A Scalable Software-Based Self-Test Methodology for Programmable Processors

Li Chen*, Srivaths Ravi†, Anand Raghunathan†, Sujit Dey*

*Dept. of ECE, University of California at San Diego
La Jolla, CA
{lichen,dey}@ece.ucsd.edu

†NEC Laboratories America, Inc.
Princeton, NJ
{sravi,anand}@nec-labs.com

ABSTRACT

Software-based self-test (SBST) is an emerging approach to address the challenges of high-quality, at-speed test for complex programmable processors and systems-on chips (SoCs) that contain them. While early work on SBST has proposed several promising ideas, many challenges remain in applying SBST to realistic embedded processors. We propose a systematic scalable methodology for SBST that automates several key steps. The proposed methodology consists of (i) identifying test program templates that are well suited for test delivery to each module within the processor, (ii) extracting input/output mapping functions that capture the controllability/observability constraints imposed by a test program template for a specific module-under-test, (iii) generating module-level tests by representing the input/output mapping functions as virtual constraint circuits, and (iv) automatic synthesis of a software self-test program from the module-level tests. We propose novel RTL simulation-based techniques for template ranking and selection, and techniques based on the theory of statistical regression for extraction of input/output mapping functions. An important advantage of the proposed techniques is their scalability, which is necessitated by the significant and growing complexity of embedded processors.

To demonstrate the utility of the proposed methodology, we have applied it to a commercial state-of-the-art embedded processor (Xtensa™ from Tensilica Inc.). We believe this is the first practical demonstration of software-based self-test on a processor of such complexity. Experimental results demonstrate that software self-test programs generated using the proposed methodology are able to detect most (95.2%) of the functionally testable faults, and achieve significant simultaneous improvements in fault coverage and test length compared with conventional functional test.

Categories and Subject Descriptors

B.8.1 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance

General Terms

Reliability, Measurement, Experimentation, Algorithms

Keywords

Microprocessor, manufacturing test, at-speed test, software-based self-test, test program, scalability

1. INTRODUCTION

The IC industry has witnessed an ever-lasting tug-of-war among test methodologies such as functional test, scan test, and built-in self test (BIST) [1]. Scan test offers a systematic methodology with short turn-around times for high-complexity ICs. Functional test, on the other hand, is known to detect speed-defects and other untargeted faults, and continues to be relied upon for testing the performance of high-speed devices such as microprocessors, which cannot tolerate the performance degradation induced by the insertion of scan chains. While BIST addresses the issue of at-speed test, the hardware overheads are even higher than scan.

The key challenge in functional test is whether the process of generating high-coverage functional tests can be made scalable and applied at low turn-around times and non-recurring engineering (NRE) costs. The lack of scalability in traditional functional test is caused by the massive manual test writing effort required for generating high-coverage test programs. One approach to automate the test writing process is to use randomized instruction sequences [2][3][4]. Since the test generation process is not guided by any particular fault model, achieving high fault coverage may require a large number of instructions. For realistic processors, this translates to not only prohibitively long test application times, but also long fault simulation times for fault grading.

Linking instruction-level tests with low-level fault models, Software-Based Self-Test (SBST) has been introduced as a promising technique for testing high-performance microprocessors [5][6]. Based on a divide-and-conquer approach, SBST first generates test patterns for specific modules (sub-circuits) within the processor, targeting structural faults within the module. Processor instructions are then used as a vehicle for delivering the patterns to module inputs and collecting test responses from module outputs. The result is a test program consisting of processor instructions. During test application, SBST employs a self-test scheme wherein the processor simply executes the test program at-speed from the on-chip memory. A low-speed structural tester is used to load and unload the on-chip memory. The use of a similar self-test scheme has been recently reported on the Intel Pentium™ 4 processor [4]. Applications of SBST to the testing of path delay faults, interconnect crosstalk faults, and fault diagnosis have been developed in [7][8][9], respectively. An enhancement of SBST using deterministic tests for arithmetic modules has been studied in [10], whereas [11] focuses on the application of SBST to processor control subsystems.

SBST aims to generate high-coverage tests that can be applied at-speed using low-cost testers. It achieves this goal by combining the fault-driven nature of gate-level test generation and the at-speed test delivery mechanisms inherent in functional tests. Gate-level test generation is performed only for individual circuit blocks, avoiding scalability problems. A key requirement of SBST is that, since module tests must be delivered functionally using instructions, they must satisfy instruction-imposed constraints.

To make SBST a scalable solution in the face of increasing processor complexity, each of the above steps must be automated using efficient techniques. In general, previous approaches to perform the steps involved in SBST suffer from high complexity. For a large processor, it is virtually impossible to extract module-level constraints manually. Automated constraint extraction methods have been proposed in [12][13], in which constraints imposed by the hardware environment surrounding a module can be extracted from the RTL description. However, structural constraints cannot be directly used in SBST, as they are only a subset of instruction-imposed functional constraints. Moreover, from a practical point of view, the complexity of any structural-analysis-based constraint-extraction method increases drastically as the design complexity increases. An alternative approach is to extract constraints using formal verification techniques such as symbolic simulation [14]. Given extracted constraints, an automated test program synthesis method was proposed in [15] based on a backtrack-based search algorithm similar to that used in sequential ATPG.

1.1 Paper overview and contributions

While initial work on SBST has proposed several promising ideas, *realizing the potential of SBST requires a systematic methodology and automation tools*. In this work, we propose a comprehensive methodology for SBST that consists of scalable methods to automate the key steps. We identify the following key steps in-

This work was supported by the MARCO/DARPA Gigascale Silicon Research Center (GSRC) and a summer internship at NEC Laboratories America, Inc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2003, June 2-6, 2003, Anaheim, California, USA.
Copyright 2003 ACM 1-58113-688-9/03/0006... $5.00.

Authorized licensed use limited to: Southern Taiwan University of Science & Technology. Downloaded on February 08,2023 at 13:13:47 UTC from IEEE Xplore. Restrictions apply.

3.4 Test program synthesis

We developed a prototype tool in *Perl* to synthesize test programs given the test patterns generated using constrained ATPG. On the Sun Enterprise 250 server, the CPU time for generating the test program was 1.45 seconds. Table 3 shows the comparison between the synthesized software-based self-test program (column *SBST*) and a functional test program (column *Functional*). The functional test program was generated by enumerating all instructions and assigning pseudorandom values to their operands. For each instruction, a set of 128 pseudorandom operand assignments was used. Peripheral instructions were also added for controllability and observability purposes, as in the case of SBST.

In Table 3, Rows 1 and 2 show the program size in terms of the number of instructions and the number of bytes, respectively. This does not include the reset and termination code inserted by the compiler before and after the main program. Row 3 shows the execution time of the test program in terms of the number of processor cycles, again excluding the reset/termination code. Row 4 shows the CPU time taken for fault simulating the test program at the processor level (using *Mentor Flextest* on the Sun Enterprise 250 server). The reset/termination code is included here since it contributed marginally to the fault coverage. Row 5 shows the fault coverage measured on `EX1` after dropping all known functionally untestable faults from consideration (this is a lower bound on the actual fault coverage with respect to functionally testable faults).

Table 3. Comparison of SBST and functional test

	SBST	Functional
Program size [instructions]	7602	9169
Program size [bytes]	20373	25066
Program Execution time [cycles]	27248	41844
Fault simulation time [hours]	27.5	41.3
Fault cov. on func. testable faults	95.2%	85.3%

The program sizes of SBST and random functional test are close, even though random functional test exercises each instruction 128 times while SBST applies only 288 ATPG patterns over all instructions. The reason is that the random functional test was written manually in assembly and was thus more compact, whereas SBST programs were generated automatically using test program templates and a cross-compiler. Many instructions in SBST can in fact be collapsed together, resulting in much smaller test programs.

For SBST, the fault coverage exceeds the fault coverage projected by module-level constrained ATPG (90.1%) due to the addition of the reset/termination code, as well as the collateral coverage resulting from the peripheral instructions used for delivering the tests. Overall, the software self-test programs resulting from the proposed methodology achieve a coverage of at least 95.2% on functional testable faults.

It can be seen that SBST achieves a high fault coverage much faster than random functional test. This corresponds to a shorter test loading time, smaller memory requirement, shorter test application time, and a much shorter fault simulation time. In random functional test, processor-level fault simulation is required to evaluate the coverage of the test program. The fault simulation time can be prohibitively long due to the complexity of the processor and the length of the test program. SBST can reduce fault simulation time not only by reducing test length, but also by reducing design complexity: given accurate output constraints, module-level ATPG can be used to give an accurate projection of the fault coverage. In this case, processor-level fault simulation is only needed if one intends to evaluate the collateral coverage on other modules.

SBST enables the generation of functional tests in a deterministic manner. In the cases when pseudorandom tests must be used (*e.g.*, to reduce the on-chip memory required for storing deterministic patterns, or to detect unmodeled faults), SBST can be used in conjunction with random tests in the following ways: (a) In traditional functional test that uses randomized instructions, without any human knowledge of the architecture, a uniform instruction mix is usually used, resulting in inefficient test programs [2]. In SBST, the results of constrained ATPG can be used as a guideline for determining an efficient instruction mix for pseudorandom tests (*e.g.*, see Figure 6). (b) In scan test, random tests need to be topped-off with deterministic tests. In functional test, SBST provides, for the first time, a scalable mechanism for topping-off random tests with deterministic tests targeted at hard-to-detect structural faults.

The proposed methodology applies not only to stuck-at faults, but also to other fault models, such as bridging faults and transistor-level faults. Furthermore, the same simulation-based approach for extracting constraints can be extended for extracting multi-timeframe constraints, enabling deterministic test generation for performance-related faults at the functional level [7].

4. CONCLUSIONS

For today's high-speed microprocessors, functional test continues to be relied upon for catching speed defects undetected by scan tests. However, traditional functional test lacks scalability and cannot be used to target low-level structural faults. Software-based self-test (SBST) has been previously proposed as a promising approach for tackling this problem. In this work, we propose a comprehensive systematic methodology for SBST and automate its key steps.

We demonstrate the scalability of the proposed method by applying it to the Tensilica Xtensa™ embedded processor. Our experiments show that, at the module level, extracted instruction-imposed constraints are close to the true constraints, and that the test patterns generated under these constraints can detect most, if not all, functionally testable faults. Software self-test programs generated using the proposed methodology result in simultaneous improvements in test length and fault coverage compared with traditional functional test. We believe that the proposed SBST methodology is an important step towards realizing the potential of SBST for realistic programmable processors.

5. REFERENCES

[1] D. Wu et al., "Can scan achieve the quality level we are looking for?" Panel session, *Proc. Intl. Test Conf.*, Baltimore, MD, Oct 2002, pp. 1194-1199.

[2] J. Shen and J. A. Abraham, "Native mode functional test generation for processors with applications to self test and design validation," *Proc. Intl. Test Conf.*, Washington DC, Oct. 1998, pp. 990-999.

[3] K. Batcher and C. Papachristou, "Instruction randomization self test for processor cores," *Proc. 17th IEEE VLSI Test Symp.*, Dana Point, CA, April 1999, pp. 34 – 40.

[4] P. Parvathala, K. Maneparambil, and W. Lindsay, "FRITS – A microprocessor functional BIST method," *Proc. Intl Test Conf.*, Baltimore, MD, Oct 2002, pp. 590-598.

[5] L. Chen and S. Dey, "DEFUSE: A Deterministic Functional Self-Test Methodology for Processors," *Proc. 18th IEEE VLSI Test Symp.*, Montreal, Canada, May 2000, pp. 255-262.

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

[7] W.-C. Lai, A. Krstic, and K.-T. Cheng, "On testing the path delay faults of a microprocessor using its instruction set," *Proc. 18th VLSI Test Symp.*, Montreal, Canada, May 2000, pp. 15-20.

[8] L. Chen, X. Bai, and S. Dey, "Testing for interconnect crosstalk defects using on-chip embedded processor cores," *J. Electronic Testing: Theory and Applications*, vol.18, (no.4), August 2002, pp. 529-538.

[9] L. Chen and S. Dey, "Software-based diagnosis for processors," *Proc. 39th Design Automation Conf.*, New Orleans, LA, June 2002, pp. 259-262.

[10] N. Kranitis, D. Gizopoulos, A. Paschalis, Y. Zorian, "Instruction-based self-testing of processor cores," *Proc. 20th VLSI Test Symp.*, Monterey, CA, April 2002, pp. 223-228.

[11] S. Almukhaizim, P. Petrov, and A. Orailoglu, "Low-cost, software-based self-test methodologies for performance faults in processor control subsystems," *IEEE Custom Integrated Circuits Conf.*, San Diego, CA, May 2001, pp. 263-266.

[12] P. Vishakantaiah, J. Abraham, and M. Abadir, "Automatic test knowledge extraction from VHDL (ATKET)," *Proc. 29th Design Automation Conf.*, Anaheim, CA, June 1992, pp. 273-278.

[13] R. Tupuri, A. Krishnamachary, and J. Abraham, "Test Generation for Gigahertz Processors Using an Automatic Functional Constraint Extractor," *Proc. 36th Design Automation Conf.*, New Orleans, LA, June 1999, pp. 647-652.

[14] W.-C. Lai, *Embedded Software-Based Self-Test for System-on-a-Chip Design*, PhD thesis, Univ. California at Santa Barbara, March 2002.

[15] W.-C. Lai, A. Krstic, and K.-T. Cheng, "Test program synthesis for path delay faults in microprocessor cores," *Proc. Int. Test Conf.*, Atlantic City, NJ, Oct. 2000, pp. 1080-1089.

[16] R. Tupuri and J. Abraham, "A Novel Functional Test Generation Method for Processors using Commercial ATPG," *Proc. Intl. Test Conf.*, Washington, DC, Nov. 1997, p.743-752.

[17] W.N. Venables and B.D. Ripley, *Modern Applied Statistics with S-PLUS*, Springer-Verlag, 1998.

[18] http://www.tensilica.com/xtensa_overview_handbook.pdf, *Xtensa™ Microprocessor Overview Handbook*, Tensilica Inc, August 2001.

[19] Design Compiler™, Synopsys Inc., http://www.synopsys.com

[20] Modelsim™, Model Technologies Inc., http://www.model.com

[21] FlexTest™, Mentor Graphics Corp., http://www.mentor.com

Authorized licensed use limited to: Southern Taiwan University of Science & Technology. Downloaded on February 08,2023 at 13:13:47 UTC from IEEE Xplore. Restrictions apply.

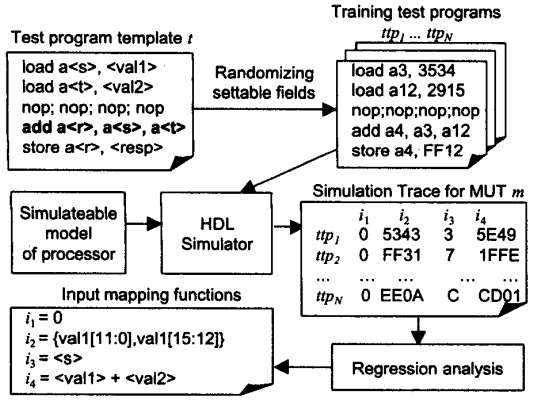

Figure 3. Deriving input/output mapping functions

processor in the input logic cone of the MUT is one instance of the mapping function. In practice, a close approximation to this mapping may be derived without any loss of accuracy (fault coverage), while resulting in significant simplification. We next show how to derive highly compact, yet accurate mapping functions using the theory of statistical regression.

Regression analysis refers to the process of determining a function that best fits a set of data observations obtained [17]. In order to accommodate the potentially wide range of mapping functions that may occur in practice, we attempt to derive mapping functions at both the *word-level* and the *bit-level*. Deriving a word-level mapping function involves expressing the input of a MUT (I) as a function of variables $X_1 \ldots X_m$, which may represent the settable fields in the test program template, as well as polynomial terms involving the settable fields. In other words, we express a mapping function for I as follows.

$$I \sim \alpha_0 + \alpha_1.X_1 + \alpha_2.X_2 + \ldots\ldots + \alpha_m.X_m \qquad [3]$$

Regression analysis tools that are widely available [17] can be employed to derive estimates of the coefficients in the mapping function ($\alpha_0, \alpha_1 \ldots \alpha_m$). Regression is performed using the data values for X_i's in the training test programs and the corresponding values for I in the HDL simulation trace, while minimizing the error of the fit.

Consider, for example, the MUT input i_4 shown in Figure 3. The general mapping function seen in Equation 3 is defined to include all first-order and second-order polynomial terms involving the settable fields (*<val1>, <val2>, <val1>*<val2>, <val1>^{2}, <val2>^{2}*). Figure 3 shows that, for MUT input i_4, the desired mapping function determined by regression is *<val1> + <val2>*.

While the use of regression analysis is effective in establishing word-level correspondences, it fails when the actual mapping function is a relational operator, Boolean function, or involves bit-level manipulation of the settable fields. Since the space of such functions is exponentially large in the cumulative number of bits in the settable fields, we use a pre-defined library of candidate binary mapping functions (that output 1/0) defined over the settable field input space. These functions cover all the standard relational operators between any two settable fields ($<, >, =, \neq, \geq, \leq$), and, all Boolean functions defined over any two bits in the settable field space. For each input bit of the MUT, we consider each candidate library function, and compute difference between the values evaluated by the candidate function and the actual values returned in simulation. For example, if the simulation trace records values of *(1,1,1,1,0)* for $i_2[0]$ (the first bit of i_2), while the mapping function given by *(<val1>[0] AND <val2>[1])* evaluates to *(0,0,1,1,1)*, the difference in evaluated and expected outputs is captured by *(1⊕0, 1⊕0, 1⊕1, 1⊕1, 0⊕1) = (1,1,0,0,1)*. In other words, the likelihood of error in estimation due to the use of this mapping function is *0.6*. The candidate function associated with the least likelihood of error is returned as the bit-level mapping function for that input bit.

Bit-by-bit error estimates are also obtained for the word-level mapping function determined using regression. The mapping function that results in minimum error for a given bit determines the mapping function used for that bit. Since each bit is considered independently, in general, a mix of word-level and bit-level functions may be used even within a single word.

Similarly, output mapping functions can be derived to encapsulate the propagation of an error at the outputs of a MUT to observable locations *(e.g.*, registers that are stored to memory in the test program template). If an error appearing at a MUT output can propagate to an observable location, we consider this output to be observable. In order to decide the observability of a MUT output, we inject an X value at the MUT output during the HDL simulation of the training test programs to see if an X value propagates to the observable destination. Since this binary outcome (observable/otherwise) is again contingent on the values assigned to the settable fields in the template, we use the regr ession analysis techniques detailed above to determine the output mapping function.

2.4 Constrained test generation

Given instruction-imposed constraints, we perform constrained test generation based on the concept of *virtual constraint circuits* (VCCs) introduced in [16]. We propose a utilization of VCCs that not only enforces the instruction-imposed constraints (as abstracted by the mapping functions) during test generation, but also facilitates the translation from module-level test patterns to instruction-level test programs.

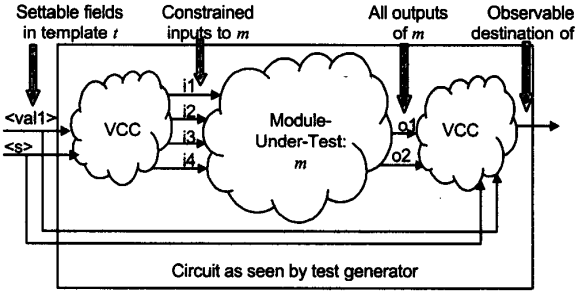

Figure 4. Constrained test generation using VCCs

In [16], VCCs were proposed to enable the generation of module-level tests under the constraints imposed by the *hardware environment* surrounding the MUT. We modified the concept of VCCs to enable the modeling of *instruction-imposed* constraints (Figure 4). To generate tests for MUT m under constraints imposed by test program template t, we first insert a VCC on the input side of the MUT. The generation of the VCC is automated, since it simply implements the mapping functions between settable fields in t and inputs of m (as described in Section 2.3). The constraints on the inputs of m are described implicitly by the mapping functions. If the mapping function for a particular input port is unknown, we wire it to X's (unknown values) in the VCC. This results in a conservative estimation of coverage during module-level test generation. In practice, we observed the loss of coverage due to unknown mapping functions to be small. The test generator is free to assign any patterns to the inputs of the VCC, which are the relevant settable fields in t. Enforced by the logic in the VCC, any patterns appearing at the inputs of the MUT are guaranteed to satisfy the instruction-imposed constraints. Similarly, on the output side of the MUT, we insert another VCC that embodies the output mapping functions.

During constrained test generation, the test generator sees the circuit including m and the two VCCs, as shown in Figure 4. The goal of test generation is to detect faults in m. Thus, faults from the VCCs are removed from the fault list during test generation. The patterns generated by the test generators are directly in terms of values assigned to the settable fields in t. Thus, they can be easily translated into test programs, as will be seen in Section 2.5.

2.5 Test program synthesis

Given test patterns $P_{m,t}$ generated for MUT m under the constraints imposed by test program template t, Figure 5 shows the generation of the corresponding test program, $TP_{m,t}$.

In Step 1, we identify values assigned to settable fields in t from the test patterns produced by the test generator. These are the

Authorized licensed use limited to: Southern Taiwan University of Science & Technology. Downloaded on February 08,2023 at 13:13:47 UTC from IEEE Xplore. Restrictions apply.