H2020-COMPET-2014

MaMMoTH-Up

Massively extended Modular Monitoring for Upper Stages

Type of Action: Research and Innovation Action (RIA)

Topic: COMPET-02-2014 Independent access to space

Grant Agreement no: 637616

MaMMoTH-Up
MASSIVELY EXTENDED MODULAR MONITORING FOR UPPER STAGES

Report on fault injection and self- testing on a selected case study (draft) (v. 4.0) (Deliverable D4.3)

Start date of the project: May 1, 2015 Duration: Three years

Organisation name of lead contractor for this deliverable: PDT

Project funded by the European Commission within the Horizon 2020 programme for research, technological development and demonstration (2015-2018)		
Dissemination Level		
PU	Public, fully open	☒
CO	Confidential, restricted under conditions set out in Model Grant Agreement	☐
CL	Classified	☐

Notices

Table of Revisions

Version	Date	Description and reason	Author	Affected sections
0.1	15-9-2016	First draft version	M. Violante	All
1.0	9-12-2016	Section 4 added	R. Schmidt	Section 4
2.0	12-12-2016	Added Section 1 and Section 2	M. Violante	Section 1, Section 2
3.0	12-12-2016	Modified Section 4		Section 4
4.0	15-12-2016	Modified Section 1, 3, 5, and 6	R. Schmidt, G. Fey	Section 1, 3, 5 and 6

Author, Beneficiary

- Serhiy AVRAMENKO, PDT
- Stefano ESPOSITO, PDT
- Matteo SONZA REORDA, PDT
- Massimo VIOLANTE, PDT
- Robert SCHMIDT, DLR
- Görschwin FEY, DLR

Executive Summary

The following document presents the techniques proposed within the MaMMoTH-Up project for dependability evaluation.

.

List of Abbreviations

Acronym	Meaning
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
AQB	Acquisition board
COTS	Commercial-Off-The-Shelf
CPU	Central Processing Unit
FPGA	Field Programmable Gate Array
GDB	The GNU project debugger
HDL	Hardware Description Language
ISS	Instruction Set Simulator
MBU	Multiple bit upsets
MTBF	Mean Time Between Failures
NMR	N-Modular Redundancy
OBCS	On-board Computing System
TC/TM	Telecommand board
SBST	Software-based Self-Test
SET	Single Event Transient
SEU	Single Event Upset
SoC	Systems on a Chip
SW	Software
TMR	Triple Modular Redundancy

Table of Contents

1 Introduction

This deliverable describes the work done for assessing the dependability of a case study consisting of a microprocessor-based system for compressing telemetry data during the ground to orbit mission of a space launcher such as Ariane 5 or Ariane 6.
Two aspects are considered, fault injection to assess the impact of transient faults in the system during the mission, and self-testing to guarantee functional integrity during the mission.
The document is organized as follows: Section 2 presents the selected case study for which fault injection techniques are presented in Section 3. An outlook on further radiation environment modeling for fault injection is given in Section 4. Section 5 exposes the self-test and compaction for the selected case study, and Section 6 concludes.

2 The selected case study

The case study we considered is a single board computer (on-board computing system, OBCS) hosting an OpenRisc microprocessor and static RAM to store program/data the processor elaborates.
The computer performs the compression of sensor data coming from different instruments and prepares the telemetry packets to be sent to ground stations.
In our analysis, we assume that:

- Input data are provided by an acquisition board (AQB) that manages the interface with the instruments and loads the data in a shared RAM accessible to the OBCS;
- The OBCS runs the compression algorithm computing the compressed bitstream and stores it in a shared RAM with the telemetry and telecommand board (TC/TM);
- The transmission of the compressed bitstream to the ground station is done by the TC/TM board;
- The target of both fault injection and self-testing activities is the OBCS, only;
- We considered different compression algorithms to evaluate their intrinsic robustness with respect to the fault of concern;
- As far as fault models are concerned, we considered single bit flips as model of radiation-induced transient faults that may affect the OBCS behavior during the mission, and permanent stuck-at faults as model of the permanent fault due to the harshness of the launch phase.

3 Fault injection techniques for the selected case study

The approach we develop uses the C version of the algorithm under investigation, and performs fault injection campaigns where bit flips are applied to the variables of the programs, and then on classifying the resulting effects.
The goal of our work is to get a first draft indication about the percentage of faults leading to misbehaviors. By repeating fault injection on several different algorithms, the obtained results provide precise indications about which algorithm is likely to

produce the highest number of misbehaviors due to transient faults, and about the probability that faults are detected by intrinsic mechanisms, e.g., related to memory management. Clearly, the obtained figures can in no way forecast the final dependability figures of the final system, which also depend on the amount of memory used by each algorithm (which impacts on its sensitivity to radiation, which also depends on the possible mechanisms for protecting the different memory levels) and on its duration (the longer the time required to perform a given task, the higher the probability that a fault affects it).
The set-up we devised for fault injection execution is composed of a run-time environment and a fault injection environment, as described in the following sections, and depicted in Figure 1.

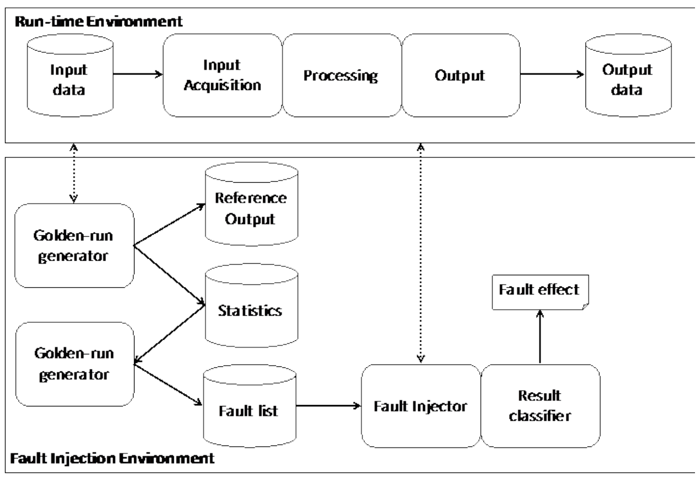

Figure 1. The experimental set-up

3.1 The run-time environment

The run-time environment is intended for executing an algorithm on a pre-defined set of input data. The software executed in the run-time environment is composed of the following modules:

- *Input acquisition module*: this module is in charge of loading from the file system the pre-defined set of input data in memory. This module makes use of the operating system API to read data from a binary file and to store them into a suitable input memory array.
- *Processing module*: this module is in charge of running the algorithm under analysis on the input memory array, producing an output memory array. This module is coded in C language and does not make use of any operating system API. Thanks to this approach, the compression algorithm is highly portable, and can be reused in any target hardware platform (e.g., an embedded processor, as well as an ASIC/FPGA after high-level synthesis).
- *Output module*: this module is in charge of downloading the output memory array into the file system, using operating system API to create a binary file.

We implemented three versions of the run-time environment as described in the following:

- *Native run-time environment*, where the input acquisition, processing and output modules are executed through a Linux-powered x86 machine, which is also executing the fault injection environment;

- *Instruction set simulator (ISS)-based run-time environment,* where the input-acquisition, processing and output modules are executing through an instruction set simulator of an embedded processor. This implementation allows to validate the results obtained using the native environment and to capture hardware-specific behaviors. In this implementation, the input acquisition and output modules are not used to avoid dependencies with any operating system. The processing thus makes use of an input memory array pre-initialized with the input data set, and runs on the simulated processor as bare metal code;
- *HDL-based run-time environment*, where the where the input-acquisition, processing and output modules are executing through an HDL simulator of a Verilog model of the OBCS. In this implementation, the input acquisition and output modules are not used to avoid dependencies with any operating system. The processing thus makes use of an input memory array pre-initialized with the input data set, and runs on the simulated processor as bare metal code.

3.2 The fault injection environment

The fault injection environment we devised allows to evaluate the effects of single bit flips in the data structures of a program in case of native run-time environment, or the simulated processor registers in the case of the ISS-based run-time environment, or any signal in case of HDL-based run-time environment. The fault injection environment is composed of four modules:

- *Golden-run generator*, which runs the algorithm once to collect: data needed for generating the list of faults to be injected, and the reference output data set to classify fault effects. More in details, the golden-run generator collects:
    - Statistics about the data structure/registers the algorithm uses while processing the pre-defined set of input data. Only those data structures/registers that are read/written more than one time during data compression are considered for fault injection. The rationale behind this approach is that algorithms may have data structures (such as large arrays) that are initialized once during execution (typically at the beginning), few entries of which are actually used to store meaningful data during algorithm execution. As a result, by avoiding injecting faults into unused data structures during algorithm execution we can save a significant amount of runtime.
    - The binary file storing the output memory array, which is used as reference for classifying the effects of faults during fault injection.
- *Target generator*, which is in charge of producing the list of faults to be injected. Based on the memory elements the algorithm (including both scalar variables such as temporary variables, indexes, as well as large data array) or the set of registers in the simulated processor, or the signals of the HDL model, each defined as the tuple (*identifier name, size*), and the average execution time of the algorithm, the target generator produces randomly a predefined number of faults (i.e., the *fault list*), each defined as the tuple (*identifier name*, *bitmask*, *injection time*) where bitmasks correspond to the single bit in the selected identifier that must be bit-flipped at the injection time to model the effect of a SEU. A shell script is produced for each fault

containing the instructions to run the fault injector, and the result classifier described in the following;

- *Fault injector,* which is a script that for each fault starts the execution of the program, advances the execution until injection time, inoculates the bit-flip in the identifier to attack, and resumes the execution of the program until its termination. The fault injection makes use of the run-time environment for algorithm execution: in case of native run-time environment, the algorithm is executing through the GDB debugger; in the case of ISS-based run-time environment the ISS of choice supports remote target connection to GDB; in case of HDL-based run-time environment the MentorGraphics ModelSim HDL simulator is used. Fault injection takes place after the input acquisition module completed its activity, and before the processing module completed. As a result, faults are inoculated only during the execution of the algorithm under investigation.
- *Result classifier,* which is a shell script to analyze the termination status of the program under fault injection, and to compare the produced outputs with respect to a predefined reference (golden run).

Faults are classified according to the following categories:

- *Silent*: the faulty execution completes within a pre-defined amount of time and produces the same results as the golden run.
- *Timeout*: the faulty execution does not complete within a pre-defined amount of time (i.e., the time needed for completing SW execution during golden run).
- *Wrong output*: the faulty execution completes within a pre-defined amount of time and the produced results differ from the golden run.
- *Detected:* the faulty execution completes within a pre-defined amount of time and triggers some error detection mechanism. In the case of native run-time environment the exception is the segmentation fault produced as a result of a memory access violation. In the case of ISS-based run-time environment, the category is further detailed according to the exception the processor notified, that are:
    - *Bus error*, when the injected fault leads the program to access and invalid memory address.
    - *Alignment error*, when the fault leads the program to access a misaligned memory address.
    - *Illegal instruction*, when the fault leads the processor to access a memory area not containing valid instructions.
    - *Other exception*, when the faults triggered exceptions other than the previous ones (such as the division by zero exception).

3.3 Preliminary results

We performed our experiments on an Intel Core i7-powered workstation running a 64-bit Linux distribution. The run-time environments have been prepared to accommodate the execution of the two algorithms we considered: LZW and RICE. The characteristics of the two programs are reported in Table I, where we can find:

- The number of lines of code composing the two programs (*Lines of code*). We considered only the lines of code actually used to implement the compression algorithm, while we do not consider the lines of code for implementing the

Input/Output operations needed to load into memory from the file system the data to be compressed, and to store the compression results to the file system.

- The *average execution time* for the two programs on the same set of input data.
- The *memory occupation* of the two programs. In this column we report only the size of the data structure needed for the compression, while the input/output buffers are neglected. From this column the much higher footprint of the LZW program is evident that, being based on a dictionary, requires a much higher amount of memory compared to RICE.

TABLE I. PROGRAM CHARACTERISTICS

	Lines of code [#]	Average execution time [μs]	Memory occupation [bytes]
RICE	180	2,753.00	64
LZW	110	1,827.00	21,091

As far as the ISS-based run-time environment is considered, we used the OpenRisc 1200 ISS as this soft-core is being considered as possible target hardware for our telemetry system.

We collected an initial set of results considering 100.000 randomly-generated faults using the native run-time environment. The results we obtained are reported in Table II.

By analyzing the results, we can draw the following conclusions:

- As far as the faults classified as Silent are considered, we have that about 89% of faults fall in this category for LZW, while about 50% for RICE. We believe that this result does not indicate a superior robustness of LZW versus RICE. Indeed, if we consider the memory occupation of the two algorithms, we can see that LZW occupies 300x the memory used by RICE, and due to the considered input data set most of this memory, which is devoted to store the dictionary, is never referenced. As a result, a large number of randomly-generated faults affect unused, allocated memory, resulting in no visible effect on the program execution.
- As far as the faults classified as Timeout are considered, we can see that for LZW about 0.1% of the injected faults resulted in an abnormal program duration, while this is not the case for RICE. This result can be explained by considering the nature of the two compression techniques. Being based on a dictionary, LZW makes lookups in a large data array for each data word to be coded. As a consequence, in case the fault affects the index used for the search, it is possible that the number of iterations of the table lookup is affected greatly, thus leading to an abnormal program runtime. Conversely, RICE being based on simple arithmetic operations it is insensible to such kind of effects.
- As far as the faults classified as Wrong Output are considered, we can see a substantial difference between the two programs, with faults producing about 11% of Wrong Output in LZW, and about 48% in RICE. We expect that the limited number of corrupted execution in LZW is again due to the high

number of unused memory locations in the dictionary. Conversely, when considering the faults classified as Detected, we can see that about 0.02% of the injected faults fall in this category for LZW, while about 3% for RICE.

TABLE II. NATIVE RANDOM FAULT INJECTION RESULTS

	LZW		RICE	
	[#]	[%]	[#]	[%]
Silent	88,982	88.98	49,330	49.33
Timeout	94	0.09	0	0.00
Wrong Output	10,906	10.91	47,834	47.83
Detected	18	0.02	2,836	2.84
TOTAL	100,000		100,000	

To better investigate the effects of faults in the two algorithms, we repeated the fault injection experiments using the native run-time environment, by generating the list of faults to be injected considering the actual usage of each memory structure. In this case, the number of faults generated for a given identifier is proportional to the number of times the identifier is referenced during the algorithm execution. As a result, a data structure seldom used by the algorithm has less chance of being affected by a fault than a highly used data structure. The results of these focused fault injection experiments are reported in Table III.

TABLE III. NATIVE FOCUSED FAULT INJECTION RESULTS

	LZW		RICE	
	[#]	[%]	[#]	[%]
Silent	33,177	33.18	39,177	39.18
Timeout	1,466	1.47	0	0.00
Wrong Output	42,364	42.36	57,419	57.42
Detected	22,994	22.99	3,404	3.40
TOTAL	100,000		100,000	

As the reader can observe, the number of faults classified as Silent for the RICE algorithm does not change significantly with respect to the random fault generation. Conversely, the number of Silent-classified faults in the LZW case changes significantly. This is motivated by the fact that only actually used data structures are targeted during fault injection. Table III also highlight the intrinsic robustness of the LZW algorithm. Indeed, as the focused fault injection results suggest, a significant amount of the injected faults that produce visible effects can be detected as memory access violations, therefore designers can detect them easily using a memory protection unit/memory management unit available in any modern microcontroller or microprocessor. Conversely, faults affecting RICE are seldom detected through built-in mechanisms. As a consequence, in case of RICE error detection is completely under the responsibility of the application designer, while in case of LZW a significant portion of fault can be detected leveraging the built-in mechanisms the selected processor offers off-the-shelf.

To characterize the considered algorithms, we performed a further set of experiments by exploiting the ISS-based run-time environment. The results we gathered by injecting 100,000 randomly selected faults in the OpenRISC 1200 register file (registers R1 to R31) are reported in Table IV.

TABLE IV. ISS-BASED RANDOM FAULT INJECTION RESULTS

	LZW		RICE	
	[#]	[%]	[#]	[%]
Silent	79,661	79.66	51,410	51.41
Timeout	1,811	1.81	6,755	6.76
Wrong Output	6,078	6.08	24,866	24.87
Detected	12,450	12.45	16,969	16.97
TOTAL	100,000		100,000	

If we compared Table II and Table IV, where in both cases randomly generated faults are considered with uniform probability for all the data structures/registers, we can see that:

- The percentage of faults classified as Silent is consistent between native and ISS-based run-time environment. Although the absolute numbers are different, we can see that the trend observed in the native environment are the same in the ISS-based run-time, that is, LZW exhibits a higher number of Silent than RICE. The same observation holds true for the faults classified as Wrong Outputs and Detected.
- The number of Timeout recorded during ISS-based fault injection does not respect the trend observed during native fault injection. This can be motivated by that fact that some of the registers attacked in ISS-based fault injection may have impact on the execution flow, being R1 the stack pointer and R9 the link register. As a result, when injecting in the register file it is more likely to alter severely the execution flow, resulting in Timeout, than during native injection. Indeed, if we remove from the fault list the faults affecting R1 and R9, the faults leading to Timeout during ISS-based fault injection dramatically reduce, making the overall results in line with what observed during native injection.

Table V details the faults classified as Detected during ISS-based fault injection.

TABLE V. FAULTS CLASSIFIED AS DETECTED USING ISS-BASED FAULT INJECTION

	LZW	RICE
	[%]	[%]
Bus error	84.61	88.57
Alignment error	3.88	5.42
Illegal instruction	2.50	0.23
Other error	9.01	5.78

Because the LZW algorithm uses a larger portion of the data memory than RICE, the number of invalid addresses/misaligned addresses generated by faults is lower than in

the RICE case. Moreover, it is more likely that a fault jumps program execution into a portion of the data memory, producing Illegal instructions, in the LZW case than in the RICE case.
Results concerning HDL injection will be provided in the final version of the report.

4 Towards radiation environment-aware fault injection

To further refine the fault injection representativeness, and to estimate the protection capabilities of gate-level circuit techniques investigated during the MaMMoTH-Up project, a gate-level run-time simulation-based environment has been developed that exploits simulations with back annotated timing information from a 90 nm standard cell library [36]. The run-time environment is extended with a radiation source model, described in this section, to simulate different radiation environments for the device under test. This allows designers to compare the behaviour of protected and unprotected circuits exposed to a certain radiation environment, guiding them in system-level reasoning about applicable protection techniques.

4.1 Considered faults and errors

Radiation effects on circuits are diverse, ranging from transient influences to permanent damage. For example, if high-energy neutrons strike silicon, the ion products of the inelastic scattering transfer energy into the material.
For each 3.6 ± 0.3 eV transferred into silicon an electron hole pair is produced [37], which are rapidly collected at the depletion region of reverse-biased junctions, creating a large current transient at that node [38].
If such single event transients (SETs) happen at or reach the feedback node of a latch, the state of the latch might be corrupted: Single event upsets (SEUs) happen.
Multiple bit upsets (MBUs) describe a state corruption of more than one latch logically related to each other.
As long as the state corruption is non-destructive for a device, soft errors are correctable by performing one or more normal device operations [39].

4.2 Radiation source model for gate-level simulations

Interactions of radiated particles and the circuit of interest are physical in nature. During gate-level simulation physical quantities are abstracted away to a high degree: Only cell timings, connection, and logical information remain. Furthermore, the simulation time T_S is discretized resulting in uniform time steps t_Δ.Our radiation source model uses the remaining information and connects them to radiation related quantities.
A particle flux [40] $f = n/t_\Delta$ with n radiated particles per time step t_Δ is picked up in the particle fluence rate $F = f/A_T$, with the total design area A_T, to derive the particle fluence [40] Φ:

$$\Phi = \int_0^{T_S} F \, \mathrm{d}t = \frac{1}{A_T}\int_0^{T_S} \frac{\mathrm{n}}{t_\Delta} \, \mathrm{d}t = \frac{n\, T_S}{t_\Delta A_T}$$

During simulation, the number of SEUs e is counted to calculate the SEU cross section $\sigma = e/\Phi$.
Armstrong reports measured and calculated 1 to 10 MeV neutron fluence rates of 0.26 $\mathrm{cm}^{-2}\mathrm{s}^{-1}$ [41] at the top of the atmosphere for a geomagnetic latitude of 42°N. Ground-level measurements in New York reveal a total neutron fluence rate of 0.0134

$\mathrm{cm}^{-2}\mathrm{s}^{-1}$ [42]. These fluence rates F are inserted into the particle fluence equation to obtain the total amount of radiated particles $N = F\,T_S A_T$ for the radiation source model representing the intended environment during the simulation.
The radiation source is written as a script that traverses the circuit to find all nets in the fan-in of latches. For a requested number of particles within the simulation time a random signal level (0 or 1) is placed on uniformly chosen nets, simulating an SET.
All simulated SETs are distributed uniformly in time as well, resulting in different combinations of SETs, SEUs, and MBU.
During reset of the circuit the radiation source is disabled. Once the circuit is in a known state every part of the circuit under test is exercised to maximize the impact of each fault on the observable outputs. All signal switching activities are recorded and are used for power estimation with information from the standard cell library using industry grade tools.
The simulations are repeated several times for each circuit to gather a dataset for all chosen numbers of radiated particles that cause a single event effect. Due to the non-determinism introduced by the radiation source, statistical analysis of the data is necessary.
The proposed fault injection environment enhances gate-level simulations with the possibility to recreate a specific radiation environment. This allows designers to compare the behaviour of protected and unprotected circuits exposed to a certain radiation environment, guiding them in system-level reasoning about applicable protection techniques.

5 Self-test for the selected case study

In the confidential deliverable D3.2 “Prototype of Monitoring Framework with Self-Test Extensions” we described the role that SBST may play in the identification of possible faults affecting the processor in a safety-critical system and reported some results related to test program addressing the OR1200 processor core.

In this Section we focus on the further issue of compacting these test programs, so that their execution time can be reduced, while still maintaining the same Fault Coverage.

In functional test only the functional input signals of the unit under test are stimulated and only its functional output signals are observed. A common approach to functional test for processor-based systems is *Software-Based Self-Test* (*SBST*) [3], which consists of forcing the CPU to execute a test program and checking the produced results.

SBST is frequently used to complement other kinds of test as it has important properties:

- it allows testing the system at-speed, since the test program is executed at the same frequency of application programs;
- it does not require costly, high-speed testers, since low frequency interfaces can be used to upload the test program into a memory accessible by the processor and to retrieve the results at the end of the test;
- it implicitly tests both the modules composing a system (such as processor, bus, memories, peripherals) and their interconnections;
- finally, being based on executing a piece of code, the test can be more easily tweaked to tackle different types of defects, to match new constraints (e.g., related to power, as in [35]), and to provide better diagnostic information [6].

Functional test based on SBST is widely adopted for the test of single devices such as Systems on a Chip (SoCs), boards, and complete systems [4]; its adoption spans from end-of-manufacturing to incoming inspection (also known as qualification test) and in-field testing. In the last case, the role of functional test is particularly relevant in safety-critical applications, where standards and regulations specify the target fault coverage to be achieved, and both minimum cost and minimal invasiveness on the application environment are highly desirable [32]. The constraints mandated by the ISO26262 for automotive applications are a paradigmatic example.

A major limitation of SBST lies in the cost for the development of suitable test programs. Although the first solutions for creating functional test programs were proposed more than three decades ago [1] [2], only recently research led to techniques that allow test engineers to reliably devise test programs achieving good fault coverage for processors and controllers [32].

Recently, automatic approaches for generating a test program have been proposed [31][5], at least for small- and medium-size microprocessors. However, the typical approaches still rely on manual effort and pseudo random generators [28], and in both cases test program size and duration are usually far from being optimal [7].

This work tackles test compaction, an important activity as in many different contexts the duration of the test is a critical parameter. For instance, when a test program is run in the field, it often exploits the time slots left idle by the main application [33], and a long duration is likely to impair its applicability. Similarly, when a test program is part of the end-of-manufacturing test process, its duration directly impacts the cost of the test process, and any reduction immediately turns into a money saving.

Test compaction techniques can be classified either as “dynamic” or “static”. The former are performed directly during test generation, while the latter are applied on an existing test set in a separate phase. The two types of approaches are not necessarily exclusive, and some authors proposed to mix them [18].

Dynamic test compaction is known to be able to reduce both the size and the duration of a test program. However, it may significantly increase the complexity of the test generation process, and, more importantly, it cannot be used when the test set is already available. The possibility to reuse, with possible minor modification, existing sets of test programs is a key advantage: test programs need to be optimized after they have been generated.

A basic idea in static test compaction approaches is to pinpoint and remove the parts of a test that are not strictly necessary for achieving the target fault coverage. In the context of test program compaction, it translates into finding and omitting all the instructions that do not contribute to the target fault coverage.

Any test program can be translated into a sequence of binary stimuli, and such a sequence can then be compacted resorting to a variety of methods [8]-[21]. However, the resulting binary sequence may not correspond to a valid sequence of instructions any more, and most of the advantages of SBST would be lost. Consequently, any test program compaction methodology is required to operate at the level of assembly program, facing some specific problems and challenges that are not present in binary test sequence compaction:

- test programs are commonly executed on systems that include memories; hence, compaction should take care of a more complex scenario when dealing with a test program.
- test programs may include control-flow instructions that may force the execution of the same block of instructions more than once, or the skip of a block. Differently from binary test sequences for generic sequential circuits, in SBST the test duration is not necessarily proportional to the size of the test program: although instruction removal tends to decrease test duration, too, a reduction in size does not necessarily turn into a corresponding reduction in test duration.
- the removal of an instruction may trigger special events (e.g., exceptions, infinite loops) which in some cases hang the whole system, or force it into hard-to-manage situations.

At present, there are few works on test program compaction in the literature. In [7] a method was proposed, based on extracting from a test program a set of independent fragments (called spores); an optimization algorithm can then select the minimum sequence of spores whose execution allows achieving a given fault coverage. The method is effective on blocks like the arithmetic unit, but can only be applied under strict constraints and requires an ad-hoc simulator to perform the analysis. The method proposed in [34] deals with the special case in which multiple test programs are available, and we want to select the subset that minimizes the duration, while keeping the same global fault coverage.

This work is amongst the first proposing a fully automatic procedure for removing instructions from an existing test program to reduce its duration without reducing its fault coverage. The scenario we consider is very common in practice: we do not assume the knowledge of any specific information about the test program itself, and

we simply aim at compacting it while preserving the initial fault coverage with respect to the given fault model.
We considered different solutions. As a baseline we devised a simple, exhaustive local-search: we remove one instruction at a time from the test program, checking (via fault simulation) whether the resulting fault coverage remains the same [9]. This approach, although sometimes very effective, requires high computational effort. We also propose more complex solutions based on instruction restoration [11]: a group of instructions is initially removed from the test program; omitted instructions are then restored one by one until the initial fault coverage is achieved. In this way we can reduce the required computational effort, while still achieving significant compaction figures.
Experimental results are reported using the OpenRISC processor as a test case, and considering several test programs addressing faults in specific modules within the processor itself. Although our experiments were targeted to single stuck-at faults, the proposed approach is independent of the adopted fault model.

5.1 Background

Extensive work has been performed on the compaction of binary test sequences.
The first work to show that it is possible to omit a test vector from a given sequence without losing fault coverage is the one reported in [9]. The possibility of omitting test vectors exists even if test generation is carried out with dynamic test compaction. It results from the fact that the test generation procedure cannot avoid the inclusion of unnecessary test vectors in the sequence. Various implementations of test vector omission were described in [9]-[17].
In general, if the test vector at clock cycle u is omitted from a test sequence T, every fault that is detected by T at clock cycle u or higher may lose its detection. The procedure described in [9] simulates these faults in order to determine whether or not the test vector at clock cycle u can be omitted. To reduce the computational effort, if the test vector at clock cycle u can be omitted, the procedure applies binary search to compute the longest subsequence that can be omitted starting at clock cycle u.
The restoration-based procedure described in [11] is a more efficient variation of test vector omission. This procedure initially omits test vectors from the sequence without considering the fault coverage. It then restores test vectors into the sequence in order to restore the fault coverage. Test vectors are restored by considering one fault at a time. This contributes to the efficiency of the procedure. In the procedure described in [14], parallel simulation is used to accomplish the restoration process with a computational effort that is equivalent to that of fault simulation of the sequence.
The procedure described in [15] modifies the test vectors in a sequence. The procedure described in [16] allows omitted test vectors to be reintroduced into the sequence. The goal of both procedures is to achieve test compaction beyond that achievable with the basic test vector omission or restoration procedures.
A higher level, assertion-based dynamic test compaction procedure was described in [19]. The compaction of a set of binary test sequences was considered in [8] and [20]. In [20], subsequences of the test sequences in the set are merged to form a single compact test sequence. A reduction in the storage requirements of a binary test sequence is targeted in [21]. The test sequence is compacted under the assumption that several different test sequences will be applied based on a single stored test sequence.

The first results with an approach similar to the one presented here were reported in [27]. However, in [27] we targeted a reduction in the size of the program code size, while here the focus is on the execution time. This goal is more important in real applications, since it directly affects the test cost and feasibility (for in-field test), and makes the addressed problem much harder to solve. Moreover, this work considers test programs addressing a whole processor, while in [27] we focused on test programs aimed at testing faults belonging to two modules, only.

5.2 Proposed method

This work describes two approaches (denoted as A0 and A1) to compact SBST test programs. To evaluate the two approaches, we consider both their computational requirements and their compaction performance. Although the target of the test programs considered in this paper is the set of faults within a processor, the approaches can be generalized to test programs targeting other modules (e.g., peripheral components) within a processor-based system.

We assume that the test program T is already available. T is a sequence of N instructions $T=(I_0,I_1,\ldots,I_{N-1})$ executed starting from a completely specified state, i.e., a state where all memory elements, such as RAM cells and flip-flops, have known values. T achieves a fault coverage FC with respect to a given set of faults F. In the experiments reported in this paper we consider single stuck-at faults, but the proposed approaches can be applied to different fault models.

In the experiments, we mark a fault as detected when a difference with respect to the fault-free system is observed on any output signal. This assumption is reasonable when end-of-manufacturing test is considered, but other observation points can be considered as well, such as user registers, performance counters, or cache lines, possibly compacted into one or more signatures.

The goal of the methodology is to find a new test program $T'=(I_0',I_1',\ldots,I_{M-1}')$ composed of a subset of the instructions of T, in the same relative order, that achieves at least the same fault coverage when started from the same initial state, and minimizes the execution time. Being composed of a subset of the original instructions, as a side effect, the methodology will also reduce the size of the test program.

We assume that T is a program suitable for being used in a SBST scenario (e.g., it triggers no exception), and we initially neglect additional constraints that could possibly exist on the behavior of the processor during the execution of the test. Nevertheless, the removal of instructions may yield invalid test programs: the most common causes are exceptions (such as division by zero), and infinite loops. As the effect of the removal of a single instruction cannot be foreseen with a static analysis, we assume to be able to identify all these situations through simulation.

We call *valid* a test program that can be executed without triggering exceptions or violating some constraints imposed by the current scenario, and that terminates correctly without entering an infinite loop.

5.2.1 Compaction by random instruction removal: A0 and A0/2

The most straightforward solution to test program compaction is based on instruction removal. The first algorithm we consider here, called A0, is a greedy local search, and follows the idea proposed in [9] for binary test sequences.

The A0 algorithm is sketched in Fig. 2. In each step, one random instruction $I_i \in$ TP is selected and removed, and the resulting test program TP'=TP\{I_i} is fault simulated. If TP' is no longer a valid test program or the attained fault coverage FC' is less than the original one, or the execution time does not decrease, then I_i' is pushed back into T' and marked so it will never be chosen again. Otherwise, I_i' is removed from T'. The process is then repeated until all instructions have been selected once.

```
1    Fault simulate TP, let F be the set of faults detected by TP
2    For every instruction Ii, selected in a random order {
3        Let TP'= TP\{I_i}
         (i.e., let TP' be the test program obtained by removing Ii from
         TP)
4        Fault simulate TP'
5        If TP' is a valid test program AND all the faults in F are detected
         AND TP' has a shorter execution time than TP then TP = TP'
     } // end for
```
Figure 2. Pseudo-code for the A0 algorithm

The number of steps required by A0 to compact T is O(N). In the generic i-th step (i $\in$ [0,N-1]), the evaluation of the fault coverage involves the simulation of a test program composed of a sequence of N_i to N instructions. Fault simulation itself has at least a polynomial complexity in the size P of the unit under test, and represents the sole computationally demanding step of the algorithm. Hence, the complexity of A0 is $O(N^2 \times P^ß)$, where $P^ß$ represents the computational complexity of fault simulation.
A0 can be iterated by running a second optimization step on the final test program obtained after the first step. When this technique is used, the algorithm is denoted as A0/2. While theoretically possible, additional iterations are avoided because of their computational complexity.

5.2.2 Restoration-based algorithms: A1xx

An advantage in terms of CPU effort can be achieved by a family of algorithms (denoted as A1xx), which are based on first removing a given block of instructions, and then restoring them one at a time until the original fault coverage is obtained.
A1xx algorithms were considered in [27]; however, the target of the optimization was the reduction in the test program size, without considering its duration. The underlying idea is similar to the restoration-based procedure from [11], where that idea was applied to binary sequence compaction. However, in [11], all or most of the test vectors of a binary sequence are initially omitted. In the case of a test program, removing all or most of the instructions may lead to exceptions during the restoration process, and preventing the exceptions may require the restoration of a large number of instructions. This issue, that does not exist with binary sequences, is avoided with test programs by removing small blocks of instructions and restoring instructions from every block immediately after it is removed in order to restore the fault coverage. Moreover, removing one or more instructions from a test program does not necessarily reduce its execution time: hence, at every step the algorithm must check whether the current test program is still valid and still guarantees the same fault coverage and has a shorter execution time.
A pseudo-code of the generic A1xx algorithm is shown in Fig. 3. In the proposed algorithm the original test program is split into m segments (step 2). For every

segment S_i (starting from the last), we first remove all the instructions it is composed of, thus typically reducing the fault coverage. Let Φ_i be the set of faults that are detected by the segments S_i to S_{m-1}, i.e., faults that may become undetected due to the removal. Then, we start restoring one instruction from S_i at a time until all the faults in Φ_i are detected again.
As A0, A1xx algorithms are based on a preliminary fault simulation (step 1), and as many iterations as segments, each corresponding to another fault simulation (step 6). However, the advantage of A1xx algorithms over A0 lies in the fact that the number of fault simulations may become lower than the number of instructions (like in A0), because iterations corresponding to instructions that do not need to be restored (because the fault coverage has already been restored) are not performed. As a consequence, the computational advantage of A1xx algorithms with respect to A0 is greater when the achieved compaction is higher.
Step 4 does not require any computational effort, since the computation of Φ_i (i.e., the set of faults to be considered at each iteration) can be performed once during step 1.

```
1   Fault simulate TP. Let F be the set of faults detected by TP. Mark each
    fault with the instruction that first detects it
2   Split TP into m segments $S_0$ … $S_{m-1}$
3   For every segment $S_i$, starting from the last one {
4       Let $\Phi_i$ be the set of faults which are first detected by the
        instructions in the segments from Si to Sm-1 (included)
5       Let TP' be the test program initially obtained by removing from
        TP all instructions belonging to Si
6       Fault simulate TP'
7       If TP' is a valid test program AND it detects all the faults in $\Phi_i$
        AND has a shorter execution time then
```

```
                    Set TP = TP'
                    goto 3 (next iteration)
8               Select one instruction I from Si
9               TP' = TP' U I (i.e., restore I)
10              goto 6
        } // end for
```

Figure 3. Pseudo-code for the generic A1xx algorithm

Segments are selected starting from the last one in the program. The advantage of this approach is that the fault simulation cost of the first steps is rather low, since they require the fault simulation of faults belonging to the selected segment and the following ones, only. In the following steps, the computational cost typically decreases due to the performed compaction.

Several versions of A1xx are possible, depending on how the segments are defined (step 2), and how the instructions of the generic segment are selected for restoration (step 8).

Concerning the former point, we only consider here the algorithms for which the original test program T is partitioned into m segments, each composed of a fixed number, n, of consecutive instructions (apart from the last). Hence, the first segment S_0 will be composed of the first n instructions in T (I_0 to I_{n-1}), the second segment of the instructions I_n to I_{2n-1}, and so on. Clearly, the choice of the optimum value for n may be relevant. The A1xx variants with segments of variable size are not analyzed here, as they have already been shown to be unlikely to yield satisfactory results [27]. For the same reason, in this work we do not analyze the effect of applying two different restoration techniques in sequence.

Concerning the latter point, i.e., the order according to which instructions belonging to the currently considered segment are restored, the following policies are considered:

- Forward ($A1F_n$): instructions are restored one by one starting from the first in the segment.
- Back ($A1B_n$): instructions are restored starting from the last in the segment.
- Random ($A1R_n$): instructions are restored following a random order.

These three variants, with n ranging from 2 to 10, have been thoroughly evaluated and the results are reported in Section 5.3.

To reduce the computational effort of this approach, only a subset of faults is simulated during each run of the algorithm; in fact, when removing the instructions belonging to a given segment Si, all faults which have been detected by the instructions belonging to segments S0 to Si-1 will still be detected. Hence, the fault simulation of step 6 uses a sub-list of faults, only containing those detected by instructions in segments S_i to S_{m-1}. The identification of the instruction that first detects each fault is done as a byproduct of step 1.

5.3 Experimental Evaluation

The proposed algorithm has been implemented, and we are currently gathering experimental results to be discussed in the final version of the document.

6 Conclusions

Although quantitative evaluations can provide better insight on prospective system architectures, system-level decisions are often taken resorting to the experience of designers due to the lack of adequate (i.e., fast, simple yet accurate) system-level analysis techniques. In this report, we proposed a quantitative system-level analysis of two compression algorithms under investigation for a data logger system to be deployed in a satellite launcher.

Fault injection into the actually used data structures of two compression algorithms is used to identify the intrinsic robustness of the algorithms, as well as to draw system-level considerations about the most suitable hardware architectures to implement each algorithm.

The results gathered while abstracting the hardware implementation of the considered algorithms shall take into account the actual usage of the data structures to better quantify the possible misbehaviors.

Moreover, the results gathered while considering a possible hardware implementation of the algorithms points out that, although hardware specific results are recorded, the general trends observed are consistent with those observed while neglecting the implementation.

The final version of the document will conclude on the experimental results from validation of the proposed algorithm once they are available.

7 References

[1] S. Thatte and J. Abraham, “Test Generation for Microprocessors”, IEEE Transactions on Computers, vol. 29, no. 6, pp. 429–441, June 1980.

[2] D. Brahme, J. Abraham , “Functional testing of microprocessors”, IEEE Transactions on Computers, vol. C-33, no. 6, pp. 475 - 485, June 1984

[3] M. Psarakis, D. Gizopoulos, E. Sanchez, M. Sonza Reorda, “Microprocessor Software-Based Self-Testing”, IEEE Design & Test of Computers, vol. 27, no. 3. May-June 2010, pp. 4-19.

[4] A. Jutman, M. Sonza Reorda, H.-T. Wunderlich, “High Quality System Level Test and Diagnosis”, Proc. of IEEE Asian Test Symposium (ATS), 2014.

[5] A. Riefert, L. Ciganda. M. Sauer, P. Bernardi, M. Sonza Reorda, B. Becker, “An effective approach to automatic functional processor test generation for small-delay faults”, Proc. of the Conf. on Design, Automation and Test in Europe (DATE), 2014.

[6] P. Bernardi, E. Sanchez, M. Schillaci, G. Squillero, M. Sonza Reorda, “An Effective technique for the Automatic Generation of Diagnosis-oriented Programs for Processor Cores”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 27, pp. 570-574, 2008.

[7] E. Sánchez, M. Schillaci, G. Squillero, “Enhanced Test Program Compaction Using Genetic Programming”, Proc. of the 2006 IEEE Congress on Evolutionary Computation (CEC), pp. 865-870, 2006

[8] T. M. Niermann, R. K. Roy, J. H. Patel, J. A. Abraham, "Test compaction for sequential circuits", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, pp. 260 - 267, 1992

[9] I. Pomeranz and S. M. Reddy, "On Static Compaction of Test Sequences for Synchronous Sequential Circuits", in Proc. Design Autom. Conf., 1996, pp. 215-220.

[10] M. S. Hsiao, E. M. Rudnick and J. H. Patel, "Fast Algorithms for Static Compaction of Sequential Circuit Test Vectors", in Proc. of the VLSI Test Symp., 1997, pp. 188-195.

[11] I. Pomeranz and S. M. Reddy, "Vector Restoration Based Static Compaction of Test Sequences for Synchronous Sequential Circuits", in Proc. of IEEE International Conference on Computer Design (ICCD), 1997, pp. 360-365.

[12] M. S. Hsiao and S. T. Chakradhar, "State Relaxation Based Subsequence Removal for Fast Static Compaction in Sequential Circuits", in Proc. Design Automation and Test in Europe (DATE), 1998, pp. 577-582.

[13] S. K. Bommu, S. T. Chakradhar and K. B. Doreswamy, "Static Compaction using Overlapped Restoration and Segment Pruning", in Proc. of International Conference on Computer-Aided Design (ICCAD), 1998, pp. 140-146.

[14] X. Lin, W.-T. Cheng, I. Pomeranz, S. M. Reddy, "SIFAR: Static Test Compaction for Synchronous Sequential Circuits Based on Single Fault Restoration", in Proc. of VLSI Test Symposium (VTS), 2000, pp. 205-212.

[15] I. Pomeranz and S. M. Reddy, "Vector Replacement to Improve Static Test Compaction for Synchronous Sequential Circuits", IEEE Trans. on Computer-Aided Design, Feb. 2001, pp. 336-342.

[16] I. Pomeranz and S. M. Reddy, "Enumeration of Test Sequences in Increasing Chronological Order to Improve the Levels of Compaction Achieved by Vector Omission", IEEE Trans. on Computers, July 2002, pp. 866-872.

[17] I. Pomeranz and S. M. Reddy, "Vector Restoration Based Static Compaction using Random Initial Omission", IEEE Trans. on Computer-Aided Design, Nov. 2004, pp. 1587-1592.

[18] Stelios N. Neophytou, "Test Set Generation with a Large Number of Unspecified Bits Using Static and Dynamic Techniques", IEEE Transactions on Computers (Vol 59, Is 3), 2010, pp. 301-316

[19] J.G. Tong, M. Boulé, Z. Zilic, “Test compaction techniques for assertion-based test generation”, ACM Trans. on Design Automation of Electronic Systems (TODAES), December 2013, pp. 1-29.

[20] I. Pomeranz, “Concatenation of Functional Test Subsequences for Improved Fault Coverage and Reduced Test Length”, IEEE Transactions on Computers (Vol 61, Is 6), 2012, pp. 899-904

[21] I. Pomeranz, “Two-Dimensional Static Test Compaction for Functional Test Sequences”, IEEE Transactions on Computers (Vol 64, Is 10), 2015, pp. 3009-3015

[22] "miniMIPS Overview," opencores.org, 2009. [Online]. Available at http://opencores.org/project,minimips.

[23] "NanGate FreePDK45 Generic Open Cell Library", [Online]. Available at https://www.si2.org/openeda.si2.org/projects/nangatelib.

[24] P. Bernardi, R. Cantoro, L. Ciganda, B. Du, E. Sanchez, M. Sonza Reorda, M. Grosso, O. Ballan, “On the Functional Test of the Register Forwarding and Pipeline Interlocking Unit in Pipelined Processors”, Proc. of 14th International Workshop on Microprocessor Test and Verification (MTV), 2013, pp. 52-57

[25] P. Bernardi, R. Cantoro, L. Ciganda, E. Sanchez, M. Sonza Reorda, S. De Luca, R. Meregalli, A. Sansonetti, “On the in-Field Functional Testing of Decode Units in Pipelined RISC Processors”, Proc. of IEEE Intl. Symposium on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT), 2014, pp. 298-303

[26] E. Sanchez, M. Schillaci, G. Squillero, Evolutionary Optimization: the µGP toolkit, Springer, 2011

[27] M. Gaudesi, M. Sonza Reorda, I. Pomeranz, “On Test Program Compaction”, Proc. of IEEE European Test Symposium (ETS), 2015.

[28] P. Parvathala, K. Maneparambil, W. Lindsay, “FRITS - a microprocessor functional BIST method”, Proc. of IEEE International Test Conference (ITC), 2002, pp. 590-598

[29] D. Sabena, M. Sonza Reorda, L. Sterpone, “A new SBST algorithm for testing the register file of VLIW processors”, Proc. of Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012, pp. 412 – 41

[30] D. Gizopoulos, A. Pachalis, Y. Zorian, M. Psarakis, “An effective BIST scheme for arithmetic logic units”, Proc. of IEEE International Test Conference (ITC), 1997, pp. 868 – 877

[31] S. Gurumurthy, S. Vasudevan, J. A. Abraham, “Automatic generation of instruction sequences targeting hard-to-detect structural faults in a processor”, Proc. IEEE International Test Conference (ITC), 2006

[32] P. Bernardi, R. Cantoro, S. De Luca, E. Sánchez, A. Sansonetti, “Development Flow for On-Line Core Self-Test of Automotive Microcontrollers”, IEEE Transactions on Computers, 2016, Volume: 65, Issue: 3, pp. 744 – 754

[33] A. Merentitis. G. Theodorou, M. Giorgaras, N. Kranitis, “Directed Random SBST Generation for On-Line Testing of Pipelined Processors”, Proc. of the 2008 14th IEEE International On-Line Testing Symposium, pp. 273 – 279

[34] A. Touati, A. Bosio, P. Girard, A. Virazel, P. Bernardi, M. Sonza Reorda, “An effective approach for functional test programs compaction”, Proceedings of the 2016 IEEE 19th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)

[35] Jun Zhou, H. -J. Wunderlich, “Software-Based Self-Test of Processors under Power Constraints”, Proceedings of the Design Automation & Test in Europe Conference, 2006

[36] *TCBN90LPHP TSMC 90nm Core Library Databook.* Ver. 150a. Taiwan Semiconductor Manufacturing Company, Ltd. Dec. 2005.

[37] K. G. McKay and K. B. McAfee. “Electron Multiplication in Silicon and Germanium”. In: Physical Review 91.5 (Sept. 1953).

[38] Robert C. Baumann. “Radiation-Induced Soft Errors in Advanced Semiconductor Technologies”. In: IEEE Trans. Device Mater. Rel. 5.3 (Sept. 2005), pp. 305-316.

[39] *Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices.* JEDEC Standard JESD89A. Arlington, VA, USA: JEDEC Solid State Technology Association, Oct. 2006.

[40] S. M. Seltzer et al. “ICRU Report No. 85: Fundamental Quantities and Units for Ionizing Radiation”. In: Journal of the ICRU 11.1 (Oct. 2011). Revised.

[41] T. W. Armstrong, K. W. Chandler, and J. Barish. ”Calculations of neutron flux spectra induced in the earth's atmosphere by galactic cosmic rays”. In: Journal of geophysical research 78.16 (1973), pp. 2715-2726.

[42] M. S. Gordon et al. “Measurement of the Flux and Energy Spectrum of Cosmic-Ray Induced Neutrons on the Ground”. In: IEEE Trans. Nucl. Sci. 51.6 (Dec. 2004), pp. 3427-3434.