About Performance Faults in Microprocessor Core in-field Testing

About Performance Faults in Microprocessor Core in-field Testing

Julio Perez Acle*, Ernesto Sanchez† and Matteo Sonza Reorda†

* Facultad de Ingenieria, Universidad de la Republica, Montevideo, Uruguay
Email: julio@fing.edu.uy
† Dip. Automatica e Informatica, Politecnico di Torino, Torino, Italia
Email: {ernesto.sanchez, matteo.sonzareorda}@polito.it

***Abstract*—When microprocessor-based devices are used in safety-critical applications (e.g., in automotive systems), it is common to adopt solutions aimed at testing them in-field, so that permanent faults that may affect them are identified before they cause critical consequences. In this way, the required reliability figures can be achieved. A popular solution to perform in-field test (especially when executed concurrently to the application) is based on triggering the execution of proper procedures (composing a Self-Test Library, or STL), which are able to activate faults and make them visible when checking the produced results (e.g., in memory). Unfortunately, a special class of faults exists (named *Performance Faults*), which do not impact the value of the results, but only the timing behavior of the processor. This paper describes a set of experiments aimed at quantitatively evaluating the number of these faults in a simple processor core, and outlines some observation techniques that can be used for their detection.**

***Index Terms*—test; reliability; functional safety; performance faults;**

I. Introduction

When electronic systems are used in safety critical applications (e.g., in the space, avionic, automotive or biomedical areas), we need to guarantee that the probability of failures due to faults of any kind (unreliability) is lower than a specified threshold. One of the possible causes of failures are defects affecting the hardware components. Different techniques can be used to reduce the chance that hardware defects can occur (e.g., acting on the adopted semiconductor technology) or to minimize the probability that they produce critical failures (e.g., by introducing redundancy). Unfortunately, most of these techniques have a severe impact on the cost of the resulting product. In some cases (especially when advanced semiconductor technologies are used, e.g., to achieve enough performance) the probability of failures is anyway too large. Hence, a solution which is commonly used in several scenarios lies on periodically performing a test able to detect the occurrence of any fault before it produces a failure (*in-field test*). This kind of test can resort either to Design for Testability (DfT) or to functional approaches. The latter solution is normally based on forcing the CPU inside the Device Under Test (DUT) to execute a properly written test program, which is able to activate possible faults and to make their effects visible in some observable locations (e.g., the data memory). This approach (also called *Software-based Self-test*, or SBST [1]) has the advantage of testing the DUT exactly in the same conditions of the application, and is more suitable for concurrent in-field test, since it is less intrusive than DfT. On the other side, SBST solutions may require a large effort to develop a suitable test program.

The SBST approach is currently experiencing a growing success, mainly because it offers the possibility (besides the other advantages) to the semiconductor company manufacturing the device (and knowing its internal structure) to develop the test code, grade it in terms of achieved Fault Coverage, and pass it to the system company, which eventually integrates it in the application code. The system company is also in charge of developing the code in charge of launching the test and retrieving the results it produced, managing the situations where a fault is detected. Since the test code is often activated in small chunks, whose execution can fit in the idle times of the application, it is convenient to organize it in a set of procedures, composing a *Self-Test Library* (STL). STLs are currently offered by several semiconductor and IP companies [2] [3] [4] [5] [6] [7].

When developing the code of a STL, special techniques must be followed to activate the target faults and to make them visible. The latter point is particularly important, especially because during in-field test the observability of the DUT behavior is necessarily limited. Hence, several solutions can be adopted, possibly involving the support of existing or ad hoc hardware [8].

When considering permanent faults that may affect an electronic device (such as a microprocessor, or a System on Chip), a special class is represented by *Performance Faults*, i.e., those faults which do not affect the results of the computation, but simply the timing behavior of the DUT. Examples of these faults can be found in a Branch Prediction Unit (BPU). If the BPU is faulty, it may always produce a wrong branch prediction. The final result of the program execution will be correct, but the time required for the execution will be larger. Performance Faults can be found in other parts of a processor, such as the cache and memory controller. Clearly, the relevance of Performance Faults from a practical point of view may change depending on the application. Since real-time constraints are often important, in many cases they cannot

be neglected and need to be detected.

Some previous works already dealt with Performance Faults. In particular, in [9] the authors describe a method to detect them resorting to Performance Counters, i.e., those hardware structures which are often included in a processor to count the occurrence of internal events (e.g., cache misses, or wrong branch predictions), mainly to support the manufacturer in assessing the correct behavior of the design. In [10] a method to estimate the impact of performance faults on different performance-related modules in a CPU is proposed.

The goal of this paper is to expand the work done in [8] focusing on performance faults and showing first of all some figures to quantitatively assess their presence in the different parts of a pipelined CPU module. Secondly, the paper lists a number of observation mechanisms, and experimentally evaluate their effectiveness in detecting performance faults. The reported analysis can be precious for the test engineer to cleverly decide which observation mechanisms have to be implemented in a STL, and for the designer of a CPU or SoC to judge whether it is worth to add some special hardware to support SBST code development. The rest of the paper is organized as follows: Section II presents the different observation mechanisms we considered, Section III describes the experimental setup and analyze the obtained results, and Section IV presents the conclusions.

II. Observation Mechanisms

In the following, the solutions considered to observe the effects of possible faults during in-field SBST testing of a processor-based system are briefly presented. A detailed description can be found in [8] and [11]. Some of the observation methods are considered only to be taken as ideal reference solutions, as they can hardly be adopted during in-field SBST.

Processor-Level Observation (solution *S2*) assumes that all the processor outputs can be continuously monitored. It is one of the scenarios that are commonly adopted for end-of-manufacturing test. Due to the need of constant monitoring of all the processor outputs, this observation solution requires the use of an ATE and thus, cannot be adopted by in-field SBST.

System Bus Observation (solution *S3*) is similar to the previous one but excluding from observation all the processor outputs not related with the system bus. Again, it is very difficult to adopte this technique for in-field testing.

When using *Memory Content Observation* (solution *S4*), 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. No dedicated tester or any other equipment is needed, so this is one of the most commonly adopted solutions for in-field SBST. Since the test results 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 observation mechanism.

Performance Counters Observation (solution *S5*). Performance Counters (PeCs) measure the number of occurrences of different internal events from modules like caches, pipeline stages and bus interfaces. They are already quite common in general-purpose high performance processors, and are used 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, and then read again the value of the PeC comparing it to the expected one. The usage of these counters as part of the observation mechanism adopted in testing was proposed in several works, such as [9] [12] [13] [14]. Regarding observability issues, the PeCs may provide deeper details on internal events that may not reach the output ports, and allow the detection of several performance faults. Thus, exploitation of PeCs and propagation of their values to system memory increase observability and may represent a valuable solution during in-field SBST.

Debug Interface Observation (solution *S6*) uses the features currently provided by many processors to support the debug of the software. Examples of such debug interfaces are the vendor independent standard Nexus IEEE-ISTO 5001 [15] [16] extensively used in U.S. automotive applications, and the ARM CoreSight On-chip Trace and Debug Architecture [17], widely adopted by chip vendors using ARM Cortex cores.

These features typically allow tracing the sequence of instructions executed by the processor (without slowing it down), either by writing them to an external interface that can be accessed on-the-fly or by storing them in a special memory.

In order to use the mentioned external interface to observe in-field test results, ad hoc hardware is required to produce a signature of the flow of data produced by the debug interface. Although such ad hoc hardware is usually not available in a typical in-field test scenario, it may be added if some programmable hardware is available on the board or on-chip. This approach is particularly attractive in the SoC platforms provided by FPGA vendors, often equipped with ARM processors and the mentioned ARM CoreSight Architecture, e.g., the Zynq-7000 SoC platform by Xilinx or Cyclone V SoC family by Intel. On these platforms, the ad hoc port provided by the debug interface is connected to the FPGA fabric, so that an on-the-fly monitor can be added with a moderate effort. Such a scheme is presented in [18] and its effectiveness in particular to detect control flow errors, i.e., faults that modify the normal program execution flow, is experimentally evaluated.

III. Experimental setup and results

A. Experimental setup

The experimental setup we use for the purpose of this work is similar to the one described in [8], with two additional observation methods. It uses a MIPS-like 5-stage pipeline processor, based on the RT-level 3,131 lines VHDL description available at [19]. When sinthesized with a generic library, the processor size is 41,959 equivalent gates and 2,112 equivalent D flip flops. The fault list consists of the single stuck-at faults in the whole processor (268,424 faults).

The test program used was manually developed by a test engineer knowing the netlist of the processor, targeting the maximization of the stuck-at fault coverage for the whole processor when using processor level observation (S2). It is

written in assembly language, occupies 1,576 bytes and the execution duration is 19,298 clock cycles.

In order to compute the fault coverage that can be achieved resorting to the different observation mechanisms, a module wrapping the processor was added to provide all the elements required by them, exporting the different sets of observed signals as primary outputs of the wrapper during fault simulation (Fig. 1). Two different performance counters were implemented: one obtains the test duration using a timer register enabled at the end of the test to capture the value of a timer input *t_in* provided by the testbench (labeled S5*); the other (*Perf_cnt*) increments each time an incorrect branch prediction arises (S5). A set of internal signals produced by the execution stage of the pipeline was exported up to the wrapper interface to support solution S6. Finally, a second read port was added to the RAM memory to support the memory content observation mechanism, allowing to observe the memory content without interfering with the normal memory access operations performed by the test program. Fault simulation

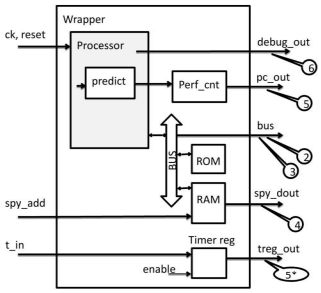

Fig. 1. Experimental environment and considered observation solutions.

was performed using Synopsys TetraMAX. The 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: *ck*, *reset*, *spy_addr* (to select the RAM location observed at *spy_dout*) and *t_in* (the timer signal provided by the testbench). At the end of the test program execution, *spy_addr* continuously scans the address interval selecting the RAM zone to be observed through *spy_dout*. A memory write operation added at the end of the test program triggers the Timer register to capture the final time value.

Fault simulation experiments were run using each of the different observation solutions shown in Fig. 1. Solutions S2 and S3 are identical in this case because all the outputs of the MIPS-like processor are system bus related. Solutions S4 to S5* can be implemented in-field by a few instructions at the end of the test program to read the final value of the observed entity and compare it with the expected one. Instead, in order to assess the fault coverage using a fault simulation tool like TetraMAX, the final value of the observed signal must be presented as a primary output of the top entity being simulated, i.e., the wrapper, and the fault simulator must be instructed to only observe this primary output, and to observe it only at the end of the experiment.

A summary of the signals observed by each solution follows:

- S2/S3: all the original processor outputs (memory data, address and control signals labeled as *bus* in Fig. 1) during the whole simulation.
- S4: outputs *spy_dout*, (while inputs *spy_addr* sweep the RAM area used for program results) at the end of test.
- S5: PeC value (*pc_out*), at the end of test.
- S5*: timer register output (*treg_out*) at the end of test.
- S6: the address (*instr_adr*) and opcode (*instr*) of the instruction being executed, the instruction being executed (*instr_adr_valid*) and instruction branch (*instr_bra*) flags, grouped as *debug_out* in Fig. 1, observed at cycles in which a branch instruction is executed in the non-faulty execution (golden run).

B. Results

To better assess each observation method strengths and weaknesses, the coverage was obtained for each internal module of the MIPS-like processor. TABLE I lists the internal modules along with the number of single stuck-at faults in it. The execution stage is by far the biggest module with about 60% of the faults, followed by the register bank (a 32x32 register bank with 2 read and 1 write ports) with about 17% and the Branch prediction unit with 10% of them.

In TABLE I the column labeled S2/S3 presents the fault coverage (detected/total module faults) obtained for each module using processor level observation, i.e., the one for which the test program was originally developed. This column shows that the test program best performances are obtained for the Execution unit (97,11%) and the Register bank (96,27%). These are the bigger modules and consequently have the bigger impact on the overall coverage. On the other hand, the poorer performances are for the Address calculation (51,34%) and Memory access (53,42%) stages and for the Branch prediction unit (57,49%), being the former the one that influences the most on overall coverage due to its greater size.

The next column labeled S4 presents the percentage of the faults detected by S3 which are also detected by S4. Note that as stated in [8], all the faults detected by S4 are also detected by S3. This column shows that in the modules that are well covered by S3, like the Execution stage and the Register bank, solution S4 covers almost all the faults covered by S3. It also behaves well in the Coprocessor system and in the connections between modules. However, it has a poor performance in modules like the Address calculation stage (36,8% of the faults covered by S3), the Branch Prediction unit (61,3%) and the Bus controller unit (72,7%).

The next three columns in TABLE I present the coverage obtained by S5, S5* and S6, also as a percentage of the number of faults detected by S3. But unlike the case of S4, methods S5, S5* and S6 include signals not observed by S3 and hence they allow the detection of faults uncovered by S3. As a consequence, the figures in these three columns can be greater than 100% as is the case of the interconnect row using observation method S6. The results show that these new methods are complementary with S4 as they perform better

TABLE I
RESULTS

per module stuck-at faults			FC	FC relative to S3				FC increment wrt S4				
	Faults		S2/S3	S4	S5	S5*	S6	S5	S5*	S6	all	all
	[#]	[%]	[%]	[%]				[# of faults]				[%]
u1_pf (Address calculation)	2,244	0.8	*51.34*	*36.8*	70.2	70.1	**99.0**	**490**	**489**	**716**	**716**	**31.9**
u2_ei (Instruction extraction)	1,876	0.7	72.87	88.3	85.8	85.5	96.1	5	5	113	113	6.0
u3_di (Instruction decoding)	8,468	3.1	71.76	90.5	81.1	79.5	95.1	255	250	**1,113**	**1,117**	**13.2**
u4_ex (Execution)	**165,876**	**61.8**	**97.11**	**99.7**	6.0	6.1	6.5	126	176	**860**	**936**	0.6
u5_mem (Memory access)	3,394	1.3	*53.42*	95.7	65.7	66.0	65.7	15	19	16	20	0.6
u6_renvoi (Bypass unit)	3,964	1.5	79.06	94.5	72.1	69.6	71.9	165	139	159	165	4.2
u7_banc (Register bank)	**45,274**	**16.9**	**96.27**	**99.9**	26.2	25.1	26.2	0	0	39	39	0.1
u8_syscop (System coprocessor)	8,524	3.2	63.96	**100.0**	1.0	1.0	1.0	0	0	0	0	0.0
u9_bus_ctrl (Bus controller)	1,422	0.5	75.67	*72.7*	41.8	43.0	44.1	6	16	30	40	2.8
u10_predict (Branch prediction)	**27,354**	**10.2**	*57.49*	*61.3*	**93.4**	**93.4**	**95.8**	**5,061**	**5,057**	**5,437**	**5,441**	**19.9**
interconnect (Interconnections)	28	0.01	25.00	**100.0**	**100.0**	**100.0**	**185.7**	0	0	6	6	21.4
whole_circuit	268,424	100.0	89.59	96.4	19.4	19.2	20.4	6,123	6,151	8,489	8,593	**3.2**

than S4 in modules like the Address calculation stage and the Branch prediction unit where the S4 coverage is bad.

Finally, the rightmost block in TABLE I shows, detailed by module, the coverage increment obtained by adding the faults detected by S5, S5*, S6 and all the three methods to those detected by the memory content observation method S4. The coverage increments are presented as the amount of faults. The rightmost columns repeats the "all" column values expressed as a percentage of the module faults. The newly covered faults can be identified as performance faults. The main contributions to the coverage enhancement come from the Branch Prediction unit, and from the Decode instruction, Execution and Address calculation stages. Looking at the whole circuit, the most important result is that the addition of the three methods to method S4 gives a coverage increment of 3.2% of the total faults and rises the total fault coverage to 89.59%, essentialy the same value obtained by S2.

IV. CONCLUSIONS

A set of observation methods was evaluated, aimed at obtaining a good coverage of performance faults in an in-field SBST scenario. The per module coverage results were presented and analyzed along with the results obtained with observation methods commonly used in end-of-manufacturing (S2) and in-field test scenarios (S4). The results comparison allows to identify about 3% of the total faults as performance faults contributing to overall coverage enhancement. We can also conclude that the new considered observation methods are an effective complement to the traditional memory content observation method S4, as they present very good coverage in the modules that are poorly covered by S4. This complementary behavior allows to obtain in the field a coverage similar to the one obtained using processor level observation S2.

The provided information can be fruitfully used from one side by the test engineer in charge of developing SBST code to chose the most suitable observation mechanisms, and from the other by the designer, who may possibly introduce some hardware structures to better support the SBST code development and increase the achieved Fault Coverage.

Moreover, the paper describes in detail how a conventional fault simulator can be used to compute the Fault Coverage achieved by a Self-Test Library.

REFERENCES

[1] M. Psarakis *et al.*, "Microprocessor Software-Based Self-Testing," *IEEE Des. Test Comput.*, vol. 27, no. 3, pp. 4–19, may 2010.

[2] https://www.hitex.com/tools-components/software-components/selftest-libraries-safety-libs/pro-sil-safetcore-safetlib/

[3] "Guidelines for obtaining IEC 60335 Class B certification for any STM32 application. Application note AN3307," STMicroelectronics, Tech. Rep. April, 2013.

[4] "Cypress AN204377. FM3 and FM4 Family, IEC61508 SIL2 Self-Test Library," Cypress, Tech. Rep., 2014.

[5] https://www.renesas.com/eu/en/products/synergy/software/add-ons.html

[6] "Microchip. 16-bit CPU Self-Test Library User ' s Guide," Microchip, Tech. Rep., 2012.

[7] https://developer.arm.com/technologies/functional-safety

[8] J. Perez Acle *et al.*, "Observability solutions for in-field functional test of processor-based systems: a survey and quantitative test case evaluation," *Microprocess. Microsyst.*, vol. 47, no. PB, pp. 392–403, 2016.

[9] M. Hatzimihail *et al.*, "A methodology for detecting performance faults in microprocessors via performance monitoring hardware," in *2007 IEEE Int. Test Conf.* IEEE, 2007, pp. 1–10.

[10] N. Foutris *et al.*, "Measuring the performance impact of permanent faults in modern microprocessor architectures," in *2013 IEEE 19th Int. On-Line Test. Symp.* IEEE, jul 2013, pp. 181–184.

[11] B. Du *et al.*, "FPGA-controlled PCBA power-on self-test using processor's debug features," in *2016 IEEE 19th Int. Symp. Des. Diagnostics Electron. Circuits Syst.* IEEE, apr 2016, pp. 1–6.

[12] E. Sanchez and M. S. Reorda, "On the Functional Test of Branch Prediction Units," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 23, no. 9, pp. 1675–1688, sep 2015.

[13] G. Theodorou *et al.*, "Software-Based Self Test Methodology for On-Line Testing of L1 Caches in Multithreaded Multicore Architectures," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 21, no. 4, pp. 786–790, apr 2013.

[14] ——, "Software-Based Self-Test for Small Caches in Microprocessors," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, vol. 33, no. 12, pp. 1991–2004, dec 2014.

[15] H. O'Keeffe, "IEEE-ISTO 5001 TM-1999, The Nexus 5001 Forum TM Standard providing the Gateway to the Embedded Systems of the Future," Ashling Microsystems Ltd., Tech. Rep. January, 2000.

[16] N. Stollon, *On-Chip Instrumentation Design and Debug for Systems on Chip*. Springer, 2013, vol. 53, no. 9.

[17] https://www.arm.com/products/system-ip/coresight-debug-trace

[18] B. Du *et al.*, "Online Test of Control Flow Errors: A New Debug Interface-Based Approach," *IEEE Trans. Comput.*, vol. 65, no. 6, pp. 1846–1855, jun 2016.

[19] "miniMIPS." http://opencores.org/project,minimips