Observability solutions for in-field functional test of processor-based systems: a survey and quantitative test case evaluation

J. Perez Acle(1), R. Cantoro(2), E. Sanchez(2), M. Sonza Reorda(2), G. Squillero(2)

(1) Universidad de la República, Montevideo, Uruguay

(2) Dipartimento di Automatica e Informatica, Politecnico di Torino, Torino, Italy

julio@fing.edu.uy, {riccardo.cantoro, ernesto.sanchez, matteo.sonzareorda, giovanni.squillero}@polito.it

Abstract— The usage of electronic systems in safety-critical applications requires mechanisms for the early detection of faults affecting the hardware while the system is in the field. When the system includes a processor, one approach is to make use of functional test programs that are run by the processor itself. Such programs exercise the different parts of the system, and eventually expose the difference between a fully functional system and a faulty one. Their effectiveness depends, among other factors, on the mechanism adopted to observe the behavior of the system, which in turn is deeply affected by the constraints imposed by the application environment. This paper describes different mechanisms for supporting the observation of fault effects during such in-field functional test, and it reports and discusses the results of an experimental analysis performed on some representative case studies, which allow drawing some general conclusions. The gathered results allow the quantitative evaluation of the drop in fault coverage coming from the adoption of the alternative approaches with respect to the ideal case in which all the outputs can be continuously monitored, which is the typical scenario for test generation. The reader can thus better evaluate the advantages and disadvantages provided by each approach. As a major contribution, the paper shows that in the worst case the drop can be significant, while it can be minimized (without introducing any significant extra cost in terms of test generation and duration) through the adoption of a suitable observation mechanism, e.g., using Performance Counters possibly existing in the system. Suitable techniques to implement fault simulation campaigns to assess the effectiveness of different observation mechanisms are also described.

1 Introduction

In several domains (e.g., automotive, biomedical, space and aircraft industries) electronic systems are commonly used in mission- and safety-critical applications. In these domains, a misbehavior due to a defect affecting the hardware may have catastrophic effects, including hurting humans and provoking huge economical losses. Hence, there is a strong push to devise techniques able to minimize the probability that a misbehavior caused by a defect arises, and to suitably handle it in case it manifests itself anyway. When considering the latter point, different solutions have been proposed, and the best solution depends on the specific constraints of each scenario. Standards and regulations (e.g., IEC 61508 for generic safety-related industrial systems, ISO 26262 for automotive applications, RTCA/DO-254 for avionics) also play a significant role, forcing companies to devise and adopt solutions able to achieve some predefined target in terms of dependability.

Most of the electronic systems involved in safety-critical applications include a microprocessor or microcontroller. For these systems, it is possible to force programmable units to run test programs able to reveal the presence of defects by activating them and propagating their effects up to an observable location (e.g., a special memory area). Eventually, the application may trigger suitable actions to prevent catastrophic consequences, such as turning the system to a safe status, or reconfiguring it so that the faulty module is not used any more. To minimize the impact on the system, these test programs are often limited to use the time periods left idle by the core applications, or run during the start-up/power-off phases. Such an approach is referred to as *Software-Based Self-Test* (SBST) [1], and generically labeled as "functional" as it relies directly on the normal functions of the system. SBST does not require any specific Design-for-Testability (DfT) structure, although it may exploit available hardware features, and can be used to test any processor-based system, no matter whether it is a System-on-Chip (SoC) or a board. As a major advantage, test based on SBST can be run at the processor operational speed, thus allowing the detection of defects which are only activated at the maximum frequency. For this reason, it is often used during the manufacturing test phase as a supplement to other techniques to increase the final defect coverage.

SBST is currently adopted in quite different test scenarios, including both end-of-manufacturing test and in-field test. When applied for end-of-manufacturing test, either the automatic test equipment (ATE) drives the processor inputs while it executes the program and observes the outputs, or it loads a program into the cache of the processor, forces it to execute at full speed, and eventually extracts some test syndrome from a special (hidden) register. On the other side, when in-field SBST is considered, a common solution lies in storing the test program in a flash memory, activating its execution when required, and finally checking the content of some selected memory variables, where the test program stores its results.

When comparing the SBST solutions adopted for end-of-manufacturing test with those for in-field test, a major difference is that the former can, in some cases, benefit from full accessibility to the input and output signals of each device (such a test scenario is called "open loop test" in [32]). On the contrary, in solutions oriented to in-field testing, the tester cannot be used and existing DfT structures are in most of the cases not available (e.g., because they have been destroyed or made inaccessible to better protect the system security, or because they are not documented by the device providers). Hence, the only feasible solution for the system company in charge of developing the in-field test is to adopt a purely functional approach, i.e., without resorting to any DfT feature. Additionally, in-field constraints may be quite severe: for example, the memory area usable by the test could be limited to a specific size and location, and some faults may become functionally untestable [9] (i.e., no test stimuli exist for them under the in-field test scenario). Although untestable faults by definition cannot affect the system behavior, they may significantly limit the fault coverage that can be achieved, even using a high-quality test program. Hence, it is desirable to be able to identify untestable faults.

Concerning observability, some solutions adopted for end-of-manufacturing test may allow the continuous monitoring of all the output signals of the device under test by the ATE. On the contrary, with in-field SBST the ATE cannot be used, and thus the effects of faults are typically observed by checking, at the end of the test program execution, the values left by the program in some specified memory locations. This limited observability may significantly reduce the achievable fault coverage; some specific fault categories are known to be untestable if fault detection is only based on looking at the final memory content. In particular, faults that only affect the time behavior of the processor (e.g., by delaying some operation) found in modules such as Cache Controllers [8] and Branch Prediction Units [7] cannot be detected in this way. The test of these *performance faults* [3] can be successfully faced by resorting to the so-called performance counters existing in most of the current microprocessors and microcontrollers [4]. Alternatively, one can resort to special hardware modules that can be added to a processor, able to monitor the bus during the execution of a test program and then compute a signature. As a result, re-using any test programs developed for high-observability end-of-manufacturing SBST for in-field SBST may be either very expensive, or result in a significant drop of the achieved fault coverage. Recently, some papers specifically focused on the generation of test programs for in-field SBST [16].

The main purpose of this paper is to survey the different solutions that can be adopted in practice to support the observation of fault effects when SBST is adopted for in-field test, discussing the advantages and limitations of each of them. Secondly, the paper uses two test cases to quantitatively evaluate the benefits and cost of each observability solution: one targets the branch prediction unit (BPU) in a MIPS-like processor based system, and the other targets the cache controller logic in a dual-core LEON3 system. This paper is the first to report extensive experimental results to compare the fault coverage that can be achieved with the different solutions, thus allowing the reader to have a better understanding of the advantages and disadvantages provided by the different solutions[1]. Finally, the paper outlines some techniques to compute fault coverage figures related to the usage of an SBST approach with different observation mechanisms.

The paper is organized as follows: Section II provides an overview of the state-of-the-art in the area of SBST, with special emphasis on in-field SBST. Section III describes the different observability solutions we considered in this paper. Section IV describes the experiments we performed on the two sample systems to quantify the effects stemming from the adoption of the different observability solutions. Finally, Section V draws some conclusions.

2 Background on Software-Based Self-Test

The term *Software-Based Self-Test* (SBST) was first proposed by Chen and Dey in [14], but the approach itself has been proposed few years before under the name "*Native Mode Functional Test*" in [32] and [33]. SBST broadly identifies all test methodologies based on forcing a microprocessor/microcontroller to execute a program and checking the results to detect the presence of possible defects affecting the hardware. Indeed, the pioneering idea of testing a microprocessor with a program dates back to 1980. In [10], Thatte and Abraham devised fault models and procedures for building test programs able to detect permanent defects in different functional units of a simple processor. A wide adoption of their methodology was hindered by the difficulties in automating the generation of such test programs, especially when targeting complex processors.

In general, the usage of SBST requires:

1. Generating a suitable test program. This is typically a hard job, which is still mainly performed by hand. Moreover, the complexity and effectiveness of this task depends on the adopted metric, which in turns depends on the available

[1] A preliminary version of this work was presented in [31]. In the current version we significantly extended (among the other things) the Experimental Results section by adding a further test case and improving the results analysis.

information: in some cases, both RTL and gate-level models of the target system are available, while in others functional information is available, only. For the purpose of this paper, we assume that the gate-level netlist is available, and it is possible to compute the fault coverage achieved by the generated test program with respect to the most common structural fault models (e.g., stuck-at).

2. Creating an environment to support its execution. Once the test program is available, it must be stored in some memory accessible by the processor, the processor must be triggered to execute it at the due time, and the results produced by the processor during its execution must be observed. In this paper we specifically focus on the last issue.

Nowadays, the complexity of processors has significantly increased; the micro-architectural details play a fundamental role, and devices cannot be accurately modeled using information about the Instruction Set Architecture (ISA) alone. However, SBST is getting more and more important: it commonly supplements other kinds of tests, as functional programs may detect unmodeled defects that escape traditional structural tests (the so-called "collateral coverage" [11]). By definition, the functional approach tests the system in its operational mode, without activating a test mode and without reconfiguring the system; hence, it is guaranteed not to cause *overtesting* (or *overkilling*). Moreover, several producers exploit functional stimuli to validate their design or to run post-silicon verification.

In some cases, test programs are generated pseudo-randomly [28], possibly using simulation feedback, and may even use some hardware support to make the test phase more efficient [12].

Amongst the several recent works focusing on SBST, some aim at developing algorithms to generate effective test programs for common modules starting from the knowledge of its ISA alone (e.g., for an OpenSPARC T1 processor [17] or a MIPS-like ISA [7]), eventually combined with RTL description (e.g., [13] for two different implementations of the MIPS ISA). Others focus on the possible automation of the test program generation procedure (e.g., [15] using a LEON2 processor, [16] using miniMIPS). The possible usage of SBST for diagnostic purposes has also been explored (e.g., [18] using an 8-bit accumulator-based microprocessor, and [19] for transition-delay faults in an i8051-compatible microcontroller). Finally, a number of works study how to apply SBST for in-field test (e.g., [20] for a MIPS architecture processor).

In this last domain, regulations and standards mandate the adoption of effective solutions to early detect permanent faults, and SBST has the big advantage that it does not require access to any systematic DfT solution (such as scan or BIST), whose usage details are often considered as proprietary by the manufacturer. SBST can be used not only to test the CPU, but also the other components in a microcontroller or SoC: for example, several works addressed its adoption for the test of peripherals [38], memories [39] (possibly implementing transparent test [40]) and cache memories [36][37]. SBST usage can also be extended to the test of multi-core systems [35]. In some cases, the usage of existing hardware resources introduced for non-test-related purposes (e.g., for debug, design validation, performance assessment) allows significantly reducing the size and duration of SBST test programs [34].

Moreover, SBST can more easily match the constraints of the environment where the processor is employed. When adopted for in-field test, SBST typically means activating a test program either at the system power-on, or during the application idle times. In the latter case, additional constraints about the duration of the test exist, due to the limited duration of the available time slots. Unfortunately, the constraints posed by the application environment may severely impair the effectiveness of the method when applied to test a processor. When functional test is used for end-of-manufacturing test, processor inputs and outputs can often be fully controlled and observed by an ATE. Nevertheless, during in-field test some parts of the processor (e.g., the test and debug structures) might not be accessible by the test procedure, thus resulting in untestable faults [9], i.e., faults for which no input stimuli exists, able to detect them. In other words, some parts of a processor which are not used anymore during the operational phase may contain faults, which cannot be tested in this phase, but are also guaranteed not to affect the system behavior. Besides, not all the processor inputs may be freely controllable in the in-field scenario: for example, activating the reset signal is hardly possible, thus preventing the test of the reset logic. More in general, possible Control/Status input signals coming from other devices may be hard to control [33]. Finally, observability during SBST in-field test could be limited, since only the produced results (e.g., in memory) can be observed. The set of faults which cannot be tested in the in-field environment due to these additional constraints are known as *functionally untestable faults* [9]. As previously mentioned, it is important to be able to identify untestable faults, since they limit the achievable fault coverage.

This paper focuses on the latter issue, related to observability: firstly, it describes in detail different observability solutions that can be adopted for SBST in-field test, comparing them with the ideal case in which all the outputs of the module under test (or of the processor) are continuously observable during the test. Secondly, it uses two test cases to quantitatively evaluate the loss in fault coverage that stems from the adoption of the different solutions.

3 Observability Solutions

In this Section, the main solutions that can be adopted to observe the effects of possible faults during in-field SBST testing of a processor-based system are described, namely: module-level, processor-level, system bus, memory content, and performance counters. The above list also includes a few solutions that only represent references, although they can hardly be adopted during in-field SBST. For the purpose of this work we limited ourselves to bare metal systems, i.e., we didn't consider solutions that would require the presence of an Operating System (e.g., based on monitoring its performance, or analyzing the event logs).

We assume that the targeted faults are those inside a given module within the processor. For every solution, the adopted mechanism as well as the main advantages and disadvantages are detailed, and a preliminary analysis about the forecasted coverage is reported. Fig. 1 summarizes the considered observability solutions, referring to the architecture of a simple processor-based system.

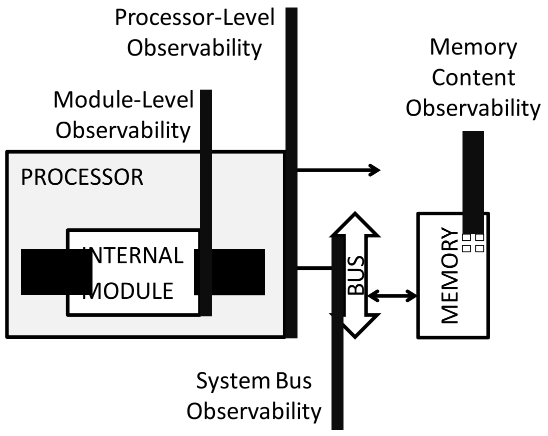
Processor-Level Observability
Module-Level Observability
Memory Content Observability
PROCESSOR
INTERNAL MODULE
BUS
MEMORY
System Bus Observability

Fig. 1 Generic system under test: the observation points adopted by the techniques described in the text are highlighted

3.1 Module-Level Observability

When a generic module inside the processor is considered for the test, the *ideal* level of observability is the boundary of such a module, i.e., it is assumed that all the output ports are available for observation.

The test program is supposed to be able to properly stimulate the input ports of the considered module, to excite the faults and then to propagate them towards the module output ports, which are test observation points.

This observability solution can be adopted during simulation and fault simulation processes. However, when working on real chips, for a number of reasons it is hardly feasible neither in-field nor, in most cases, at end-of-manufacturing. These reasons are, firstly, that the module output ports usually do not coincide with the circuit pinout, and even if they do, it is normally not affordable to continuously observe the circuit behavior without resorting to additional hardware; secondly, when an instrument is attached to the observation points, the observed signals can only be read in test mode through a dedicated tester.

Therefore, this solution is introduced here only as a reference, because it establishes an upper bound to the fault coverage figure obtainable through SBST test approaches.

3.2 Processor-Level Observability

This solution assumes that fault effects can be observed at the processor level, i.e., that all the processor outputs can be continuously monitored. While module-level observability is very specific and may be not feasible in practice, observability at processor-level represents one of the scenarios that are sometimes adopted for end-of-manufacturing test. Considering an internal module which has to be tested, the test program must not only propagate the fault effects up to the module output ports, but must be able to propagate them also to the processor output ports.

According to these considerations, the observability we can get with this solution is generally lower than the one obtained at module-level. In most of the cases, propagating the faulty behavior requires an additional effort in order to reach the processor outputs. In the case of a functional testing approach based on test programs, this additional effort may imply the addition in the code of specific instructions able to propagate the fault effects to the processor outputs. As an example, faults within an arithmetic unit can be easily activated by executing suitable arithmetic instructions (thus propagating their effects on the module outputs), and can then be made observable on the processor outputs via store instructions that propagate the result of the arithmetic operation up to the processor output ports.

Faults may also exist that, even with the addition of instructions, cannot be observed on the processor outputs. This situation may happen when the processor design includes some redundant circuitry, for example left from previous releases or included for future extensions of the design. Clearly, the related faults can be classified as untestable. However, the identification of untestable faults may often represent a relevant problem.

Due to the need of constant monitoring of all the processor outputs, this observability solution requires the use of an ATE and thus, cannot be adopted by in-field SBST.

3.3 System Bus Observability

This solution mandates that the control, data and address signals of the system bus are continuously monitored. When comparing this solution with the previous, all the processor outputs not related with the system bus are excluded from observation.

End-of-manufacturing scenarios may offer a high level of observability when the constant monitoring of the output ports of the processor is possible. Such a powerful scenario is not representative of an observation mechanism for in-field testing. However, more and more processors (especially for embedded systems) are equipped with specific components in charge of monitoring the interconnections between the processor and the memory subsystem, in some cases including external caches. Examples of such modules are MISRs attached to the bus, which update a signature every time new data are going to be written to the memory. This solution has been adopted by commercial microcontrollers, e.g., from Freescale [21]. In other cases, dedicated programmable embedded cores are in charge of tracing specific bus transactions (e.g., ARM [22]) and of storing a history of processor execution in a local memory, which is accessible through a dedicated port (e.g., for debug purposes). An example of IP core specially devised for SoC testing is described and demonstrated for a processor compliant with the SPARC v8 architecture in [23]. The presence of caches significantly limits the amount of data flowing through the bus, and hence the number of faults whose effects can be observed by observing it.

SBST programs using this observability solution should include specific sequences of instructions that permit the propagation of the fault effects up to the system bus.

As an example, let us consider the faults affecting the circuitry that supports an external coprocessor. If the external coprocessor is connected to the processor with dedicated ports, the effects of such faults –propagated up to this interface in the original program– need to be read back from the coprocessor and stored to the system memory in order to become observable in the system bus. This solution adds complexity to the test program and is not always feasible. In case of faults whose effects can only be observed on non-functional output signals and never read back, the fault coverage reduction cannot be recovered, thus resulting in a potentially less-effective observation mechanism in general.

Also, if an SBST program developed for processor-level observability is evaluated resorting to this observability solution, a significant fault coverage reduction could be observed. This fault coverage loss is motivated by the reduction of the observed signals, as they are a subset of the output signals of the processor.

3.4 Memory Content Observability

According to this solution, a fault is marked as detected if the content of the system memory is different than the expected one at the end of the execution of the SBST program.

All the previously presented observation mechanisms rely on the fact that some output ports of the circuit can be constantly monitored, e.g., by a dedicated tester which is physically connected to test points or to the interface with on-board instruments. This is not the usual case of SBST in general. In a manufacturing at-speed SBST scenario, the functional program is often uploaded in the system memory (e.g., a cache, or a dedicated flash) and run at-speed, storing its responses in some available memory elements, such as internal registers, caches, or main memory, and hence permitting a low-cost tester to access them at the end of the execution. Similarly, during in-field SBST, at the end of the test program run the processor itself or another module (e.g., another processor) may perform an access to the specific memory cells in order to compare their values with the expected ones.

According to the presented scenario, this observation mechanism assumes that the test program collects in some way the information about test results and saves this information in the system memory. The information collected by the test program may be compacted by the test program itself and then (at the end of the test process) saved in few selected memory cells. Alternatively, the information saved by the program may be written to a set of memory cells, according to the targeted module characteristics as described in [24] for a MIPS-like processor and an industrial System-on-Chip.

Since the test results correspond to the values generated by the test program, which are checked only at the end of the test program execution without taking into account when these results are produced, some performance faults may escape when using this observability mechanism. For example, in the case of Branch Prediction Units, some performance faults may not modify the final test program results, but only delay the actual execution time [4], e.g., by turning a correctly predicted branch into a mispredicted one.

3.5 Performance Counters Observability

Performance Counters (PeCs) measure the number of occurrences of different internal events, making their observation easier from the outside. They exist in many processors, mainly for design validation, performance evaluation and to support silicon debug. Their values can normally be accessed via software. Hence, a test program may read the value of a given PeC, execute a sequence of instructions exercising a given module, and then read again the value of the PeC comparing it to the expected one. Possible differences may allow the detection of faults inside the module.

The most common types of PeCs include those that count internal events related to:

- caches, counting the number of miss and hit events;
- Branch Prediction Units (BPUs), counting the number of correctly or incorrectly predicted branches;
- pipeline stages, counting the different types of stalls;
- Memory Management Units (MMUs), counting the number of hit/miss accesses to the TLB;
- exception units, counting the number of triggered exceptions, often divided by type;
- bus interfaces, counting the number of performed bus transfers, also often divided by type.

These counters are already quite common in general-purpose high performance processors, and their adoption is growing in microcontrollers for embedded applications.

The usage of these counters as part of the observability mechanism adopted by a testing procedure was proposed in several works, such as [4][7] that use variants of the MIPS architecture, or [5][6] working with the OpenSPARC T1 processor. The PeCs have also been proposed as feedback in automatic test program generation [2] and in [37] to simplify the test programs aimed at detecting faults in caches. They are crucial for the detection of some specific types of faults, such as performance faults. Moreover, they can facilitate the test of faults belonging to some modules, such as Branch Prediction Units, Cache Controllers, TLBs. They may also be used to support the test of specific modules within the pipeline, such as those controlling the activation of stalls.

Regarding observability issues, the PeCs may provide deeper details on internal events affecting the module that may not reach the output ports, and allow the detection of several performance faults. Thus, exploitation of PeCs and propagation of performance values to system memory increases observability and may represent a valuable solution during in-field SBST.

4 Experimental results

In this section, we present some experimental figures aimed at assessing the fault detection abilities of the different observation mechanisms described in the previous section; every observability solution is denoted as follows:

- S1: module-level observability
- S2: processor-level observability
- S3: system bus observability
- S4: memory content observability
- S5: performance counters observability.

Two test cases were considered, each corresponding to a different processor-based system. For each test case, a module was selected within the processor and a test program was produced, targeting the faults inside the selected module.

For every observability solution (S1-S5), a fault simulation campaign was run in order to determine whether a fault produces a different processor output compared with the golden run (i.e., the simulation of processor in a fault free condition). A commercial fault simulator was carefully setup in order to mimic the described observability solutions targeting the stuck-at faults in the selected modules. In some cases, it was necessary to introduce some slight modifications to the system oriented to replicate during the fault simulation campaign the observability solutions employed in every experiment. In every one of the two presented test cases the same test program was used during the fault simulation of all the observability solutions.

4.1 Test Case #1: Branch Prediction Unit

For this test case we considered a MIPS-like processor based on the RT-level VHDL description available at [29]. This processor is a 32 bits core composed of:

- a pipeline with 5 stages: instruction extraction (EI), instruction decoding (DI), execution (EX), memory access (MEM), and registers update or write back stage (WB);
- a Program Counter calculation module (PF);
- a System Coprocessor, responsible for the management of interrupts and exceptions;
- a Register Forwarding and Pipeline Interlocking unit, dealing with data hazards among the pipeline stages;
- a Branch Prediction unit (BPU), implementing a Branch Target Buffer.

In this case we considered the faults belonging to the Branch Prediction Unit (BPU), which is a commonly used solution for decreasing the negative impact of branches on the performance of pipelined architectures. One of the most widely used implementations of a BPU is based on the Branch Target Buffer (BTB), i.e., a data structure consisting of a set of entries, each one containing:

- the address of a branch instruction;
- the expected target address in case the branch is predicted as *taken;*
- possibly, one or more bits, storing the prediction (*taken*, or *not taken*).

During the instruction fetch cycle and providing that the instruction is a branch instruction, the processor core is able to anticipate the branch outcome by accessing the BTB. In case of a correct prediction, the processor can fetch the correct instruction during the next clock cycle. Fig. 2 shows the BPU inputs and outputs and their connections to the related pipeline stages of the MIPS-like processor.

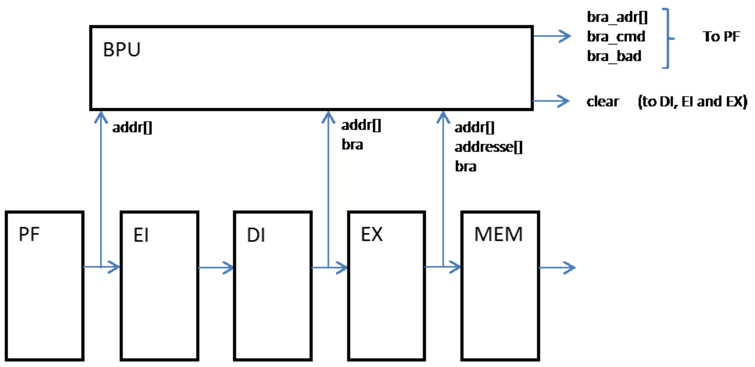

Fig. 2 Branch Prediction unit and related pipeline stages of the MIPS-like processor

This module is particularly interesting, due to the fact that faults affecting it do not always cause the generation of erroneous results, but often impact the processor behavior by simply slowing down the system, i.e., increasing the number of mispredictions and possibly causing the system not to match the expected target in terms of performance. Such faults are also referred to as *performance faults* and can be tested by using some properly devised performance counters.

4.1.1 System setup

Minor changes were done to the processor design, mainly to correct a bug affecting the Branch Prediction Unit[2]. The VHDL description was synthesized using the Synopsys Design Compiler with a technology library developed in-house. The complete processor area is 41,959 equivalent gates and 2,112 equivalent D flip flops, and is described in 3,131 lines of VHDL code, while the BPU accounts for 4,248 equivalent gates, 283 equivalent D flip flops and 274 VHDL lines. The considered fault list, i.e., the single stuck-at faults inside the BPU, has a total of 27,354 faults, corresponding to about 10% of the stuck-at faults inside the whole processor (268,424 faults). No undetectable faults were marked as such by the fault simulation tool.

With the purpose of testing the Branch Prediction Unit, a functional test program was manually developed following the algorithm described in [7]; it occupies 308 bytes of ROM memory, and its execution requires 5,229 clock cycles. The program is a sequence of taken and not taken branches, suitably crafted to exercise each entry of the BTB memory, and each bit of the associated comparator used to determine if the address of a branch instruction being processed matches with the one stored in the table entry. A signature that compacts the sequence of the executed branch addresses is produced and written to memory at the end of the test. Using this test program, we ran several fault simulation experiments aimed at assessing the effectiveness of the different observation mechanisms when addressing single stuck-at faults. Fault simulation experiments were performed using Synopsys

[2] In the original version, the BTB of the processor was composed of 3 entries and the BPU's control logic was buggy; in detail, in case of exception/interrupt preceding the very first branch instruction in the program, the BPU's control logic was wrongly interpreting some (pre-set) addresses as branch instructions, hence the prediction was triggered. In the modified version, the control logic was fixed, and the number of entries was modified to 4.

TetraMAX, which is a well-known tool used for manufacturing testing-oriented fault simulation. A proper framework based on TetraMAX was devised in order to evaluate the considered observability solutions.

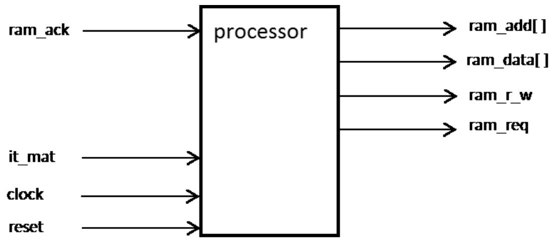

Fig. 3 MIPS-like processor external interface

Fig. 3 shows the interface of the MIPS-like processor. In order to consider a realistic scenario, the synthesized processor was embedded together with a RAM memory and a ROM memory containing the program code. For the sake of simplicity, no peripheral cores were included in the system, thus no external interrupts could have been raised during the simulation experiments.

More in details, a proper module wrapping the processor was included in the system to be used as the top module in TetraMAX fault simulations. This wrapper has a double function: on one hand, it contains all the elements required by the different observation mechanisms, while exporting the different sets of observed signals as primary outputs of the wrapper as described below. On the other hand, it allows the inclusion of the program memory inside the system fault simulated by TetraMAX, hence giving a much closer match between the simulated and real behaviors in the presence of faults. The wrapper includes the RAM memory used by the observation mechanism S4 and a Performance Counter aiming at implementing the observation mechanism S5 described in the previous section. Such PeC is able to count the occurrences of incorrectly predicted branches; it is enabled by a signal (*clear*) internally produced by the Branch Prediction Unit when a wrong prediction is detected.

The wrapper is shown in Fig. 4 and has the following characteristics:

- Two blocks of memory are used; the ROM memory contains the test program code, while the RAM memory is initialized before the simulation and receives the test program results at the end of execution;
- the *clock* and *reset* primary input signals are applied to the processor by the wrapper; at the system reset, the value of the processor's Program Counter corresponds to the first available address in the ROM memory;
- a second read port was added to the RAM memory to support the memory content observation mechanism. Using the additional address inputs (*spy_addr*) and data outputs (*spy_dout*), the memory content is observed without interfering with the normal memory access operations performed by the test program;

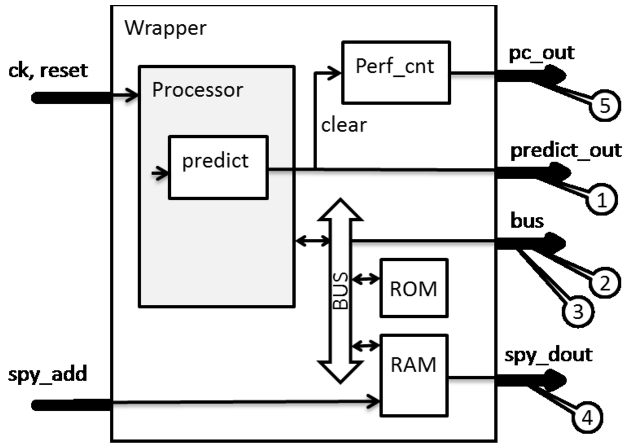

Fig. 4 BPU's test case: experimental environment and considered observability solutions

The fault simulation experiments were driven by a Value Change Dump (VCD) file produced via logic simulation by means of a test-bench surrounding the wrapper, which represents the top module in the TetraMAX simulations. The VCD file provides the proper stimuli to the wrapper's primary inputs (*clock*, *reset*, and *spy_addr*). In detail, the *reset* signal was only triggered as soon as the simulation was started and then remained inactive until the end. At the end of the test program execution, the *spy_addr* value continuously scans the memory interval between the first and the last address of the RAM zone to be observed (through *spy_dout*).

During the fault simulation campaigns, TetraMAX was provided with the same fault list for all of the performed experiments (consisting of all the stuck-at faults on the BPU). The different observation solutions (also shown in Fig. 4) were implemented as follows:

- S1: the primary outputs of the Branch Prediction module (the predicted address, and other few control signals shown in detail in Fig. 2 and grouped as *predict_out* in Fig. 4) are observed during the whole simulation. To make this possible, the processor interface was modified to export the outputs of the internal Branch Prediction module up to the wrapper interface.
- S2: the sub-set of signals composed of all the original processor outputs is observed during the whole simulation (the memory data, address and control interface signals detailed in Fig. 3 and labeled as *bus* in Fig. 4).
- S3: the original processor outputs related to the system bus are observed during the whole simulation. This solution is identical to S2 in this case study because all the outputs of the MIPS-like processor are system bus related.
- S4: the content of the memory area where the test program is expected to write is observed at the end of the test program execution; this is implemented using the wrapper. More in details, upon the test program completion, the observation is enabled on the outputs *spy_dout,* while the inputs *spy_addr* sweep the RAM memory interval devoted to save the program results. Two words are enough for the present example because a signature is used to compact the test program results.
- S5: upon the test program completion, the observation of the primary outputs corresponding to the Performance Counter value is enabled (*pc_out* in Fig. 4).

4.1.2 Results

TABLE I BPU'S TEST CASE: FAULT SIMULATION RESULTS

Class	**S1**	**S2/S3**	**S4**	**S5**	**S4+S5**
Detected	21,631	21,614	11,253	19,887	19,887
Not Detected	5,723	5,740	16,101	7,467	7,467
Fault coverage [%]	79.08	79.01	41.14	72.70	72.70
Fault coverage (normalized to S1) [%]	100	99,92	52,02	91,94	91,94

The fault simulation results for the Branch Prediction Unit of the MIPS-like processor are reported in TABLE I. As explained above, for this processor the observed signals in mechanisms S2 and S3 are exactly the same. For this reason a single column labeled as S2/S3 is shown in TABLE I. About 79% of the total faults in the module were detected by exploiting the observation mechanism S1: this represents the maximum achievable fault coverage, given the test program we considered. Only a few of these faults were not propagated through the system bus (S2/S3). This happens because the test program used in the experiments was carefully developed focusing on the end-of-manufacturing scenario; thus, a small minority of the faults detected by S1 were marked as not observed by TetraMAX.

On the contrary, a significant fault coverage drop can be noticed when moving from S2/S3 to S4, when the content of the main memory is observed at the end of the test program execution. In this case, only 41% of faults are marked as detected, meaning that about a half of the faults that were propagated by the test program to the system bus (experiment S2/S3) were not consistently saved in the main memory, turning them into not observable in a typical in-field scenario. As described in the introduction, part of these faults are masked, however, those of them that are not performance faults may become observable if correctly propagated up to the system memory. Thus, this result could be improved by the test engineer in charge of test program, if this scenario had to be targeted.

The interesting result concerns the experiment which used the Performance Counters (S5): in this case we have about 73% of detected faults, meaning that only 6% of faults were not observable with respect to the results of the processor-level observability (S2/S3). Since the PeCs is tightly related to the module under consideration, its effectiveness was very high, providing a good example of how resources existing in real processors can be exploited for in-field testing. This result also suggests that in the present case, performance faults are by far the most important factor contributing to the coverage reduction from S2/S3 to S4.

Further analysis on the fault lists of the different experiments showed that all the faults that were detected for S4 were also included in the detected fault list of S5, and consequently the results of combining S4 and S5 (rightmost column in the table) are identical to the results obtained with only S5. As discussed above, the PeCs can usually be accessed as a system peripheral, thus its values can be read by the test program at the end of its execution and then, used to update the test signature. Therefore, in a typical in-field scenario, the adoption of S5 allows achieving a high fault coverage, with the minor additional effort of the PeCs reset and reading operations.

4.2 Test case #2: Data Cache Controller

The second test case corresponds to a multi-core system based on the LEON3 processor, whose synthesizable VHDL model is freely-available for research purposes under GNU GPL license at [30], as part of the GRLIB IP library [25]. The LEON3 processor is a highly configurable 32-bit core compliant with the SPARC V8 architecture and is composed of:

- a pipeline with 7 stages;
- AMBA-2.0 AHB bus interface;
- separate instruction and data caches, which can be configured in several ways;
- several optional components: high-performance, fully pipelined IEEE-754 Floating-Point Unit; MMU; Debug Support Unit with instruction and data trace buffer.

LEON3 data cache can include data cache coherency features based on the *snooping* technique. Data cache snooping is of high importance for multiprocessor systems. The purpose of this logic is to keep the data cache synchronized with external memory and other caches, avoiding any data inconsistency. This task is accomplished by monitoring the write accesses on the AMBA AHB bus to cacheable memory locations: if another AHB master writes to a cacheable location that is currently cached in the data cache, the corresponding cache line is marked as *invalid*.

4.2.1 System setup

The GRLIB IP library provided by Cobham Gaisler AB was exploited in order to create the Symmetric Multi-Processor (SMP) embedded platform depicted in Fig. 5. Note that this platform permits to exercise the data cache coherency logic. Briefly, the system included:

- two LEON3 processors, referred to as *core0* and *core1* in the figure, each one attached to the AMBA-2.0 AHB bus as master;
- a bank of SDRAM memory, and a ROM memory loaded with the program;
- an interrupt controller;
- a 16-bit GPIO port.

Both instruction and data caches in LEON3 processors were configured as 1-way, 1Kbyte, 16 bytes/line. The data cache policy is always write-through, with no cache allocation on a write miss. The snoop mechanism was enabled to assure cache coherency.

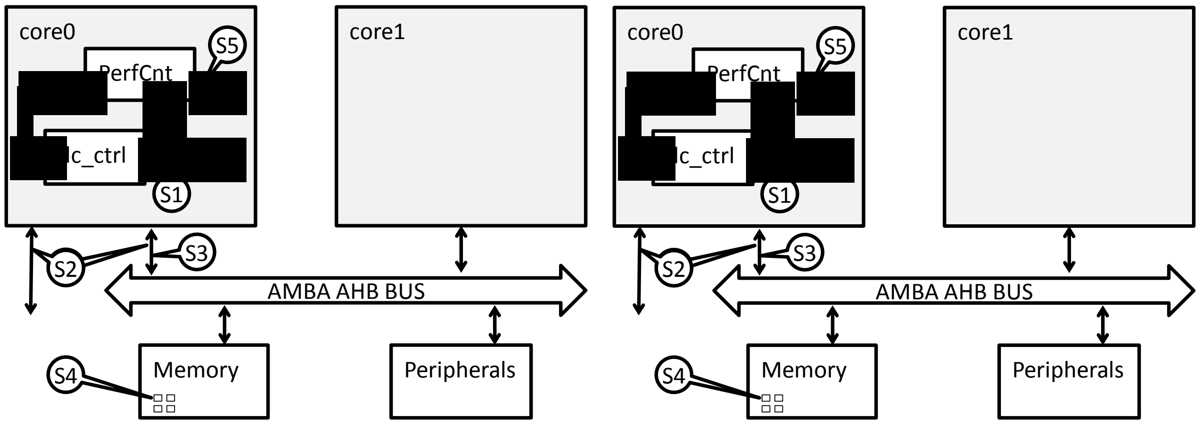

Fig. 5 Data Cache Controller test case: dual-core system under test and considered observability solutions

In this test case the top module of the system simulated in the fault simulation experiments was the *core0* processor. Previously, the whole two core system was simulated without faults to obtain the patterns applied to *core0* during fault simulation. The internal module under analysis was the data cache controller of *core0* (*dc_ctrl* in Fig. 5), and the considered faults were the stuck-at faults inside it. In order to better assess the observability solution based on performance counters, this module was enriched by adding some of the performance counters usually available on commercial processors. For this purpose, a new block was designed, called *PerfCnt* in Fig. 5, fed by a subset of the data cache controller ports. The set of performance counters implemented by the block can be classified in the following two groups:

a) *AMBA bus events*: a group of 5 counters triggered on the occurrence of specific bus transfers (write, read, byte, half word, and word transfers). These counters take their inputs from the connections between the data cache controller and the AMBA AHB interface.

b) *Cache operation events*: 3 counters for cache read hit, cache write, and cache line invalidation because of snoop events. The inputs of these counters come from signals connecting the data cache controller with the cache memory and the integer unit.

The whole system was synthesized with Synopsys Design Compiler using the Synopsys SAED32 standard cells library [26], one of the libraries provided in the synthesis toolkit of the processor. The complete Leon3 processor core is described in 9,578 VHDL code lines. Each of the two instances of the Leon3 processor core occupies 155,568 equivalent gates, and 23,784 equivalent D flip flops. As mentioned before, the module under analysis is the data cache controller in *core0* processor, which is described in 1,669 VHDL lines, and occupies 6,380 equivalent gates and 339 equivalent D flip flops. The fault list used during fault simulation experiments consists of all the possible single stuck-at faults affecting the module under analysis, which totalize 23,958 faults, a fraction of the 715,838 stuck-at faults of the whole processor core.

For the purpose of this set of experiments, a test program was created based on the techniques described in [8] and [27], aimed at detecting faults affecting different specific functionalities inside the data cache controller. One of these functionalities is the snooping logic, whose basic function is to invalidate a given cache line when another processor executes a write operation on the memory block it stores. The part of the test that targets the snooping logic, as described in [27], runs concurrently on both processors. It consists of a sequence of memory accesses exciting the inputs of the address comparators used to detect the cache line invalidation condition and verifying whether the corresponding validity bit is modified accordingly. The rest of the test program runs only on *core0*. It includes a section targeted to the replacement logic [8] and the programs provided by Cobham Gaisler AB to verify the functionalities of the cache controller. The test program was written in C language except for some small sections which are written at the assembly level; its execution time is about 300k clock cycles, and its code occupies about 26,000 bytes.

To allow a fair comparison, the test program was exactly the same in all the experiments. Each time, a sub-set of the outputs was selected and made observable by TetraMAX in order to mimic the different observability solutions presented in section 3. For each of the considered solutions, the sub-set of observed signals is listed below, as well as the observation points (also shown in Fig. 5):

- S1: the sub-set of primary outputs corresponding to all the output ports of the data cache controller (*dc_ctrl* module) is observed during the whole test program execution.
- S2: the sub-set related to all the outputs of the *core0* processor is observed during the whole test program execution. This sub-set includes the AMBA AHB bus interface outputs described in the next paragraph, the outputs to an external debug support unit (DSU3) and the outputs to an external interrupt controller (IRQMP) that also manages the startup sequence of the multiple processors.
- S3: the sub-set related to the outputs of the *core0* processor that are connected to the AMBA AHB bus is observed during the whole test program execution. It includes the 32 bit address bus, the 32 bit data bus and the full set of control signals of an AMBA AHB master interface.
- S4: the final signature computed by the test program out of the data written in memory is observed at the end of the test program execution.
- S5: the primary outputs corresponding to the *PerfCnt* module are observed at the end of the test program execution (outputs of the eight counters described above).

4.2.2 *Results*

TABLE II DATA CACHE CONTROLLER'S TEST CASE: FAULT SIMULATION RESULTS

Class	**S1**	**S2**	**S3**	**S4**	**S5**	**S4+S5**
Detected (DT)	12,112	8,517	8,516	7,511	2,479	7,796
Not Detected (ND)	9,726	9,627	9,628	10,633	17,483	17,483
Undetectable (UD)	2,120	5,814	5,814	5,814	3,996	3,996
Fault coverage [%]	50.56	35.55	35.55	31.35	10.35	32.54
Fault coverage (normalized to S1) [%]	100	68,07	68,06	60,03	19,81	64,37

The fault simulation results for the data cache controller module of the LEON3 processor are summarized in TABLE II. The gathered results show, as expected, quite different fault coverage figures when the different observation mechanisms are adopted. The limited fault coverage achieved by the different solutions (including S1) is mainly due to the high amount of redundancy (hence, untestability) existing in the considered implementation of the Data Cache Controller module, which could not be removed by the synthesis tool. This redundancy stems from the high flexibility of the LEON3 cache system, which can be configured at compile time to implement different cache solutions, not only in terms of cache size, but also of cache replacement and writing mechanism (e.g., write-back or write-through). Unfortunately, after configuration the VHDL code contains unused structures that are not always removed by the synthesis tool, leading to redundancy. TetraMAX is partially able to identify untestable faults and classify them as Undetectable (UD) under different situations: during the net-list compile, because they are located on circuitry with no connectivity to an externally observable point, or because the fault effect is blocked from propagating to an observable point due to tied logic; alternatively, during the fault simulation, when some faults are proved to be untestable since they are located on redundant circuitry, by means of formal analysis. However, part of the faults, which cannot be tested in a functional manner, escape from the analysis of testability made by TetraMAX, and simply result as part of the Not Detected (ND) set.

In order to consider that there is a high percentage of untestable faults, the bottom row in TABLE II shows the fault coverages expressed as a percentage of the one obtained with solution S1. It can be seen that the number of faults detected by solutions S2 and S3 is almost 70% of the one detected by solution S1, meaning that a significant amount of faults that are observable at module-level are not propagated up to the processor output signals.

Interestingly, solution S2 covers only one fault more than solution S3. This was partly expected, because the data cache controller module is strongly memory related. As analyzed above in Section 3.3, a more appreciable coverage difference between solutions S2 and S3 can be expected for a non-memory related module. Such a difference may be produced because the fault effects are not properly propagated to the system bus by the test program (e.g. a coprocessor module), or because the fault only affects resources that cannot be read back in a controlled way by the test program (e.g., an interrupt controller).

The provided results also show that the number of faults detected by S4 is about 88% of the number of faults covered by S3, and the number of faults covered by S5 is only 33% of the number of faults covered by S4. However, it is important to analyze the intersections and differences between the sets of faults detected by S4 and S5.

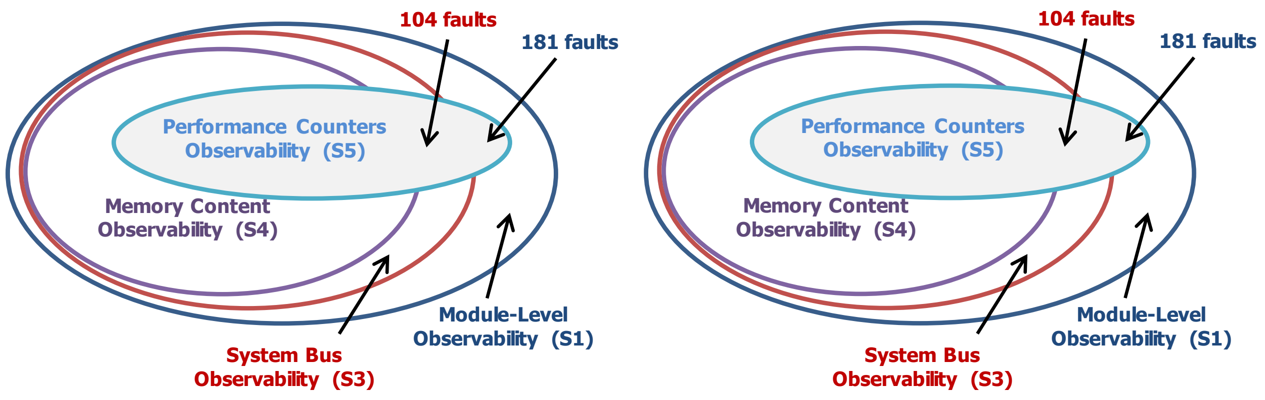
104 faults
181 faults
Performance Counters Observability (S5)
Memory Content Observability (S4)
Module-Level Observability (S1)
System Bus Observability (S3)
104 faults
181 faults
Performance Counters Observability (S5)
Memory Content Observability (S4)
Module-Level Observability (S1)
System Bus Observability (S3)

Fig. 6 Data Cache Controller's test case: sets of faults detected by the different observability solutions

As mentioned in the previous sections, in an in-field SBST approach solutions S4 and S5 are the only practically feasible ones, since they do not require the use of any additional hardware devices. Fig. 6 schematically shows the effect of adding the results obtained by S4 and S5. S5 provides 285 additional detected faults, about 3.8% of the number of faults detected by S4 alone. S4 and S5 combined detect 91.5% of the faults covered by S3. Remarkably, there are some faults (104) covered by S5 and S3 that are not detected by solution S4; these faults are associated to the group of performance counters related to *AMBA bus events* (see the previous sub-section). This group of faults shows a contribution of S5 which partly compensates the loss in fault coverage moving from S3 to S4.

Additionally, there is another set of faults (181 faults) covered only by S5 that escape the test not only by S4 but also by S3. This means that during the execution of the test program the effect of these faults is unobservable at the bus level, but modifies the behavior of the second group of performance counters (*cache operation events*, described in the previous sub-section). The considered test program is successful in exciting these faults but is not able to produce observable modifications at the bus level. The observation of the performance counters contributes to the overall coverage by making these faults observable.

4.3 Analysis of the results

Experimental results successfully confirmed some a-priori considerations on the implemented observation solutions. By construction, it can be first of all stated that, for a given test program, the higher coverage is the one obtained resorting to solution S1, which offers the highest observability even if it can hardly be implemented in a real scenario. For all the other solutions, the observed signals are a subset or a transformation of the signals observed in S1. So, for example in test case #2, if the effect of a fault inside the *dc_ctrl* module is observable at the AMBA bus level (S3), it can also be observed on the *dc_ctrl* ports (S1).

In a similar way, the following inclusion relationships (also summarized in Fig. 6) can be assured between the sets of detected faults and between the sets of undetectable faults in the different solutions. The name in parentheses identifies the solution and the two letter code denotes the fault class (*DT*: detected, *UD*: undetectable).

$$DT(S1) \supseteq DT(S2) \supseteq DT(S3) \supseteq DT(S4) \quad (1)$$

$$UD(S1) \subseteq UD(S2) \subseteq UD(S3) \subseteq UD(S4) \quad (2)$$

$$DT(S1) \supseteq DT(S5) \quad (3)$$

$$UD(S1) \subseteq UD(S5) \quad (4)$$

All these relationships were verified in both test cases by analyzing the detailed fault lists for each of the observability solutions.

An additional experiment was performed in test case #2 trying to understand how much the final fault coverage can be enhanced if a larger set of performance counters is used. In this experiment, the data cache controller subset of signals used as inputs by the performance counters module was observed. Denoting by S6[3] this new observability solution, the following extensions to the above inclusion relationships can be stated:

$$DT(S1) \supseteq DT(S6) \supseteq DT(S5) \quad (5)$$

$$UD(S1) \subseteq UD(S6) \subseteq UD(S5) \quad (6)$$

In other words, all the faults detected by S5 are also detected by S6, and this remains true even if the design of the *PerfCnt* module is modified. So, the coverage obtained by solution S6 is an upper bound to the coverage that can be obtained by any design of the *PerfCnt* module with the same inputs. The results for test case #2 show that solution S6 adds 511 new detected faults with respect to S4. Another interesting result from this new fault simulation experiment is that, when observing the performance counters inputs, the set of undetectable faults is identical to the one obtained with S1. This means that the same faults detected by S1 could be potentially detected by adding new Performance Counters and devising a suitable input sequence.

Finally, it is interesting to compare the coverage results of combining S4 and S5 in both test cases. In the Branch Prediction Unit case there are two points to highlight: S5 provides a very good coverage (92% of the coverage obtained by S1) and the set of faults covered by S4 is completely included in the set of faults covered by S5 as shown in Fig. 7. This is explained because the misbehaviors produced by the faults inside the Branch Prediction Unit naturally affect performance, and modify the count of incorrectly predicted branches, but most of them do not affect the result of calculations and therefore they do not affect the final signature value.

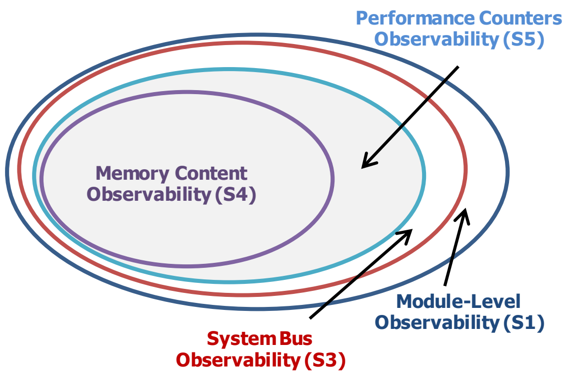
Performance Counters Observability (S5)
Memory Content Observability (S4)
Module-Level Observability (S1)
System Bus Observability (S3)

Fig. 7 Branch Prediction Unit's test case: sets of faults detected by the different observability solutions

On the other hand, for the Data Cache Controller test case, the performance counters observation was introduced aiming to cover the faults that turn a hit into a miss, or vice versa. The relatively low coverage obtained in solution S5 suggests that, in the complete cache controller, an important amount of faults exist that do not produce this kind of hit/miss permutation, and therefore modify neither the count of AMBA bus transfers, nor the count of hits/misses. Fortunately, these faults affect the processed results and can be exposed by a proper signature as shown by S4. Interestingly, by exploiting contemporarily solutions S4 and S5 (rightmost column in TABLE II), the obtained results are similar to those obtained by observing the processor primary outputs as in one of the end-of-manufacturing testing scenarios.

[3] We'd like to highlight the fact that S6 in a purely theoretical solution, which is introduced only to compute the upper bound of the fault coverage achievable with S5.

5 Conclusions

When adopting SBST for in-field test of a processor-based system, as it is often required by standards and regulations for safety-critical applications, the achieved fault coverage is often affected in a strong manner by observability issues.

In this paper, we analyzed the different solutions that can be adopted, both from a theoretical and an experimental point of view. When considering the latter approach we adopted two test cases, including a Branch Prediction Unit in a processor-based system, and a Data Cache Controller logic module in a dual-processor system based on the LEON3 processor, to gather quantitative data about the faults that can be detected using the different observability solutions. In both cases, we focused on the faults related to the processor core. Based on our experience and on data gathered with other processors and test cases, which cannot be reported in this paper for lack of space, we believe that the conclusions drawn in this paper are quite general.

A major contribution of this paper lies first in showing that when reusing the same test program developed with one observability solution to another the fault coverage may significantly change. Secondly, we experimentally demonstrated that a careful combination of observability mechanisms based on checking the memory content at the end of the test program execution and on the information coming from the Performance Counters existing in many processors allows to achieve a fault coverage figure not far from the maximum one. More generally, a suitable set of Performance Counters may allow achieving nearly the same fault coverage achieved when continuously observing all the processor outputs, without significant extra costs for their detection. We also described in details some solutions to implement suitable fault simulation campaigns able to gather the required experimental data.

Although all the experiments described in this paper refer to the single stuck-at fault model, the approach we followed for our work and the main conclusions we drew also apply to other fault models, including for example the transition delay. Further activities are currently being done to extend the analysis to extend our results in this direction.

We are now working towards the development of effective techniques to identify faults that become untestable in the operational scenario.

Acknowledgements

We would like to thank Abel Hailemichael for his initial work on this topic. This work has been partly supported by the European Commission through the FP7 STREP Project No. 619871 (BASTION).

References

[1] M. Psarakis et al., "Microprocessor Software-Based Self-Testing", IEEE Design & Test of Computers, vol. 27, no. 3. May-June 2010, pp. 4-19.

[2] Lindsay, W., Ernesto Sánchez, M. Sonza Reorda, and Giovanni Squillero, "Automatic test programs generation driven by internal performance counters", Microprocessor Test and Verification (MTV'04), pp. 8-13. IEEE, 2004

[3] N. Karimi, M. Maniatakos, C. Tirumurti, and Y. Makris, "On the Impact of Performance Faults in Modern Microprocessors", Journal of Electronic Test, 29, 3 (June 2013), Springer, pp. 351-366.

[4] M. Hatzimihail, M. Psarakis, D. Gizopoulos, A. Paschalis, "A methodology for detecting performance faults in microprocessors via performance monitoring hardware", IEEE International Test Conference, 2007, pp. 1 – 10.

[5] Theodorou, G.; Kranitis, N.; Paschalis, A.; Gizopoulos, D., "Software-Based Self Test Methodology for On-Line Testing of L1 Caches in Multithreaded Multicore Architectures," in Very Large Scale Integration (VLSI) Systems, IEEE Transactions on , vol.21, no.4, pp.786-790, April 2013.

[6] Theodorou, G.; Kranitis, N.; Paschalis, A.; Gizopoulos, D., "Software-Based Self-Test for Small Caches in Microprocessors," in Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on , vol.33, no.12, pp.1991-2004, Dec. 2014.

[7] E. Sanchez, M.Sonza Reorda, "On the Functional Test of Branch Prediction Units", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 23, no. 9, pp. 1675-1688, Sept. 2015.

[8] W.J. Perez, D. Ravotto, E. Sanchez, M. Sonza Reorda, A. Tonda, "On the Generation of Functional Test Programs for the Cache Replacement Logic", 18th IEEE Asian Test Symposium 2009 (ATS09), 2009, pp. 418-423.

[9] P. Bernardi, M. Bonazza, E. Sanchez, M. Sonza Reorda, O. Ballan, "On-line functionally untestable fault identification in embedded processor cores", Design, Automation & Test in Europe Conference & Exhibition (DATE), 2013.

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

[11] P. Parvathala, K. Maneparambil, W. Lindsay, "FRITS - a microprocessor functional BIST method", IEEE International Test Conference, 2002, pp. 590 – 598.

[12] S. Gurumurthy, M. Pratapgarhwala, C. Gilgan, J. Rearick, "Comparing the effectiveness of cache-resident tests against cycle accurate deterministic functional patterns", IEEE International Test Conference, 2014, pp. 1-8.

[13] Kranitis, N.; Paschalis, A.; Gizopoulos, D.; Xenoulis, G., "Software-based self-testing of embedded processors," IEEE Transactions on Computers, vol. 54, no. 4, pp. 461-475, April 2005.

[14] L. Chen and S. Dey., "Software-based self-testing methodology for processor cores," IEEE Trans. on Computer-Aided Design, vol. 20, no.3, March 2001, pp. 369-380.

[15] F. Corno, E. Sanchez, M. Sonza Reorda, G. Squillero, "Automatic Test Program Generation: a Case Study", IEEE Design & Test of Computers, vol. 21, pp. 102-109, 2004.

[16] A. Riefert, R. Cantoro, M. Sauer, M. Sonza Reorda, B. Becker, "On the Automatic Generation of SBST Test Programs for In-Field Test", Design, Automation & Test in Europe Conference & Exhibition (DATE), 2015, pp. 1186 – 1191

[17] G. Theodorou, S. Chatzopoulos, N. Kranitis, A. Paschalis, D. Gizopoulos, "A Software-Based Self-Test methodology for on-line testing of data TLBs",IEEE European Test Symposium (ETS), 2012.

[18] Li Chen, S. Dey, "Software-based diagnosis for processors", IEEE/ACM. Design Automation Conference, 2002, pp. 259 – 262.

[19] D. Appello, P. Bernardi, M. Grosso, E. Sanchez, M. Sonza Reorda, "Effective Diagnostic Pattern Generation Strategy forTransition-Delay Faults in Full-Scan SOCs", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 17 (11), pp. 1654-1659 .

[20] A. Paschalis, D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors", Design, Automation and Test in Europe Conference and Exhibition (DATE), 2004, pp. 578 – 583.

[21] Freescale Semiconductor, "e200z4 Power Architecture Core Reference Manual," 2009.

[22] ARM, "AMBA AHM Trace Macrocell (HTM)," 2008.

[23] Bernardi P; Grosso M.; Rebaudengo M; Sonza Reorda M "Exploiting an I-IP for both test and silicon debug of microprocessor cores", IEEE International Workshop on Microprocessor Test and Verification (MTV), 2005, pp. 55-62.

[24] P. Bernardi, L. Ciganda, M. De Carvalho, M. Grosso, J. Lagos-Benites, E. Sanchez, M. Sonza Reorda, O. Ballan (2012) "On-Line Software-Based Self-Test of the Address Calculation Unit in RISC Processors", 17th IEEE European Test Symposium (ETS), 2012

[25] Gaisler, J., Catovic, E., Isomaki, M., Glembo, K., Habinc, S., "GRLIB IP core user's manual. Version 1.3.7 - B4144", Gaisler research, 2014.

[26] Goldman, R., Bartleson, K., Wood, T., Kranen, K., Cao, C.; Melikyan, V., Markosyan, G., "Synopsys' open educational design kit: Capabilities, deployment and future," IEEE International Conference on Microelectronic Systems Education, 2009 (MSE '09), July 2009.

[27] Perez Acle, J., Cantoro, R., Sanchez, E., Sonza Reorda, M., "On the Functional Test of the Cache Coherency Logic in Multi-core Systems.", 6th IEEE Latin American Symposium on Circuits and Systems, LASCAS, 2015.

[28] Fournier, Laurent, Yaron Arbetman, Moshe Levinger, "Functional verification methodology for microprocessors using the genesys test-program generator. Application to the x86 microprocessors family", Design, Automation and Test in Europe Conference and Exhibition (DATE), 1999, pp. 434-441.

[29] miniMIPS Website: http://opencores.org/project,minimips

[30] LEON3 Website: http://www.gaisler.com/index.php/products/processors/leon3

[31] J. Perez, R. Cantoro, A.T. Hailemichael, E. Sanchez, M. Sonza Reorda, "Observability solutions for in-field functional test of processor-based systems", 30th IEEE Conference on Design of Circuits and Integrated Systems (DCIS), 2015.

[32] J. Shen and J. A. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," IEEE International Test Conference (ITC), 1998, pp. 990-999.

[33] J. Shen and J. A. Abraham, "Synthesis of native mode self-test programs," Journal of Electronic Testing, Springer, vol. 13, no. 2, 1998, pp. 137-148.

[34] J. Sosnowski, "Software-based self-testing of microprocessors", Journal of Systems Architecture, vol. 52, Elsevier, 2006, pp. 257-271

[35] M. A. Skitsas, C. A. Nicopoulos, M. K. Michael, "DaemonGuard: Enabling O/S-Orchestrated Fine-Grained Software-Based Selective-Testing in Multi-/Many-Core Microprocessors", IEEE Transactions on Computers, 2016, Volume: 65, Issue: 5, pp. 1453 – 1466

[36] S. Di Carlo, P Prinetto, A Savino, "Software-based self-test of set-associative cache memories", IEEE Transactions on Computers, Vol. 60, No. 7, July 2011, pp. 1030-1044

[37] J. Sosnowski, "Improving Software Based Self-Testing for Cache Memories," 2nd International IEEE Design and Test Workshop, 2007, pp. 49-54

[38] A. Apostolakis, D. Gizopoulos, M. Psarakis, D. Ravotto, M. Sonza Reorda, "Test Program Generation for Communication Peripherals in Processor-Based SoC Devices", IEEE Design & Test of Computers, 2009, Volume: 26, Issue: 2, pp. 52-63

[39] P. Bernardi, L. Ciganda, M. Sonza Reorda, S. Hamdioui, "An Efficient Method for the Test of Embedded Memory Cores during the Operational Phase", 22nd IEEE Asian Test Symposium, 2013, pp. 227 - 232

[40] M. G. Karpovsky, V. N. Yarmolik, "Transparent memory BIST", IEEE International Workshop on Memory Technology, Design and Testing, 1994, pp. 106-111