduce here a *golden application-level characterization* in which the collected architectural raw data are abstracted to the level of abstraction of function execution. The purpose is to build a model called *execution flow graph* (EFG) which described the golden system execution from an application point of view. The EFG is a direct acyclic graph which describes the flow of execution of the various functions within the application and specifies for each of them the timing information and the actual input/output values. The nodes of the graph represents three different events:

- **function enter** which occurs upon a function call;
- **function exit** which occurs when returning from the function;
- **function re-enter** occurring immediately after a function exit.

Similarly to the behavior of a software debugger, such events are identified during the execution by continuously monitoring the assembler instruction issued by each processor, and comparing the instruction address in the memory against the memory bounds listed in the functions table. In this way, it is possible to determine when a function is entered or exited. Moreover, if the invocation to a hardware accelerator is not modeled as discussed above with a function call, the framework will need to monitor the accelerator to detect when a start command is issued in the control register.

When an event occurs, the actual parameters can be retrieved by accessing the locations specified in the functions table. It is worth noting that for each parameter passed by reference, the identified position will contain a pointer, i.e., the address, to the actual data; therefore to access the data it is necessary to dereference the pointer. On function return, all the parameters passed by address are retrieved again since they represent the outputs of the function. Each event is thus annotated with the retrieved actual parameters, together with the time instant the event occurred, and the identifier of the processing unit (it may be a processor or a hardware accelerator) on which the event occurred. This monitoring and interpretation activity has been automated within ReSP by means of a specific tool, called *application-level interpreter*, connected to the debugging interface of the processors during the golden simulation.

In the EFG, nodes are organized in chains representing the sequence of function calls in the application execution; moreover, since the consider parallel applications, a node may fork in two different successors on the event of thread creation and may join two different predecessors in case of thread join. As an example, Fig. 5 presents a fragment of the EFG of the considered edge detector application.

Do note in case of complex applications, the EFG may became of that considerable size. For this reason, given the hierarchical organization of the function calls, we filter the EFG to consider only the "main" application tasks and ignore other lower-level functions

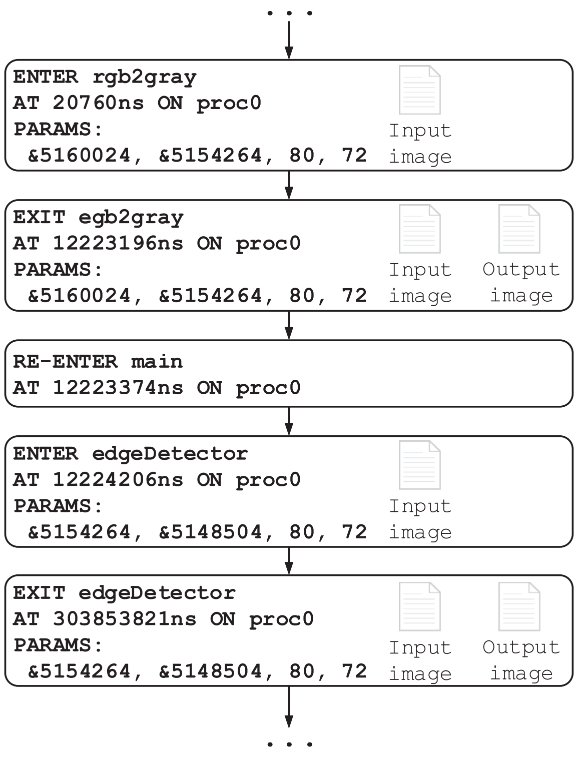

Fig. 5. A fragment of EFG for the image processing application in Fig. 2.

and system calls. For instance, in the example depicted in Fig. 2, the `edgeDetection` function may call the `abs` library function many times for each pixel in the processed image; since we are not interested in it, we can remove the related events from the graph.

Finally, the golden model may depend on the working scenario, and in particular, on the set of inputs used to stimulate the system. In fact, the output of each function, annotated on the EFG, and the dimensions of the various arrays specified in the functions table may change in relation to the provided inputs. Moreover, when the application implements an algorithm whose execution flow is data dependent, the structure of the EFG will change too. For this reason, the methodology requires to perform a golden model characterization for each workload that the designer aims at considering during the subsequent definition of the fault list and in the injection campaigns.

3.4. Definition of the fault list for the experimental campaign

The definition of the fault list to be used in the experimental campaign is a relevant activity that, as highlighted in the literature [39,40], highly affects the quality of the obtained results. In fact, the exhaustive stimulation with all possible faults (i.e., each memory bit corruption in each instant of time) is unmanageable due to the huge number of faults. At the same time, the classically-adopted blind random generation of the fault list may contain a known number of insignificant faults, such as the corruption of a "dead" register, i.e., which contains a value that will not be used in the future, and redundant ones, such as the repeated identical corruption of the same memory location containing the same value in subsequent instants of time. Since the issues related to this activity have been widely investigated in the literature, this step has been defined also by integrating some existing approaches.

First, as in [40], the proposed methodology adopts a pruning strategy based on the analysis of the register liveness for generating a more significant fault list. The aim of this analysis is to identify for each memory element and register within the system all the time windows in which the considered location contains a valid data, i.e., a value that is later used in the execution. The liveness time windows are computed by analyzing the log of the read and write access of each memory element generated during the preliminary golden simulation. As shown in Fig. 6, each time window starts on a write operation and ends on the last read operation before the subsequent write one. Moreover, in order to not to be redundant, two faults corrupting the same location with the same corruption mask must be separated by a read action. It is worth mentioning that, the injection in the TLM ports does not require such analysis since communications are corrupted only during the execution of a transaction. Finally, more advanced rules, aware of the various functionalities (as the ones proposed in [39,40]), can be adopted for a more accurate pruning.

A second relevant aspect the methodology considers during the generation of the fault list is the extracted application-level information stored in the EFG. In past works [5,3] at most the architectural characterization is used to determine which component is affected by each generated fault; therefore, the performed fault injection campaign reported at most the statistics on the vulnerability of the various parts of the architecture. Here, the

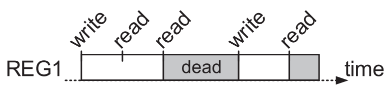

Fig. 6. Example of liveness time windows.

methodology exploits the EDF information, and in particular the timing and mapping data stored in each node, to classify the generated faults also in terms of the affected task. In this way, the final report will present also statistics in terms of the criticality and the vulnerability of the various tasks composing the application. Furthermore, when the aim is to evaluate specific reliability-oriented mechanisms implemented in specific functionalities, the EDF information can be used to generate faults focused only on the execution of the related function and, in case, to generate specific application-level errors in its execution and its input/output parameters. A final consideration is related to the necessity to provide a statistical representativeness of the experimental results, in particular, with respect to the number of injected faults. Specific techniques (e.g., [41]) can be integrated in the framework to support the definition of the fault list for each specific type of fault.

Apart from the support to the generation of a more efficient fault list, the liveness analysis is particularly interesting since it offers the opportunity to integrate into the methodology preliminary indexes that, similar to the Program Vulnerability Factor (PVF, [28]), state the vulnerability of the various memory locations of the system and of the various executed functions. The simple consideration is that the potential susceptibility of a register to a fault is directly proportional to the overall length of its liveness time windows. Later, in the last steps of the methodology, these percentage indexes may be used also to weight the statistical results obtained by the fault injection campaigns.

3.5. Definition of the monitoring and classification strategy

The core activity of the methodology consists in the definition of the monitoring and classification strategy. In fact, as discussed in the introduction, the methodology does not perform a straightforward comparison of the traces of the registers' values during the simulation against the golden counterpart, but an accurate analysis customized according to the specific system's properties and functionalities the designer aims at testing. For instance, the designer may be interested in assessing if an adopted hardening technique is able to handle the occurrence of faults, and, in case of a failure, s/he desires to identify the causes. The analysis strategy is based on a set of custom conditions expressed by the designer to monitor and analyze the status of the system with respect to the golden model at both architecture and application levels.

The *architecture-level conditions* aim at directly observing the status and the activity of the various architectural components. In particular, the designer may request to monitor either (i) a straightforward difference between the observed registers' values and the golden counterpart, or (ii) that architectural values satisfy a more complex condition ensuring the correct behavior of the system. For instance, the designer may need to assess that during the execution of each function, the processor does not perform any erroneous access to the bus, to avoid that, if affected by a fault, it corrupts data concurrently elaborated by other units. For this purpose, the designer will specify a set of conditions that, during the execution of each function, will evaluate whether the accessed addresses are within the range of values computed during the golden model characterization or not. It may be noticed that application level information, such as the scheduling and execution times of the functions, may be used for expressing such condition.

The second kind of conditions is called *application-level conditions*. In particular, according to the data provided by the application-level interpreter, the designer can express conditions for analyzing if the flow of executed functions is the expected one and the produced/exchanged data are correct. More precisely, a subset of events contained in the EFG can be considered as a sort of checkpoints which a condition is associated to. When considering the example in Fig. 2, a condition may be expressed on the exit of `RGBtoGray` function to check whether the produced image is correct with a tolerance of the 5% of wrong pixels or not.

The set of defined conditions needs to be implemented in a specific *monitoring and classification module* integrated in ReSP (shown in Fig. 7) that performs a "smart" monitoring of the error occurrence and propagation in the system: during the simulation, it will continue analyzing the status of the system exploiting the monitoring interfaces described in Section 3 to identify situations in which some conditions are violated. When a violation occurs, all relevant data are logged (e.g., the erroneous value, the processing unit experiencing the error, or the instant of time the event occurred). In this way, a detailed report on the error propagation is produced at the end of the simulation, describing the cause/effect relationship for the occurred faults and how the errors evolved among the architectural components and the application tasks.

The classification strategy supported by the methodology presents two peculiarities: (i) it can be customized by the designer according to the system's properties to be tested, and (ii) it is intended as a continuous activity providing a (partial) response during the overall simulation. To this purpose the set of classes to be used are customizable by the designer and are based on the application-level and architecture-level conditions defined above. Moreover, these classes are divided in *final classification* and *intermediate classification*. The former set describes the final status of the system at the end of the simulation, as usually done in the literature. Possible examples of classifications are shown in Fig. 8:

(a) **silent|observed,** to assess if faults activated any error visible on the results;

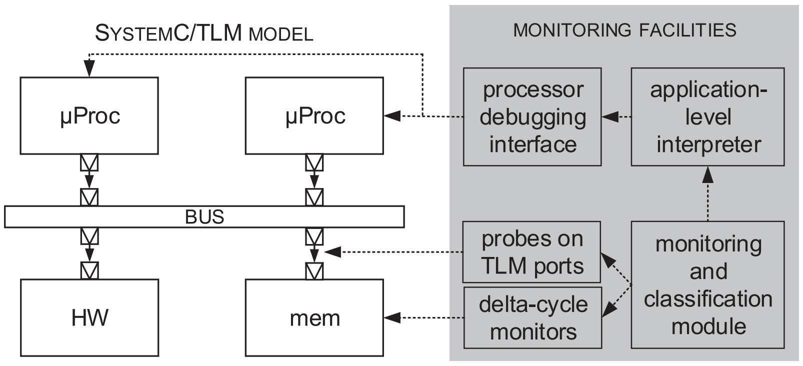

Fig. 7. The monitoring and classification module integrated in ReSP.

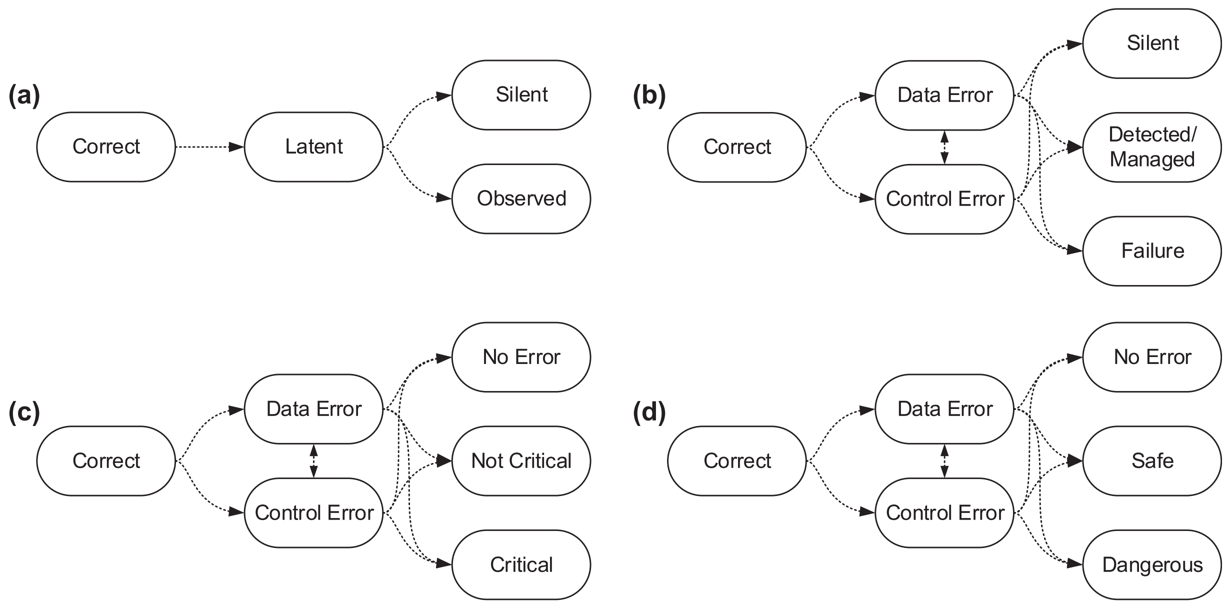

Fig. 8. Examples of classification.

(b) **silent|detected|failure,** to assess reliability mechanisms implemented into the system;
(c) **no-effect|critical|not critical,** used for both observable and silent faults, to distinguish the actual impact of the fault on the overall system behavior and output, and discriminating those faults for which mitigation strategies should be adopted;
(d) **no-effect|safe|dangerous,** used by FMEA in compliance with EN/IEC 61508, to evaluate the properties of the system with respect to functional safety.

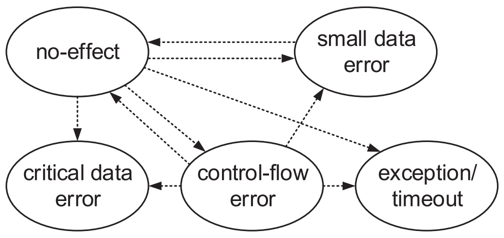

Fig. 9. Classifier for the edge detector.

Differently from the classical state-of-the-art approaches which uses at most the latent class, the methodology supports also the definition of classes for an accurate intermediate classification during the overall simulation. For instance, two intuitive but interesting intermediate classes involving the application-level analysis are the **data error**, when some function produces a wrong result, and the **control error**, when an unexpected event occurs in the execution flow graph. This set can be further specialized according to the properties of the system, and, in particular, the structure of the EFG. As shown in the examples in Fig. 8, intermediate and final classes are organized in a state diagram. Moreover classes are connected by transition edges specifying the possible evolution in the classification; transitions are fired according to the specified conditions. The monitoring and classification module within ReSP supports the specification of the classification strategy in order to automate such activity. During the simulation the module will exploit the monitoring data to decide the intermediate classification status, and at the end the final one.

As a conclusion, Fig. 9 shows the classification state diagram defined for the edge detector example in Fig. 2, omitting transition conditions for the sake of clarity. Do note that all the classes are final except for the **control-flow** error, which evolves into another class before the end of the simulation.

3.6. Statistical campaign and results' analysis

Once the set-up of the experiments has been performed, the fault injection campaign can be executed. The approach consists in a first execution of the experiments to collect statistical information on the dependability of the system from both the architectural and application points of view. During the subsequent post-processing step, the logs of the events of each experiment can be examined to mine further details on the cause/effect relationship of the occurred events. For instance, it may be noted how in the considered image processing example, a control flow error is more prone to cause a processor exception. Furthermore, experiments presenting interesting situations can be further analyzed with a more accurate simulation by means of more specific monitoring conditions to determine step-by-step the actual evolution and propagation of the errors in the system, and by specifying more focused fault lists for stimulating the system.

Since the definition of an accurate classification strategy can be a complex activity, the designer can also exploit the possibility to re-execute the experiment execution phase and the previous set-up phase to perform iterative refinements on that strategy. In particular, in the first iterations a preliminary classification diagram is considered and the focus is devoted on the error propagation analysis. Based on the outcomes on the error propagation, the designer can accurately refine the classification diagram to be used during the re-execution of the experiments to produce an accurate report on the causes of the failures.

4. Case studies

The proposed methodology has been employed in two case studies considering a hardened version of the image processing system used in the paper as a running example and an automotive anti-lock braking system (ABS). For sake of simplicity, the two case studies have a limited complexity to give the opportunity to discuss in details the results and the outcomes of the application of the methodology; for this same reason the multiprocessor architecture has been used only to execute replicated threads. Nevertheless the methodology is effective also in more complex scenarios considering multi-threaded applications.

4.1. Case study 1: The edge detector system

In the first case study we consider the image processing application. The architectural platform is based on an ARM multiprocessor connected through a bus with a shared memory and to an input/output peripheral. The static architectural analysis is performed to retrieve the list of observation points and injection ones. The first list contains the debugging interface of the various processors and the probes on the interconnection with the bus; the last ones are used also for monitoring the activities of the shared memory. The list of injection points includes the register bank of the ARM processor (containing also the status registers) and the memory cells. We also model faults on the memory controller, the bus and the processor interfaces in terms of corruptions of the transactions by means of sabouters. All fault lists considered during the experiments are generated starting from the liveness analysis.

In the preliminary proposal of the analysis framework [34], we investigated the vulnerability of the various processor's registers and application's functions. In this case study, we assume that a designer decides to harden the considered system by means of the Triple Modular Redundancy (TMR) applied at the software level and using an architecture with 3 different processors, as shown in Fig. 10; the leftmost processor executes the `main` function, one of the replicas and the voter function, while the other two processors execute the secondary thread replicas. We investigate

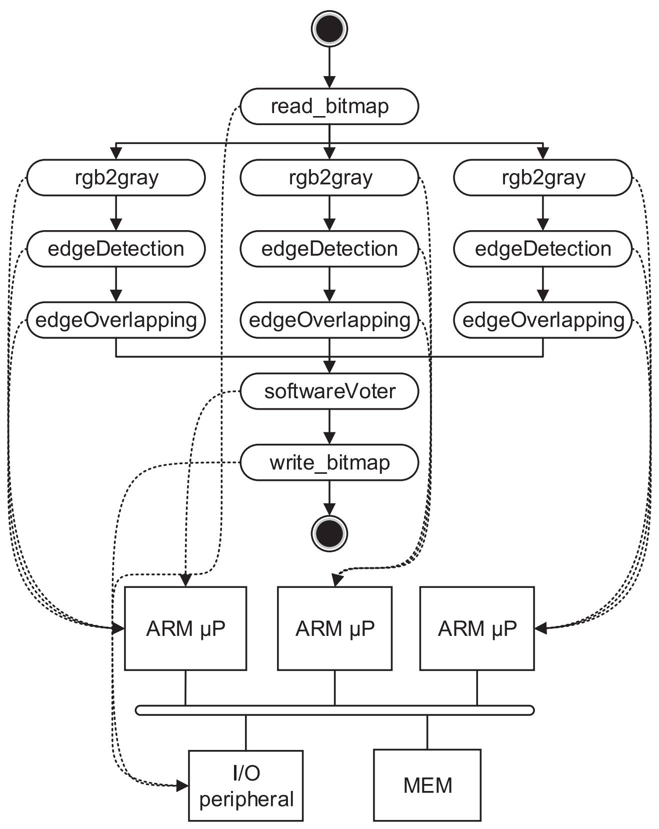

Fig. 10. The hardened version of the edge detector system.

the reliability of such a solution in particular focusing on the robustness of using three replicas.

In a first experimental campaign we aim at analyzing the final outputs of the system (as generally done in the literature), and we consequently specify a classification strategy using only final classes and no monitoring strategy. We perform a campaign with 10,000 injected in the shared memory and other 10,000 in the processors' registers. It is worth noting that in this case study (and also in the following one) the fault list a simple fault list is generated for demonstrating the capabilities of the approach to provide an accurate analysis fault/error relationship and to identify possible limitations in the adopted hardening mechanisms, without aiming at generating a statistically representative fault list.

The obtained results are presented in Table 2. RB0 to RB3 are the registers used for transmitting the parameters and for performing most of the elaborations, while the Link Register, the Program Counter (PC) and the Stack Pointer are the three main control registers. Do note that some of the registers are omitted for sake of space; moreover, results of the same registers of different processors are aggregated since they almost presented a similar trend. The table shows that the processors' registers are more critical than the shared memory. In fact, when corrupting the memory, only in the 2% of the experiments an erroneous image is produced, and in the 5.3% an exception is risen or a timeout expired (we set a maximum execution time within the system has to generate the result). Do note that the exception handler is programmed to reset the system; as a consequence, exceptions cause a timeout. On the other side, the processors' registers present a higher percentage of cases in which the final image is corrupted or an exception is thrown, especially when considering control registers such as the program counter.

A second in-depth experimental campaign is performed to investigate the causes of the two problematic situations highlighted during the previous run: the generation of an erroneous image and the high number of timeouts (and exceptions). To this purpose, two basic conditions on the processors are defined for the monitoring and classification strategy: on each enter and exit event, we check that (i) the partial EFG is equal to the golden one, and (ii) the data transmitted on each event are correct. Moreover, probes are inserted on the processors' ports to monitor the accessed memory addresses; to this purpose the golden model characterization is executed again to collect the range of addresses accessed by each function. In case of a mismatch, the monitoring and classification module is programmed to log the wrong status.

When we re-execute the relevant experiments for the first critical situation with the new conditions, we note that on the exit events subsequent the injection of the fault two replicas of the image were corrupted, and several cases also the register containing one of the pointers to the input/output images (for the considered functions RB0 and RB1 in the register bank) contained a wrong value. As a final confirm, the data provided by the monitoring probes show that the TMR strategy failed because one of the thread replicas started writing on the memory space of another thread.

In order to refine the classification of the collected results, we re-performed all the experimental campaign by using the described monitoring conditions, and we noted that the 34.6% of the faults causing an error propagated to the registers RB0 and RB1 causes an exception as a consequence of a wrong access to the memory. These results demonstrate that to avoid TMR failures it is necessary to isolate the memory spaces accessed by the three thread replicas. For sake of completeness, we report in Fig. 11 the overall error propagation diagram for the data error, generated on the basis of the logged information; some of the details (such as the transition conditions and the event on which the condition has been violated) have not been displayed for readability reasons.

Table 2
Statistical results on the final effects of the faults injected in the memory and in the registers for the edge detector case study.

Class	Memory (%)	RB0–RB1 (%)	RB2–RB3 (%)	Link Reg. (%)	SP Reg. (%)	PC Reg. (%)
No effect	92.7	35.2	79.9	22.0	30.6	22.5
Errors	2.0	30.2	7.7	0	1.4	2
Exc./T.O.	5.3	34.6	12.4	78	68	75.5

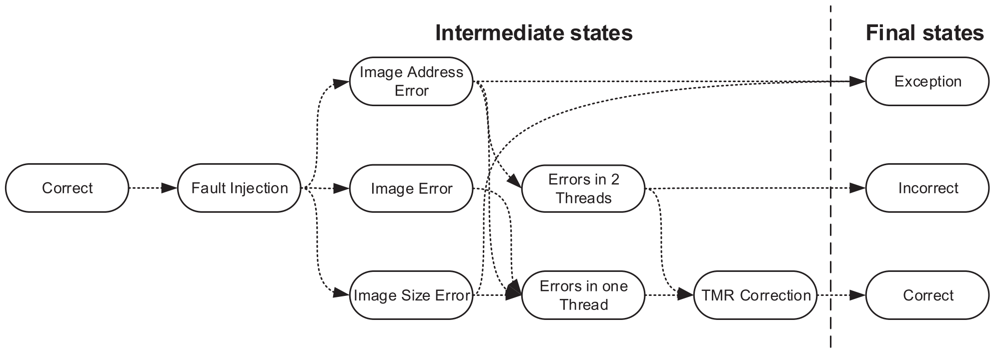

Fig. 11. The possible propagations of a data error.

When analyzing the experiments reporting an exception or a timeout, the monitoring of the sequences of function enter/exit events show that the cause of the timeout is mainly derived by a thread synchronization; in fact, when a secondary thread raises an exception, the processor is reset and the primary thread (the one executing the main function) is stuck on the `join` function call. Fig. 12 displays the propagation of control errors generated by injecting faults in the PC register of the processors running secondary threads. Experimental results show that in very few cases, the control error causes a short jump backward (re-executing few instructions), whose effect is a short delay, or the jump forward (skipping few instruction), whose effect is a data error mitigated by the voter. Instead, in most cases, the control error causes an exception and consequently a timeout. Finally, the few cases generating a corrupted image are caused by a jump forward to the end of the `main` function or in the voter one. In order to overcome this limitation, it is necessary to decouple the call of the voter from the end of the thread replicas. In particular, the voter should be invoked at the expiration of a timeout, so that its call is not strictly dependent on the correct termination of the secondary threads.

We also perform a fault injection campaign focused on the voter; we note that also that function presented a similar vulnerability to the other functions (9% of cases presenting a corrupted image and 40% of cases raising an exception or a timeout); therefore, we find out that the voter has to be executed by a specific hardened hardware module. In conclusion, the dependability analysis presented in this case study offers the possibility to identify the pitfalls of the TMR implemented in software and the related causes; according to these results, the hardening mechanism can be improved.

4.2. Case study 2: The anti-lock braking system

In the second case study we consider an automotive anti-lock braking system (ABS) executed on the multiprocessor architecture shown in Fig. 13. During an emergency brake, the system aims at controlling the braking activity until the vehicle is halted; therefore the algorithm is based on a loop of three main tasks: acquisition of the vehicle' and wheels' speeds from specific sensors, computation of the braking strength and actuation of the brake. The system is hardened (i) by duplicating the sensors and data processing tasks triggering a re-execution in case of detected error, and (ii) by using a control-flow signature checking schema [42]. Moreover, when an error is detected, a safety routine is called to reset the application and restore the system in safe state. Do note that in the figure, dashed arrows representing the mapping of fork, join and checking tasks have not been drawn for the sake of clearness in the picture; these tasks are mapped on the same processor that executed the `main` function.

The goal of the fault injection campaign is to establish whether the ABS is able to halt the vehicle within a given deadline or not, and in this second case, to investigate the causes of the failure.

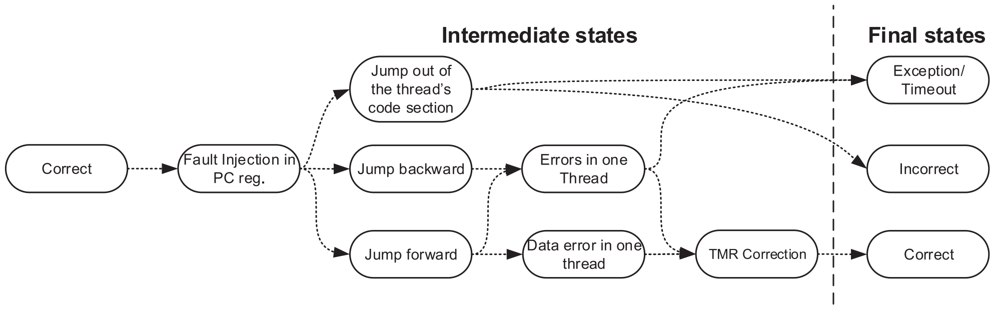

Fig. 12. The possible propagations of a control error caused by a fault in the PC register of a secondary processor.

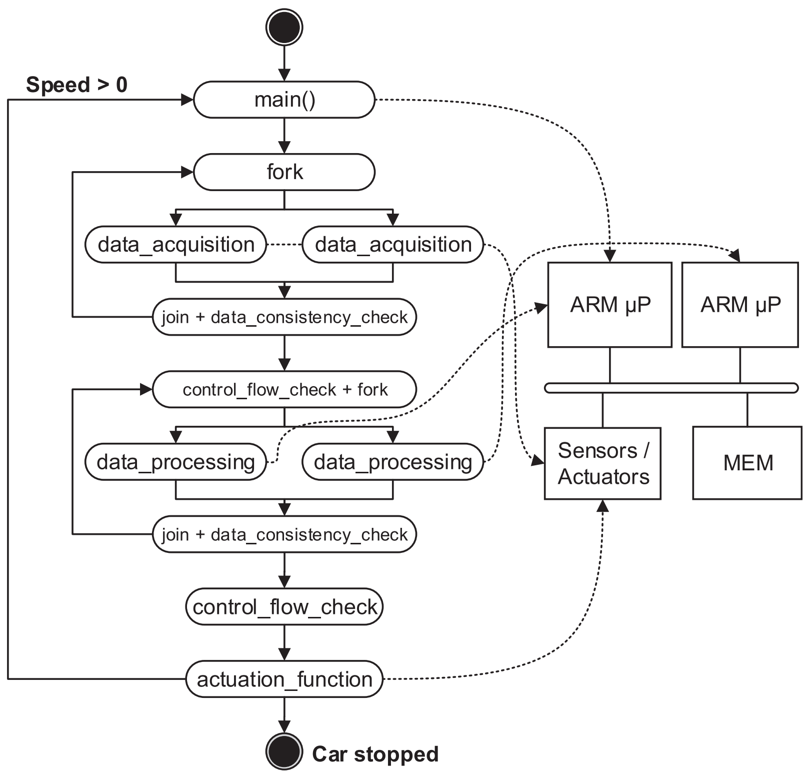

Fig. 13. The hardened version of the anti-lock braking system.

The preliminary activities for the characterization of the system is carried out similarly as discussed for the previous case study; in particular, we consider similar fault models. Moreover, due to the safety characteristics of the application, the final classification in Fig. 8d is adopted, and the following list of monitoring conditions are defined: (i) the sequence of enter/exit events must be equal to the golden one, (ii) the data transmitted on each event must be correct, (iii) if the safety function is called, an error must have been previously detected, and (iv) if a data error is detected on an event, the safety function must be subsequently called. Similarly to the previous case, in case of a mismatch, the monitoring and classification module is programmed to log the wrong status.

An experimental campaign is carried out by injecting 10,000 faults in the processors' registers and other 10,000 in the memory. Table 3 reports some aggregated statistics on the final effects. It is possible to notice that the percentage of dangerous situations is under the 10%; in fact, in the most of the cases, the fault generates an error that is then correctly detected and managed by the safety function. When analyzing the internal behavior of the system logged by means of the monitoring conditions (graphs similar to the ones in Figs. 11 and 12), we draw the following comments.

- Approximately in the 50% of the overall number of safe situations, the safety function is invoked by a processor exception, especially when corrupting the control registers.
- The task duplication and re-execution is able to handle the most of the data errors occurred during the data acquisition and processing; for instance, the 66.7% of errors affecting registers RB0 and RB1 are corrected by the technique while the left 15.3% are managed by the call to the safety function.
- The control-flow signature checking schema is a valid strategy to detect the few control errors not causing any exception.
- Errors affecting the actuation are directly mitigated by the algorithm. In fact, such an error causes a wrong brake actuation for a single iteration; therefore, during subsequent iterations the algorithm will adjust the actuation according to the newly sensed data.

When analyzing the dangerous situations, the main cause is again related to a thread synchronization problem that also the edge detector system suffers. For instance, in the 1.6% of dangerous situations collected for the registers RB0 and RB1, the secondary processor is affected during the execution of the replicated data acquisition thread or processing one; in these situations the thread does not return, thus causing the main thread to be stuck on the `join` function. In other situations, when the PC register of the main processor is affected, the `fork` function was skipped and consequently the thread blocked on the subsequent `join`. In conclusion, the vulnerabilities of the adopted fault management mechanisms are related to thread synchronization errors. It can be solved by

Table 3
Statistical results on the final effects of the faults injected in the memory and in the registers for the ABS case study.

Class	Memory (%)	RB0–RB1 (%)	RB2–RB3 (%)	Link Reg. (%)	SP-PC Reg. (%)
No effect	37.7	16.4	67	3.2	6.8
Safe	57.7	82	33	91.7	85.6
Dangerous	4.6	1.6	0	5.1	7.6

enhancing the `join` function with a timeout mechanism that forces the exit from the blocking call and the call to the safety routine.

As a final note, the execution of the 20,000 simulations required altogether 60 h for the first case study and 32 h for the second one on a 3.15 GHz Intel core 2 duo running Ubuntu. This performance is specifically related to the adopted simulator rather than to the amount of data collected and analyzed. To improve performance it is possible to port the proposed analysis framework in a more efficient simulation environment or in an emulation-based fault injector, an operation that will be taken into account in future work.

5. Conclusions

This paper has presented a methodology for the system-level dependability analysis of multiprocessor embedded systems. The methodology is based on fault injection and features an error analysis approach offering to the designer the possibility to specify custom monitoring and classification conditions at both application and architecture levels. A companion framework has been implemented within ReSP, a state-of-the-art SystemC/TLM simulation platform, for multiprocessor specifications featuring fault injection facilities. A debug-like mechanism offers the possibility to interpret architectural raw data observed during the simulation at application level with a function call/return granularity, thus offering the possibility to analyze the propagation of the errors in the various functionalities of the executed application. The experimental results in two case studies have demonstrated its effectiveness in the production of an accurate dependability report highlighting the criticalities in both the architecture and the application of the system under design.

Acknowledgements

The author would like to thank Stefano Guidobaldi (graduated at Politecnico di Milano) for his support in the development of the case studies. Moreover, the author thanks “manufacturable and dependable multicore architectures at nanoscale” (MEDIAN) European COST Action IC1103, which partially supported the proposed work.

References

[1] C. Constantinescu, Trends and challenges in VLSI circuit reliability, IEEE Micro 4 (2003) 14–19.
[2] A. Benso, P. Prinetto, Fault Injection Techniques and Tools for Embedded Systems, Kluwer Academic Publishers, 2003.
[3] D. Lee, J. Na, A novel simulation fault injection method for dependability analysis, IEEE Des. Test Comput. 26 (2009) 50–61.
[4] A. Pellegrini, K. Constantinides, D. Zhang, S. Sudhakar, V. Bertacco, T. Austin, CrashTest: a fast high-fidelity FPGA-based resiliency analysis framework, in: Intl. Conf. on Computer Design, IEEE, 2008, pp. 363–370.
[5] M. Aguirre, J. Tombs, F. Muoz, V. Baena, H. Guzman, J. Napoles, A. Torralba, A. Fernandez-Leon, F. Tortosa-Lopez, D. Merodio, Selective protection analysis using a SEU emulator: testing protocol and case study over the LEON2 processor, IEEE Trans. Nucl. Sci. 54 (2007) 951–956.
[6] P. Civera, L. Macchiarulo, M. Rebaudengo, M.S. Reorda, M. Violante, An FPGA-based approach for speeding-up fault injection campaigns on safety-critical circuits, J. Electron. Test. 18 (2002) 261–271.
[7] E. Touloupis, J. Flint, V. Chouliaras, D. Ward, Study of the effects of SEU-induced faults on a pipeline protected microprocessor, IEEE Trans. Comput. 56 (2007) 1585–1596.
[8] D. de Andrés, J. Ruiz, D. Gil, P. Gil, Fault emulation for dependability evaluation of VLSI systems, IEEE Trans. Very Large Scale Integr. (VLSI) Syst. 16 (2008) 422–431.
[9] G. Beltrame, L. Fossati, D. Sciuto, ReSP: a nonintrusive transaction-level reflective MPSoC simulation platform for design space exploration, IEEE Trans. CAD Integr. Circ. Syst. 28 (2009) 1857–1869.
[10] SystemC, Open SystemC Initiative. <www.systemc.org> (accessed 10.10.13).
[11] C. Bolchini, A. Miele, D. Sciuto, Fault models and injection strategies in SystemC specifications, in: Proc. Euromicro Conf. on Digital System Design, IEEE, 2008, pp. 88–95.
[12] A. Manuzzato, P. Rech, S. Gerardin, A. Paccagnella, L. Sterpone, M. Violante, Sensitivity evaluation of TMR-hardened circuits to multiple SEUs induced by alpha particles in commercial SRAM-based FPGAs, in: Intl. Symp. on Defect and Fault-Tolerance in VLSI Systems, IEEE, 2007, pp. 79–86.
[13] D. Appello, P. Bernardi, S. Gerardin, M. Grosso, A. Paccagnella, P. Rech, M.S. Reorda, DfT reuse for low-cost radiation testing of SoCs: a case study, in: Proc. VLSI Test Symposium, IEEE, 2009, pp. 276–281.
[14] J. Carreira, H. Madeira, J. Silva, Xception: a technique for the experimental evaluation of dependability in modern computers, IEEE Trans. Softw. Eng. 24 (1998) 125–136.
[15] M. Portela-García, C. López-Ongil, M. García-Valderas, L. Entrena, Fault injection in modern microprocessors using on-chip debugging infrastructures, IEEE Trans. Dependable Secure Comput. 8 (2011) 308–314.
[16] L. Entrena, M. Garcia-Valderas, R. Fernandez-Cardenal, A. Lindoso, M. Portela, C. Lopez-Ongil, Soft error sensitivity evaluation of microprocessors by multilevel emulation-based fault injection, IEEE Trans. Comput. 61 (2012) 313–322.
[17] A. da Silva, S. Sanchez, LEON3 ViP: a virtual platform with fault injection capabilities, in: Proc. Conf. on Digital System Design (DSD), IEEE, 2010, pp. 813–816.
[18] R. Shafik, P. Rosinger, B. Al-Hashimi, SystemC-based minimum intrusive fault injection technique with improved fault representation, in: Proc. Intl. On-Line Testing Symposium, IEEE, 2008, pp. 99–104.
[19] J. Perez, M. Azkarate-askasua, A. Perez, Codesign and simulated fault injection of safety-critical embedded systems using SystemC, in: European Dependable Computing Conference, IEEE, 2010, pp. 221–229.
[20] A. Bosio, G.D. Natale, LIFTING: a flexible open-source fault simulator, in: Proc. Asian Test Symposium, pp. 35–40.
[21] F. Lu, G.D. Natale, M.-L. Flottes, B. Rouzeyre, Laser-induced fault simulation, in: Proc. Conf. Digital System Design, pp. 609–614.
[22] A. Ejlali, S. Miremadi, Error propagation analysis using FPGA-based SEU-fault injection, Microelectron. Reliab. 48 (2008) 319–328.
[23] R. Mariani, G. Boschi, A systematic approach for failure modes and effects analysis of system-on-chips, in: Intl. On-Line Testing Symposium, IEEE, 2007, pp. 187–188.
[24] Y.-Y. Chen, C.-H. Hsu, K.-L. Leu, SoC-level risk assessment using FMEA approach in system design with SystemC, in: Proc. Intl. Symp. Industrial Embedded Systems (SIES), IEEE, 2009, pp. 82–89.
[25] A. Rohani, H.-R. Zarandi, An analysis of fault effects and propagations in AVR microcontroller ATmega103(L), in: Proc. Int. Conf. on Availability, Reliability and Security (ARES), IEEE, 2009, pp. 166–172.
[26] A. Ammari, K. Hadjiat, R. Leveugle, Combined fault classification and error propagation analysis to refine RT-level dependability evaluation, J. Electron. Test. 21 (2005) 365–376.
[27] S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, T. Austin, A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor, in: Intl. Symp. on Microarchitecture, ACM, 2003, pp. 29–40.
[28] V. Sridharan, D. Kaeli, Eliminating microarchitectural dependency from architectural vulnerability, in: Proc. Int. Symp. on High Performance Computer Architecture (HPCA), IEEE, 2009, pp. 117–128.
[29] I. Oz, H. Topcuoglu, M. Kandemir, O. Tosun, Quantifying thread vulnerability for multicore architectures, in: Proc. of Euromicro Intl. Conf. on Parallel, Distributed and Network-Based Processing (PDP), IEEE, 2011, pp. 32–39.
[30] R. Baranowski, S. Di Carlo, N. Hatami, M.E. Imhof, M.A. Kochte, P. Prinetto, H.-J. Wunderlich, C.G. Zoellin, Efficient multi-level fault simulation of HW/SW systems for structural faults, Sci. China Inform. Sci. 54 (2011) 1784–1796.
[31] G. Beltrame, C. Bolchini, A. Miele, Multi-level fault modeling for transaction-level specifications, in: Proc. of the Great Lakes Symposium on VLSI (GLSVLSI), ACM, 2009, pp. 87–92.
[32] S. Reiter, M. Pressler, A. Viehl, O. Bringmann, W. Rosenstiel, Reliability assessment of safety-relevant automotive systems in a model-based design flow, in: Proc. Asia and South Pacific Design Automation Conference (ASP-DAC), ACM, 2013, pp. 417–422.
[33] M. Michael, D. Grobe, R. Drechsler, Analyzing dependability measures at the electronic system level, in: Proc. Forum on Specification and Design Languages (FDL), pp. 1–8.
[34] C. Bolchini, A. Miele, An application-level dependability analysis framework for embedded systems, in: Proc. Int. Symp. on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), IEEE, 2011, pp. 171–178.
[35] Politecnico di Milano, ReSP web site. <http://code.google.com/p/resp-sim/> (accessed 10.10.13).
[36] A. Miele, C. Pilato, D. Sciuto, A simulation-based framework for the exploration of mapping solutions on heterogeneous MPSoCs, Int. J. Embed. Real-Time Commun. Syst. (IJERTCS) 4 (2013) 22–41.
[37] ARM, Application Binary Interface (ABI) for the ARM Architecture. <http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.subset.swdev.abi/index.html> (accessed 10.10.13).
[38] S. Campagna, M. Violante, An hybrid architecture to detect transient faults in microprocessors: an experimental validation, in: Design, Automation Test in Europe Conf. (DATE), IEEE, 2012, pp. 1433–1438.

[39] A. Benso, M. Rebaudengo, L. Impagliazzo, P. Marmo, Fault-list collapsing for fault-injection experiments, in: Proc. Annual Reliability and Maintainability Symposium, IEEE, 1998, pp. 383–388.

[40] S. Hari, S. Adve, H. Naeimi, P. Ramachandran, Relyzer: exploiting application-level fault equivalence to analyze application resiliency to transient faults, in: Proc. Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS), ACM, 2012, pp. 123–134.

[41] R. Leveugle, A. Calvez, P. Maistri, P. Vanhauwaert, Statistical fault injection: quantified error and confidence, in: Proc. Conf. Design, Automation, and Test in Europe, IEEE, 2009, pp. 502–506.

[42] O. Goloubeva, M. Rebaudengo, M.S. Reorda, M. Violante, Soft-error detection using control flow assertions, in: Proc. Int. Symp. on Defect and Fault Tolerance in VLSI Systems (DFT), IEEE, 2003, pp. 581–588.