Constraint-based Test-scheduling of Embedded Microprocessors

Nikolaos Bartzoudis
Centre Tecnològic de Telecomunicacions de Catalunya
Av. Canal Olímpic S/N, 08860,
Castelldefels – Barcelona, Spain
nikolaos.bartzoudis@cttc.es

Vasileios Tantsios and Klaus McDonald-Maier
Department of Computing and
Electronic Systems, University of Essex,
Colchester, CO4 3SQ, UK
Vasileios.Tantsios@eu.sony.com
kdm@essex.ac.uk

Abstract— **Test scheduling is a key aspect in the automation of embedded microprocessors self-testing. This paper presents a self-testing framework targeting the LEON3 embedded microprocessor with built-in test-scheduling features. The proposed design exploits existing postproduction test sets, designed for software-based testing of embedded microprocessors. The framework also includes a constraint-based approach of test-routine scheduling. The initial results show that the test execution time could be dynamically scaled by the test selection algorithm. The scheduler itself adds insignificant overheads in terms of execution cost and code size.**

I. INTRODUCTION

Software-based self-test (SBST) [1] and constraint-aware scheduling of testing patterns has attracted the attention of various researchers over the recent years. Efficient test scheduling minimizes the overall system test application time, prevents test resource conflicts and limits power dissipation during test mode.

The author in [2] tries to minimize test application time within a System-on-Chip (SoC) while considering structural resource allocation. Zhao et al present an algorithm for solving the general test scheduling problem where multiple test sets are selected [3]. Another inspiring concept in power-aware test scheduling is presenting in [4]. Zhow et al. propose a structural SBST methodology optimized for energy, average power consumption, test length and fault coverage [5]. Nourani et al. focuses on the use of power profile of non-embedded cores within a SoC platform to find the best mix of their test pattern subsets that satisfy the power and/or time constraints [6]. A cost-effective approach to the construction of diagnostic software-based test sets for microprocessors is presented in [7]. The most closely related work to our implementation is presented in [8]; a set of test routines from different test approaches are composing a test program for an embedded processor optimized for memory usage and real time requirements of the application.

This paper presents a test selection algorithm which not only ensures the periodical execution of the test, but also optimizes the test process considering the real-time requirements of the application and the execution cost. The testing framework uses functional testing and pass/fail techniques. Moreover, the proposed task scheduling is generic and could be reused across different hardware platforms or testing strategies. That is due to its extendible structure, which may include additional scheduling parameters (i.e. power consumption) on top of the test execution time that is currently considered.

II. THE TESTING FRAMEWORK

The target embedded microprocessor, LEON3 [9], is a synthesizable VHDL model of a 32-bit processor. LEON3 is part of the GRLIB IP library which is a set of reusable IPs centered on a common on-chip bus. A part of the GRLIB IP library could be simulated with the TSIM simulator [9]. TSIM is able to emulate LEON-based computer systems and a number of GRLIB IP cores through loadable modules. The self-testing framework and the test scheduling algorithm were implemented and debugged using the GNU-based cross-compilation tool-chain for the LEON3 processor.

The online testing of LEON3 was accomplished by testing its arithmetic operations and memory transactions. Two different test sets were used, to achieve efficient functional testing of embedded microprocessors. The final formation of the testing framework included 30 different tests.

A. Testing of arithmetic operations

The selected testing suite, namely "paranoia", applies arithmetic operations that test the FPU of embedded microprocessors for functional errors [10]. "Paranoia" comprises of a series of tests on arithmetic calculations (e.g. radix precision and range, consistency of comparison, presence of guard digits, underflow/overflow, square root and powers etc). An indicative representation of a test sequence is shown in fig. 1. Initially, the radix and the precision of the system are estimated. Next, the closest relative separation between two floating point numbers is calculated. Basic operations with some special numbers follow; floating point numbers have a limit in the precision of decimal numbers, thus, a small ulp is noticed between the exact value of the real number and the one that is calculated. Finally, the difference between the calculated value and the "correct" one is

The research of the authors N. Bartzoudis, V. Tantsios and K.D. McDonald-Maier was supported in part by the EPSRC grant EP/C54630X/1; by the Catalan Government under grant 2005SGR-00690; by the Ministerio de Industria Turismo y Comercio of the Spanish Government under project 2A103 (MIMOWA) from MEDEA+ program (PROFIT FIT-330225-2007-2), and by the European Commission under projects NEWCOM++ (ICT 216715) and PHYDYAS (ICT 211887).

measured. If the difference is equal to Radix to the power of the closest relative separation (21.1102230e-16), then the conclusion is that no errors appear in these kinds of operations. This test sequence is repeated four times in order to evaluate the impact of the rounding errors in the precision of the system.

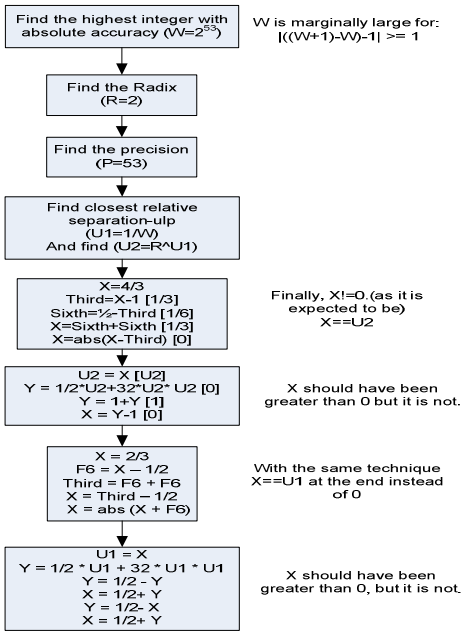

Figure 1. Estimating the radix, the precision and the rounding errors.

B. *Memory testing*

The memory testing suite uses three separate sub-tests (i.e. data bus test, address bus test and memory space test). The "shifting digit" technique is applied in all three test scenarios.

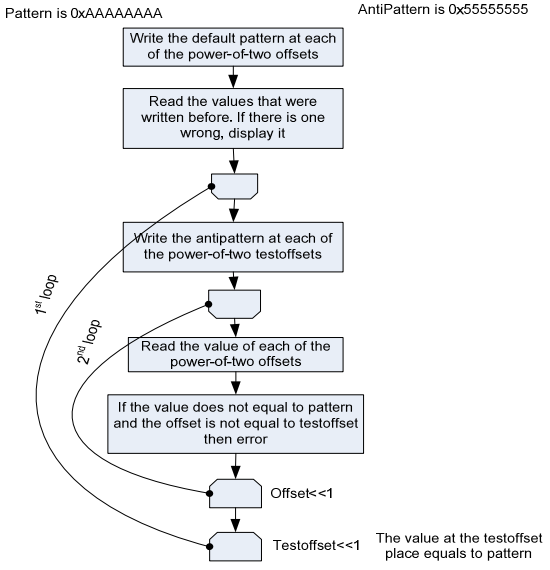

Figure 2. The test sequence of the memory address bus.

Fig. 2 shows the workflow diagram for the address bus test. Initially, the pattern value is written to each of the "power of two" offset. In the next step these values are read. At this point an additional variable is introduced (i.e. testoffset) which helps to define the memory address. The testing sequence continues with the following processing steps:

- The values of offset and testoffset are set to 1.
- The value at the testoffset address is changed. The anti-pattern is written.
- All the "power of two" offset values are read.
- The testoffset value shifts left; before that the pattern value is written again to the changed address. Thus, all the "power of two" addresses store the pattern value. The process is re-initiated.

C. *Overview of system design*

The implemented online self-testing framework comprises of three major structural components: the test pattern sets, the test selection algorithm and the testing evaluation. The execution of a test program may be initiated during system startup/shutdown. Alternatively, an OS scheduler may identify idle cycles and issue test program execution or this could be achieved in regular time intervals with the aid of programmable timers. Finally, the test programs could also be triggered manually by a user input.

The testing framework can be dynamically modified according to priorities related to system parameters. The test scheduling algorithm presented in this paper has the inherent ability to prioritize the test routines according to the available execution time. However, it may easily be extended to account for other system metrics.

The implemented online self-testing framework follows a structural strategy with pass/fail indications. The errors were classified in four categories according to their impact to the system (e.g. failures, serious defects, defects, flaws). The number of the detected errors together with the functional severity assessment, indicate the quality and health-status of the system.

III. TEST SCHEDULING

The test selection algorithm (TSA) is a complex scheduling engine, that tries to obtain the best execution order of test routines, given timing constraints (in terms of allowed time for test), while guaranteeing that conflicting tests cannot be executed concurrently. More constraints could be considered since the TSA has a generic programmable structure. The time overhead of each test routine is defined by simulating them offline.

A brief description of the algorithm with details of the processing steps is summarized next (fig. 3). The full extent of the cross comparisons is omitted for simplicity reasons (i.e. condition-checking to prevent TSA from being trapped in an infinite loop). Three lists are initially created, comprising of double linked nodes with three parameters (i.e. the name of the test, the execution time of the test, the times that were already executed). List 1 stores the tests, sorted by their execution time. List 2 stores the tests, sorted by the times they were already executed. List 3 is empty during initialization. It is used later to store the tests that were executed during the current execution cycle, preventing them from being executed again during the same execution cycle. Next, "Mergesort" is

applied to List 2. The test that is placed in the first position of List 2 is executed. If the execution time of the selected test is less than the total available time, then a) the total available time is reduced by the selected test's execution time, b) the times that the selected test was executed are increased by 1, c) the test is stored in List 3 in order not to be executed again in this execution cycle.

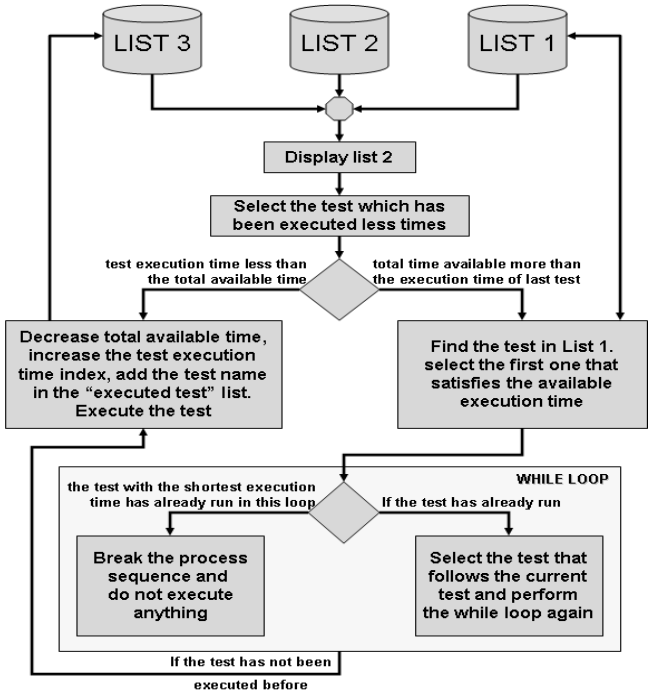

Figure 3. The test selection algorithm.

A more detailed example of the functionality of the TSA is described next using the data shown in table 1. We assume for simplicity reasons that the test set consists of 8 tests and the available time is 200 ms.

- List 2 sorts the test according to the number of times these were already executed.
- Test 6 is initially selected to be executed. Its execution time is less than the available time, thus, it is executed. The total available time will be 200-40=160ms and the times that test 6 was executed will be 3 leading test 6 to the bottom of List2.
- Test 3 is selected next. Since its execution time is less than the available time, test 3 will be executed. The total available time will be 160-60=100ms and the times that test 3 was executed will be 3 leading test 3 to the bottom of List2.
- Then, test 4 is selected. Since its execution time is less than the available time, test 4 will be executed. The total available time will be 100-60=40ms and the times that test 4 was executed will be 3 leading test 4 to the bottom of List2.
- Then, test 5 is selected. But, since its execution time is longer than the available time, test 5 will not be executed. The available time will remain 40ms.
- The TSA finds test 5 within List 1. The test that is then selected is the one with the highest execution time below the position of test 5. This is test 6.
- However, test 6, has already run during this execution cycle; thus, test 7 is selected.
- The total available time will be 40-30=10ms and the times that test 7 was executed will be 4 leading test 7 to the bottom of List2.
- Finally, the total available time that is left is less than the execution time of the test with the shortest execution time; thus the execution cycle stops.

TABLE I. AN EXAMPLE OF THE LISTS CREATED BY THE TSA.

LIST 1	Execution time (ms)	LIST 2	Times executed	LIST 2 update	Times executed
Test 1	100	Test 6	2	Test 5	2
Test 2	90	Test 3	2	Test 1	3
Test 3	60	Test 4	2	Test 8	3
Test 4	60	Test 5	2	Test 2	3
Test 5	50	Test 1	3	Test 6	3
Test 6	40	Test 8	3	Test 3	3
Test 7	30	Test 2	3	Test 4	3
Test 8	20	Test 7	3	Test 7	4

IV. RESULTS AND CONCLUSION

The effectiveness of the self-testing framework was validated by manually initiating errors. Error triggering in the memory was achieved by changing (offline) certain program variable values, while the microprocessor register values were altered at run-time (i.e. using the GDB debugger). Fig. 4 is a TSIM screenshot depicting part of the simulation process. As it may be seen no failures, serious defects or defects were detected (i.e. the detected flaw is due to compiler limitations).

```
tsim> load rep
section: .text, addr: 0x40000000, size 130832 bytes
section: .data, addr: 0x4001ff10, size 2568 bytes
read 387 symbols
tsim> go
resuming at 0x40000000
Testing the Microprocessor

I am in MAIN.

I am before 1st DO.

* * * Milestone10 * * *
-1, 0, 1/2, 1, 2, 3, 4, 5, 9, 27, 32 & 240 are O.K.

Searching for Radix and Precision.
Radix = 2.000000 .
Closest relative separation found is U1 = 1.1102230e-16 .

Recalculating radix and precision
 confirms closest relative separation U1 .
Radix confirmed.
* * * Milestone25 * * *
The number of significant digits of the Radix is 53.000000 .

* * * Milestone80 * * *

I am in J+++++++++++++++++++++++++++++++++++++++==7.

* * * Milestone220 * * *

The number of  FLAWs  discovered =           1.

The arithmetic diagnosed seems Satisfactory though flawed.
END OF TEST.

This program segment took 33623 milliseconds to execute.
Program exited normally.
```

Figure 4. The TSIM simulation environment.

The TSA was also validated with the TSIM simulator. The test-sets verified the arithmetic operations of the LEON microprocessor and its memory response while the testing framework including the TSA, was executed 39 times. The contents of List 2 are initially displayed in each execution cycle (fig. 5). The left column shows the name of the test, the

middle one shows the execution time of the test and the right column shows how many times each test was executed before the current execution cycle. The TSA is then printing the test sequence numbers for the current execution cycle and finally it reports the total available time that was not used. As it is shown in fig. 5 the different test routines were used exactly the same number of times after 39 execution cycles of the testing framework ensuring by this way high fault coverage.

```
I am in J++++++++++++++++++++++++++++++++++++++++==38.
17       120      11
7        130      11
5        80       11
14       110      11
15       100      11
19       70       11
2        140      11
12       150      11
4        80       11
16       90       11
18       60       11
8        20       11
9        30       11
13       160      11
3        170      11
0        180      11
10       190      11
6        50       11
1        200      11
############ Test 17 run ############.
############ Test 7 run ############.
############ Test 5 run ############.
############ Test 14 run ############.
############ Test 15 run ############.
############ Test 19 run ############.
############ Test 9 run ############.
available time left is  10.
I am in J++++++++++++++++++++++++++++++++++++++++==39.
 after
Program exited normally.
tsim>
```

Figure 5. Each test is executed 11 times.

An example that verifies that the TSA never runs the same test more than once during a given execution cycle, is illustrated in figure 6. The total available time left during the 36th testing execution cycle is 40ms. Tests 8 with execution time of 20ms and test 9 with execution time of 30ms could have been executed. However, these two tests were already executed in that particular execution cycle; therefore the execution sequence ends at that point.

```
I am in J++++++++++++++++++++++++++++++++++++++++==35.
3        170      10
13       160      10
9        30       10
8        20       10
18       60       10
16       90       10
4        80       10
1        200      10
6        50       10
10       190      10
0        180      10
19       70       10
15       100      10
14       110      10
5        80       10
7        130      10
17       120      10
12       150      11
2        140      11
############ Test 3 run ############.
############ Test 13 run ############.
############ Test 9 run ############.
############ Test 8 run ############.
############ Test 18 run ############.
############ Test 16 run ############.
############ Test 4 run ############.
##############DIDN'T RUN ANYTHING.
available time left is  40.
I am in J++++++++++++++++++++++++++++++++++++++++==36.
```

Figure 6. A test is not allowed to run two times at the same execution cycle.

The total execution time of all the tests is approximately 34sec. The execution time overhead that is added from the TSA could be considered minor since it is 350ms (i.e. 1% of the total execution time of the testing framework). The executions times of the test-sets and the TSA could be reduced significantly when implemented in a hardware platform.

The experimental results presented in this paper show that the TSA could optimize the overall system test application time, avoid test resource conflicts and decrease the execution cost, while ensuring that real-time system tasks will meet their deadlines. Moreover, the TSA can be tuned either for lower execution cost or higher error coverage; apparently, there is always a tradeoff between these two testing strategies. High error coverage could be achieved in a long term testing scenario by dynamically modifying the execution time of the framework and its execution frequency. The TSA is fully programmable and thanks to its extensible structure it could also be parameterized to consider different sorting conditions (i.e. sorting the test execution according to test importance).

V. EXTENDING THE TSA FUNCTIONALITY

Additional scheduling parameters could be integrated to the TSA to target other operational scenarios or testing strategies. The tradeoff for using extra cross-correlation parameters would be additional memory, performance and execution-time overheads for the TSA. The implemented testing framework can run in conjunction with the main processing activity of the LEON3. This could be further optimized by applying minor modifications to the existing implementation. The TSA could also be extended to include scheduling of VHDL-based build-in self test routines. Additional validation of the TSA at run time in a hardware platform will boost its performance and optimize its code size.

REFERENCES

[1] Janusz Sosnowski, “Software-based self-testing of microprocessors”, Journal of Systems Architecture, Vol. 52, No 5, pp. 257-271, May 2006.

[2] Škarvada Jaroslav, “Test Scheduling for SOC under Power Constraints”, in Proc. of IEEE Workshop on Design and Diagnostics of Electronic Circuits and Systems, Prague, 2006, pp. 91-93.

[3] D. Zhao and S. Upadhyaya, “Adaptive test scheduling in SoCs by dynamic partitioning,” in IEEE Int. Symp. Defect and Fault Tolerance in VLSI Systems, Nov. 2002, pp. 334–342.

[4] Richard M. Chou, Kewal K. Saluja and Vishwani D. Agrawal, “Scheduling tests for VLSI systems under power constraints”, IEEE Transactions on Very Large Scale Integration Systems, Vol. (5), No 2, pp. 175-185, Jun. 1997.

[5] J. Zhou and H. Wunderlich, “Software-based self-test of processors under power constraints”, in Proc. of Design, Automation and Test in Europe, pp. 430-435, Germany, March 06 - 10, 2006.

[6] Mehrdad Nourani and James Chin, “Power-time tradeoff in test scheduling for SoCs”, in Proc. of the 21st International Conference on Computer Design, pp. 548-553, Washington DC, Oct. 2003.

[7] P.Bernardi, E.Sanchez, M.Schillaci, G.Squillero, M.Sonza Reorda, “An Effective Technique for Minimizing the Cost of Processor Software-Based Diagnosis in SoCs”, in Proc. of the IEEE Conference on Design, Automation and Test in Europe, 2006, pp. 412-417, Germany, March 06 - 10, 2006.

[8] M. Moraes, É. Cota, L. Carro, F. Wagner and M. Lubaszewski, “A constraint-based solution for on-line testing of processors embedded in real-time applications”, in Proc. of the 18th Annual Symposium on integrated Circuits and System Design, pp. 68-73, Florianolpolis, Brazil, Sep. 2005.

[9] LEON3 - TSIM2 Simulator, Oct. 2007 (www.gaisler.com).

[10] Les Hatton, “Embedded System Paranoia: a tool for testing embedded system arithmetic”, Information and Software Technology, 47 (2005), pp. 555–563.