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.

volved in generating a software self-test program: (1) constructing instructions or instruction sequences (*test program templates*) that can be used as delivery mechanisms for a given module-under-test (MUT), (2) *extracting constraints* imposed by instructions on the MUT, (3) performing *module-level test generation* subject to the said constraints, and (4) translating module-level test patterns to software self-test programs.

For the selection of high-quality test program templates, we propose novel techniques based on functional (RTL) HDL simulation to predict the controllability and observability that a given template will provide for a specified module in the processor. Constraint extraction is performed in our methodology through the derivation of input/output mapping functions for the MUT. We have developed efficient techniques based on the theory of statistical regression to formulate mapping functions based on traces derived from functional simulation. Unlike conventional constraint extraction techniques, which are based on structural analysis [12][13], our techniques require only a simulatable functional model of the processor, impose no restriction on the HDL coding style, and are fully automated without requiring a knowledge of the processor architecture. We perform module-level test generation under the extracted constraints using the concept of virtual constraint circuits (VCCs), which was first proposed in [16]. We propose a novel utilization of VCCs such that the test patterns generated can be directly plugged in to the settable fields in the test program template, greatly simplifying the automated generation of test programs. To demonstrate the utility of the proposed methodology, we have applied it to a commercial state-of-the-art embedded processor (the Xtensa™ from Tensilica Inc.) with promising results. *To the best of our knowledge, this is the first practical demonstration of SBST on a processor of such complexity.*

The rest of this paper is organized as follows. In Section 2, we outline the overall SBST methodology and describe the automation of its key steps. The experimental methodology and results are presented in Section 3. We summarize our ideas and draw conclusions in Section 4.

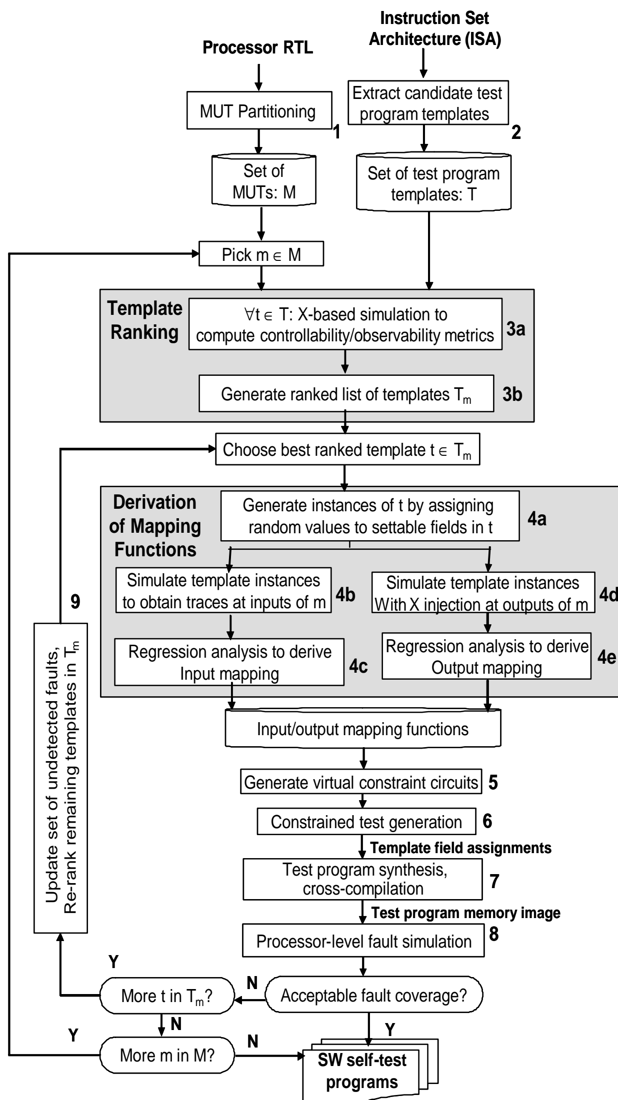

Figure 1. Overview of the proposed software-based self-test methodology

2. METHODOLOGY

Figure 1 shows the proposed SBST methodology, in which a self-test program can be automatically generated from only (a) a simulatable RTL description of the processor, and (b) the instruction set architecture (ISA) specification.

The first step in the proposed methodology involves partitioning the processor into a set (M) of modules-under-test (MUTs). This partitioning is done automatically through logic cone tracing techniques. While MUTs can be arbitrary sub-circuits, we choose combinational logic blocks as MUTs so that module-level constrained test generation can be easily handled by commercial ATPG tools. Step **2** systematically constructs a comprehensive set of test program templates (T), which will be used for the rest of the analysis. *Test program templates* (described in Section 2.1) are sequences of instructions used for test delivery, with fields in the instructions left settable whenever possible (*e.g.*, source/destination registers, data operands, *etc.*). An *instance* of a template is created by assigning specific values to the settable fields.

For each module $m \in M$, steps **3a-b** determine the test program templates that are best suited to deliver tests to m. Step **3a** computes a controllability/observability based testability metric $M_{m,t}$ (for each template $t \in T$), using a novel X-based simulation approach described in Section 2.2. Next, step **3b** generates a ranked list of templates (T_m) for module m, based on the metric $M_{m,t}$.

The best template t is removed from the ranked list T_m and considered for further analysis in Steps **4–7**. The objective of these steps is to construct a software self-test program using instances of template t that maximizes the detection of faults in m. Step **4** enables efficient test generation by abstracting the rest of the processor (all logic other than the MUT) through the use of mapping functions. *Mapping functions compactly capture the behavior of the processor for the given template. Input mapping functions* express the relationship between settable fields in the test program template and inputs of the MUT. *Output mapping functions* specify the logical and temporal conditions under which the outputs of the MUT are observable.

We solve the challenging task of extracting the input/output mapping functions using novel techniques based on functional simulation and statistical regression analysis (Step **4**). Step **4a** generates several instances of template t by assigning random values to the settable fields. Steps **4b-c** extract the input mapping function by (i) simulating the template instances to obtain traces at inputs of the module m, and (ii) using regression analysis to construct the mapping function that captures the relationship between the settable fields in t and the inputs of m. It bears mentioning that, the use of regression allows us to take a black-box approach that is agnostic to the actual physical structure (implementation) of the processor. Further, we circumvent the computationally hard problem of abstracting exact mapping functions from the structural description of a processor by designing *statistically correct mapping functions*. Steps **4d-e** similarly determine the output mapping functions, by injecting X values at the MUT outputs and observing the propagation (if any) to the specified template destinations.

The mapping functions determined in Step **4** are converted into virtual constraint circuits (Step **5**). Module-level test generation is performed on a composite circuit that consists of the MUT sandwiched between the input and output virtual constraint circuits (Step **6**). The test vectors resulting from Step **6** specify the values of settable fields in the test program template t, which are used in Step **7** to synthesize targeted software self-test programs for module m. The test programs thus generated are cross-compiled to a memory image, and fault simulated (Step **8**) to update the set of faults detected in the entire processor. If acceptable fault coverage on m has not yet been achieved, we update the ranked list of tem-

plates (Step **9**), and use the best-ranked template to target any remaining undetected faults in *m*. We repeat Steps **3-8** on remaining modules in the processor until satisfactory processor-level fault coverage is achieved, or all modules have been targeted.

We describe the details of the proposed methodology in the subsequent sub-sections. Section 2.1 describes the generation of candidate test program templates. Section 2.2 details the evaluation and ranking of templates. Section 2.3 formulates the derivation of input/output mapping functions. Sections 2.4 and 2.5 examine the tasks of constrained test generation and test program synthesis, respectively.

2.1 Extracting candidate test program templates

The state of a processor may be affected by an instruction that was executed an arbitrary number of cycles ago. Thus, it is virtually impossible to exhaust all possible test program templates, even if we are able to enumerate all permutations of instructions within a fixed number of cycles. Fortunately, not all templates are needed in order to achieve an acceptable coverage. Our goal is to generate a set of representative test program templates that can be used as basic building blocks for constructing effective self-test programs.

We classify test program templates into *single-instruction templates* and *multi-instruction templates*.

```
1 load a<s>, <val1>
2 load a<t>, <val2>
3 nop; nop; nop; nop
4 add a<r>, a<s>, a<t>
5 store a<r>, <resp>
```

(a)

```
1 load a<s>, <val1>
2 load a<t>, <val2>
3 load a<u>, <val3>
4 load a<v>, <val4>
5 nop; nop; nop; nop;
6 sub a<s>, a<u>, a<v>
7 add a<r>, a<s>, a<t>
8 store a<r>, <resp>
```

(b)

Figure 2. Test program templates

Single-instruction templates are built around one *key instruction* that is capable of delivering patterns to, and capturing responses from, the MUT. In a pipelined processor, this is the case when no pipeline forwarding is taking place. An example single-instruction template is shown in Figure 2(a), for which the key instruction is shown in bold on Line 4. The symbols enclosed in angled brackets ("<>") are defined as *settable fields*, whose values are to be set when the template is instantiated. The key instruction contains references to registers a<r>, a<s>, and a<t>, where <r>, <s>, and <t> are settable fields containing register indices. The template also contains *peripheral instructions* for aiding the execution of the key instruction. On Lines 1 and 2, two *load* instructions are used to set the contents of registers a<s> and a<t>, which are source registers in the key instruction. On Line 5, a *store* instruction is used to store the content of the destination register a<r> to a response location in memory, which is to be observed after the execution of the test program. We systematically construct a set of single-instruction test program templates by enumerating all instructions from the instruction set. To prevent pipeline forwarding, we precede the key instruction with a sufficient number of NOPs.

Multi-instruction templates can be constructed systematically in a similar manner, by including additional *supporting instructions* (*e.g.*, to trigger pipeline forwarding). An example multi-instruction is shown in Figure 2(b), where a supporting instruction is added on Line 6. The underlined field, a<s>, is responsible for triggering pipeline forwarding. Note that the relative position of the supporting instruction(s) with respect to the key instruction can be varied in different templates to cover all pipeline forwarding cases.

2.2 Template evaluation and ranking

As mentioned above, the space of all candidate templates is quite large. The objective of template evaluation is to efficiently short-list templates that are most promising for a given MUT. In order to facilitate efficient evaluation of test program templates, we developed a fast ranking mechanism that (a) uses X-based simulation to evaluate the controllability and observability that a template provides to the inputs (outputs) of a MUT, (b) ranks templates based on quantified testability metrics that capture the potential of a template to maximize fault coverage improvements, (c) removes templates that cannot serve as useful test delivery mechanisms, and (d) includes a dynamic update policy, that uses feedback from fault simulation to choose templates based on the current set of undetected faults in a module. The use of X-based simulation in the ranking formulation allows us to leverage off fast commercial HDL simulators, making the scheme applicable to any processor.

We now describe how the observability metric OBS_m^t *is* computed for a given module *m* and a template *t*. Let $O_1, O_2, O_3 \ldots O_n$ be the *n* output ports of a module *m* and $t_1, t_2, t_3 \ldots t_r$ be *r* instances of the template *t* obtained by assigning random values to the settable fields. Then, we construct an *r* x *n* Boolean matrix *F* as follows:

$F_{i,j} = 1$, if and only if, an 'X' value injected at O_j during the application of t_i gets propagated to an observable destination.

Thus, each row in the F matrix corresponds to a template instance, while each column corresponds to a module output port. By injecting an X value at O_j, when the key instruction of the template is active in the pipeline stage containing the MUT, we can monitor the subsequent propagation of the X value to an observable destination (*e. g.*, the general purpose register whose contents are subsequently written to the memory). Using the *F* matrix, we can formulate the observability metric OBS_m^t by observing that

- The potential of a template in observing O_j depends on the fraction of the *r* instances for which a 1 entry appears in the *F* matrix.
- The potential for improvement in fault coverage by observing output O_j depends on the number of undetected faults in the input logic cone of O_j (denoted by $S(O_j)$).

Equation 1 given below incorporates the above observations in defining OBS_m^t as the sum of the likelihoods of template *t* observing a response at O_j, weighted by the fault coverage improvement potential $S(O_j)$.

$$OBS_m^t = \sum_{j=1}^{n} \left(S(O_j) * \sum_{i=1}^{r} F_{ij}/r \right) \quad \textbf{[1]}$$

The metric for controllability ($CONT_m^t$) is similarly defined. Let $I_1, I_2, I_3 \ldots I_p$ be the *p* input ports of a module m and $t_1, t_2, t_3 \ldots t_r$ be *r* instances of the template *t* obtained by assigning an X value to one settable field, while assigning random values to the remaining settable fields. Then, we construct an *r* x *p* Boolean matrix *G* as follows:

$G_{i,j} = 1$, if and only if, an 'X' value assigned to any one settable field during the application of t_i gets propagated to the MUT input I_j.

Equation 2 given below defines the controllability metric $CONT_m^t$ as the weighted sum of the likelihoods of the template *t* controlling all the inputs I_j, weighted by the fault coverage improvement potential $S(I_j)$. Here, $S(I_j)$ denotes the undetected faults in the transitive fan-out cone extending from I_j to the outputs of *m.*

$$CONT_m^t = \sum_{j=1}^{p} \left(S(I_j) * \sum_{i=1}^{r} G_{ij}/r \right) \quad \textbf{[2]}$$

The product of $CONT_m^t$ and OBS_m^t yields the metric $M_{m,t}$ used to rank the remaining templates for a given module *m*. Note that this ranking is dynamically refined in Step **9** by updating $S(I_j)$ and $S(O_j)$ to include only those faults in module *m* that remain undetected after fault simulating the already derived test programs.

2.3 Deriving input/output mapping functions

Figure 3 illustrates the process of deriving the input mapping functions resulting from the use of a sample test program template *t* for a MUT *m*. Module *m* has four inputs: $i_1 \ldots i_4$ and the template has six settable fields (*<val1>,<val2>,<r>,<s>,<t>,<resp>).* Based on the template, we first construct a set of *N training test programs* ($ttp_1 \ldots ttp_N$) by assigning random values to settable fields. The training test programs should be distinguished from the final test program used for detecting faults, since they are only used for deriving the mapping functions. We simulate the execution of all training test programs on the processor using an HDL simulator. During the HDL simulation, we capture the simulation trace at the inputs of *m*, when the key instruction in *t* is active in *m*. For example, the simulation trace in Figure 3 shows that the sequence of values observed at MUT input i_2 for the training test programs ttp_1, ttp_2, … ttp_N is 5343, FF31, … EE0A (in hexadecimal format).

The next task is to determine the input mapping function that captures the relationship between the settable fields of the template and inputs of the MUT. The exact mapping may be a Boolean function of arbitrary complexity. For example, the entire logic of the

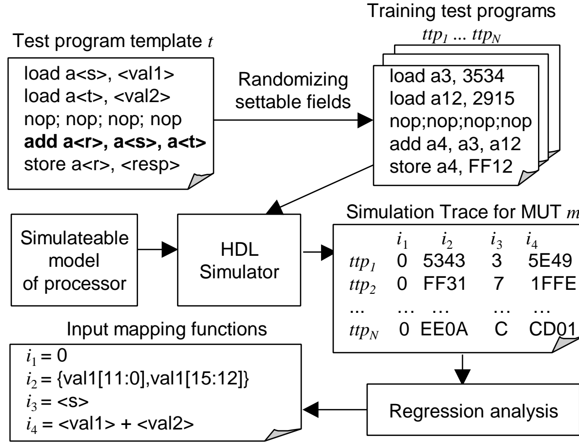

Figure 3. Deriving input/output mapping functions

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.

$$I \sim \boldsymbol{a}_0 + \boldsymbol{a}_1.X_1 + \boldsymbol{a}_2.X_2 + ... + \boldsymbol{a}_m.X_m \quad [3]$$

Regression analysis tools that are widely available [17] can be employed to derive estimates of the coefficients in the mapping function ($\boldsymbol{\mu}_0, \boldsymbol{\mu}_1 \ldots \boldsymbol{\mu}_m$). 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 (<, >, =, ≠, ≥, ≤), 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 *(1Å 0, 1Å 0, 1Å 1, 1Å 1, 0Å 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 regression 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.

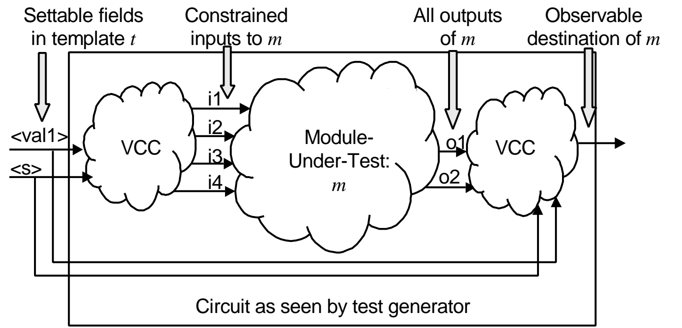

Figure 4. Constrained test generation using VCCs

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

settable fields affecting the inputs of *m*. In Step 2, we assign pseudorandom values to other settable fields in *t* in order to increase the collateral coverage in other MUTs. In Step 3, we parse the test program template *t*, identifying the positions of settable fields. In Step 4, we generate test program $TP_{m,t}$ by filling the values assigned to settable fields into their corresponding placeholders in *t*. The resulting test program $TP_{m,t}$ guarantees to apply test patterns $P_{m,t}$ to *m*.

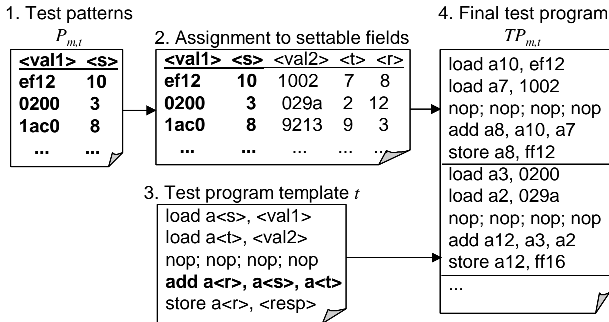

Figure 5. Test program synthesis

3. EXPERIMENTAL RESULTS

We applied the proposed SBST methodology to the Xtensa processor from Tensilica Inc.[18]. We present an overview of the processor in Section 3.1. The remaining subsections describe the experimental setup and results for the derivation of mapping functions, constrained test generation, and test program synthesis.

3.1 Overview of the Xtensa processor

The Xtensa is a commercial configurable and extensible RISC processor with 5 pipeline stages and 81+ core instructions [18]. A typical configuration of Xtensa, synthesized using Synopsys Design Compiler [19], contains 412574 collapsed stuck-at-faults and 5248 sequential elements.

We present experimental results in the context of a large logic module, called `EX1`, which was systematically extracted from the Xtensa processor. To extract `EX1`, we started from the 32-bit pipeline registers storing the results of the Execution stage (EX). We then traced the logic cones leading to these registers until we reached either a memory element or a primary input. The resulting logic is referred to as `EX1`. By construction, `EX1` has one 32-bit output port, and 81 input ports corresponding to 335 bits. `EX1` contains the basic building blocks for completing arithmetic/relational/logical operations, such as an adder, comparator, the AND/OR/XOR logic, and a funnel shifter. In addition, it contains numerous muxes for channeling data signals and control logic for interpreting control signals. Finally, it contains a set of muxes and control logic used for pipeline forwarding. The size of `EX1`, in terms of the number of faults, is 24962. `EX1` is particularly challenging due to its numerous data and control inputs, and due to the mix of datapath and control logic that it contains. In addition, the highly optimized hardware design and complex control logic of the Xtensa processor make it challenging to manually understand the link between instructions and the control signals of `EX1`.

3.2 Simulation-based mapping function extraction

By enumerating the core instructions in the Xtensa instruction set [18], we generated 42 test program templates, corresponding to 4 load instructions, 4 store instructions, 17 arithmetic/logical instructions, 7 move instructions, and 10 shift instructions. For each template, a pool of 128[1] training test programs (template instances) was created. Using the simulation framework provided in the Xtensa software suite [18], we simulated the execution of the training test programs using the commercial HDL simulation tool *ModelSim* [20]. On a SUN Fire 280R server with two 900-MHz UltraSPARC processors and 4GB RAM, the typical CPU time required for simulating all training test programs for one template is 37.8 seconds. For each training program, the simulation trace at the inputs of `EX1` was captured when the key instruction was in the EX stage. In case a key instruction was stalled in the EX stage, the capturing was performed after the stall cleared.

We first extracted the output mapping functions for `EX1`. According to the results of X-based simulation, the outputs of `EX1` can be observed in all but 8 test program templates, corresponding to the 8 load/store instructions. That is because the result of `EX1` is not used for load/store instructions. These 8 test program templates can thus be discarded.

Table 1. Mapping functions

	% bits
Constant	48.9%
One-to-one	31.0%
General	20.1%

For the remaining 34 test program templates, we extracted input mapping functions from the simulation trace. Table 1 shows the fraction of input bits of the MUT that are modeled using mapping functions that fall into the categories of *Constant* (the value of an input is constant due to the constraints imposed by the template), *One-to-One* (an identity, shift, rotate, or byte reordering relates a single settable field to a single MUT input), and *General* (more general functions were required).

3.3 Constrained test generation

For each template, we generated an RTL description of the VCCs in RTL *Verilog*, based on the extracted mapping functions. We synthesized the RTL description into a gate-level netlist using Synopsys *Design Compiler*. The gate-level description of the VCCs were then combined with that of `EX1` and given to a commercial ATPG tool (*Flextest* from Mentor Graphics [21]) for constrained test generation. On a SUN Enterprise 250 server with two 296-MHz UltraSPARC-II processors and 512MB RAM, the CPU time taken by ATPG was 283.79 seconds. The results of test generation, in terms of the number of faults detected and number of patterns generated, are shown in Table 2 and plotted in Figure 6. In Table 2, Column 1 shows the order in which the templates were considered. Column 2 shows the names of the templates. Columns 3 and 4 show the number of faults detected and the number of test patterns generated under each template. Note that the faults are disjoint, as faults detected under previously considered templates were dropped when a new template was being processed. Out of 24962 faults in `EX1`, a total of 18535 faults were detected by 288 test patterns.

Table 2. Results of constrained ATPG

	Temp	#Det	#Pat		Temp	#Det	#Pat		Temp	#Det	#Pat
1	add	5672	29	13	or	416	11	25	src_ssa8l	0	0
2	add.n	0	0	14	xor	1024	13	26	sll	175	5
3	addx2	211	7	15	moveqz	877	7	27	srl	0	0
4	addx4	153	4	16	movgez	0	0	28	sra	9	2
5	addx8	149	4	17	movltz	0	0	29	mov.n	0	0
6	addi	32	6	18	movnez	0	0	30	abs	53	8
7	addi.n	0	0	19	extui	3853	82	31	neg	0	0
8	sub	334	5	20	srli	1925	34	32	addmi	0	0
9	subx2	0	0	21	srai	1113	19	33	movi	288	4
10	subx4	0	0	22	slli	209	25	34	movi.n	166	6
11	subx8	0	0	23	src_ssr	17	5				
12	and	1859	12	24	src_ssa8b	0	0		Total	18535	288

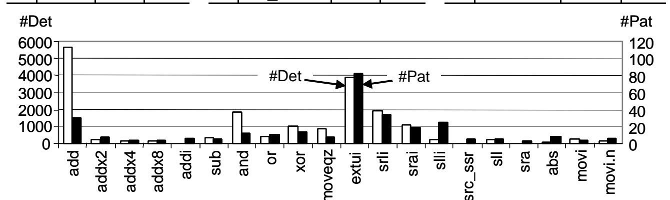

Figure 6. Results of constrained ATPG

The results of constrained ATPG correspond to a fault coverage of 74.3%. However, it is important to note that the coverage on functionally testable faults is much higher, as some of the remaining faults are functionally untestable. To identify functionally untestable faults, we performed constrained ATPG under a set of relaxed constraints that are independent of the selection of test program templates. In particular, `EX1` contains several multiplexers whose select signals must satisfy one-hot encoding constraints. The fault coverage obtained under these constraints is 81.8%, which is a loose upper bound of the functionally achievable fault coverage. Hence, a fault coverage of 74.3% for all faults translates into a fault coverage of at least 90.1% for functionally testable faults.

[1] The confidence in the accuracy of the mapping functions increases as the number of training test programs increases. In this case, we found 128 training programs to be sufficient for deriving correct mapping functions.

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