Microprocessor Software-Based Self-Testing

Mihalis Psarakis and Dimitris Gizopoulos
University of Piraeus

Ernesto Sanchez and Matteo Sonza Reorda
Politecnico di Torino

This article discusses the potential role of software-based self-testing in the microprocessor test and validation process, as well as its supplementary role in other classic functional- and structural-test methods. In addition, the article proposes a taxonomy for different SBST methodologies according to their test program development philosophy, and summarizes research approaches based on SBST techniques for optimizing other key aspects.

Ever-increasing market demands for higher computational performance at lower cost and power consumption continually drive processor vendors to develop new microprocessor generations. Every new generation incorporates technology innovations from different research domains, such as microelectronics, digital-circuit design, and computer architecture. All these technology advancements, however, impose new challenges on microprocessor testing. As device geometries shrink, deep-submicron delay defects become more prominent, thereby increasing the need for at-speed tests. Also, increases in the core operating frequency and speed of I/O interfaces necessitate more-expensive external test equipment. In addition, as multicore processor architectures become more popular, the time needed to test the chip scales with the number of cores, unless the inherent execution parallelism is exploited during testing.

These test technology challenges prompted the semiconductor industry during the past decade to consider new testing methods that can be incorporated in an established microprocessor test flow. The purpose of such methods is to achieve the target DPPM (defective parts per million) rate that high-quality product development demands, but without imposing excessive overhead in the test budget or interfering with the well-optimized, high-performance processor design. Such a testing method, which was introduced in the 1980s and has garnered renewed interest in the past decade, is functional self-testing. SBST has become more accepted for microprocessor testing, and it already forms an integral part of the manufacturing test flow for major processor vendors.

This article is the first attempt at classifying SBST approaches according to the way they generate self-test programs. Hence, we propose a taxonomy for the most representative SBST approaches according to their test program development philosophy. We also summarize research approaches for optimizing other key SBST aspects, and we highlight the potential of SBST for detecting and diagnosing faults at different stages of the microprocessor test and validation process.

SBST in the microprocessor test process

The key idea of SBST is to exploit on-chip programmable resources to run normal programs that test the processor itself. The processor generates and applies functional-test patterns using its native instruction set, virtually eliminating the need for additional test-specific hardware. (Test-specific hardware such as scan chains might exist in the chip, but SBST normally does not use this hardware.) Also, the test is applied at the processor’s actual operating frequency. A typical flow for an SBST application in a microprocessor chip comprises the following three steps (see Figure 1):

1. Test code and test data are downloaded into processor memory (i.e., either the on-chip cache or the system memory). A low-cost external tester can perform test code and data loading via a memory load interface running at low speed.
2. The processor executes test programs at its own actual speed. These test programs act as test patterns by applying the appropriate native instructions to excite faults. For example, in Figure 1 the test code...
loads two patterns and adds them to excite a fault in the adder module. Finally, the test code stores the test responses back in the processor data memory to propagate the faults. When the test process is supported only on the on-chip cache, the self-test program must be developed so that no cache misses occur during SBST execution—hence, this is sometimes called cache-resident testing.  

3. Test responses are uploaded into the tester memory for external evaluation.

The renewed interest in SBST during the past decade was primarily motivated by the existence of two opposite trends: the increasing cost of functional testers was impelling vendors toward structural-test techniques such as scan, while the doubts about the effectiveness of the structural-test patterns and the significant yield loss due to overtesting was moving them toward functional testing. Consequently, the emerging approach of SBST (in which normal, functional-test programs use low-cost structural testers to access the microprocessor chip) gained ground as a way to improve processor testing by combining the benefits of the functional- and structural-testing worlds. The question is whether a test program running on the processor can adequately test its modules by satisfying the industry-standard high-fault-coverage requirements. Achieving this test-quality target requires a composite test-program-generation phase, which is the main subject of most SBST approaches described in the literature during the past decade.

The other SBST features that confirmed its role in the microprocessor test flow include the following:

- **It is nonintrusive.** SBST does not need any extra hardware, which sometimes could be unacceptably expensive for carefully optimized circuits such as microprocessors. Moreover, it does not consume any extra power compared to normal-operation mode.
- **It allows at-speed testing.** Test application and response collection are performed at the processor's actual speed, enabling screening of delay defects that aren't detectable at lower frequencies.
- **It avoids overtesting.** SBST avoids test overkill and, thus, detection of defects that will never be manifested during normal processor operation. This leads to significant yield gains.
- **It can be applied in the field.** Self-test programs from manufacturing test can be reused in the field throughout the product lifetime—for example, for power-up diagnostics to add dependability features to the chip.

SBST approaches do not aim to substitute other established functional- or structural-test approaches but rather to supplement them by adding more test quality at low cost. This objective conforms to current industrial practice for the high-volume manufacturing (HVM) test process of microprocessors. Specifically, today's HVM test flow incorporates both functional test approaches using high-speed, high-pin-count testers, and structural test approaches (scan-based test and hardware self-test) using low-speed structural testers—thus reaching the best trade-off between DPPM, test application time, and overall test cost.

Figure 2 shows the typical representation of an HVM test flow, which includes three main phases: wafer test (which occurs before the chips are diced and packaged), package test (which involves individually testing chips after packaging), and speed binning (which determines the speed grade of the chip).
complements structural-test methods during wafer test, where functional testing cannot be performed. Second, it is employed after packaging, along with all other test methods, to achieve high-defect coverage. Speed binning typically involves functional testing with high-speed testers, but some recent approaches have also investigated using structural techniques with low-speed testers.

Chang et al. has claimed that even if scan transition tests can run only at low speeds, they might be useful during speed binning to screen both low- and high-speed defects. Tripp, Picano, and Schnarch have introduced drive-only functional testing (DOFT), which enables functional testing—particularly for speed classification—at lower cost. According to this approach, the ATE drives only the functional stimulus to the processor, whereas the test responses are either accumulated or checked on the die. Although this approach uses DFT circuitry to accumulate test responses, a software-based method could also fulfill the same purpose.

Table 1 summarizes the basic features of the three different test methods: structural testing (scan-based test with low-cost testers), functional testing (with “big-iron” functional testers), and SBST. As the table illustrates, the well-established scan-based-test and functional-test methods cannot be replaced in the current test flow: The former is a mature, fully automated method supported by sophisticated EDA tools that achieve very high fault coverage for common structural-fault models. The latter provides good collateral fault coverage and detects failures that scan-based test does not.

Self-test programs can be developed that target structural-fault models and that can be applied in native functional mode. Hence, SBST enhances the test-quality improvement provided by the other two approaches. Moreover, unlike functional testing, SBST does not require expensive test equipment and, unlike scan-based test, does not interfere with the design. Furthermore, SBST programs can be reused at different stages of the chip verification, validation, and test phases. Finally, scan-based test and functional testing require external test equipment, and their use ends after the chips are shipped to the customers. But SBST can be used in the field, either offline or periodically online.

The efficiency of an SBST approach depends on its test program development methodology and the effectiveness of the generated test routines. From Table 1, it is clear that another key success parameter is the automation of the self-test program generation process to reduce test development cost and to help quantitatively evaluate test routines for common
industrial structural-fault models. Thus, a main objective of many recent SBST research efforts is to automate the development of SBST programs.

Taxonomy of SBST approaches

To better describe the contributions from the most relevant papers on SBST, we propose a taxonomy tree. We address the most representative works of all active research teams during the past decade that reflect the basic ideas and directions in SBST research. When different papers describe successive versions of a method, we consider only the most recent one (which is typically the most comprehensive). (The “Typical Open-Source Microprocessor Benchmark Models” sidebar summarizes the most typical processor benchmark models used in open SBST research and their basic characteristics. (Only open-source processor models are included.)

# Typical Open-Source Microprocessor Benchmark Models

The academic software-based self-testing (SBST) approaches summarized in this survey article have demonstrated their effectiveness by presenting experimental results for different open-source microprocessor benchmark models. These processor models range from small and low-performance microcontrollers; to advanced RISC (reduced-instruction-set computer) processors with sophisticated performance mechanisms; all the way up to real, industrial multithreaded chip multiprocessors. Table A summarizes the most typical processor benchmark models used in open SBST research and their basic characteristics. (Only open-source processor models are included.)

<table>
<thead>
<tr>
<th>Processor model</th>
<th>CPU type</th>
<th>ISA</th>
<th>Other features</th>
<th>Gate count</th>
<th>HDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parwan</td>
<td>8-bit accumulator-based</td>
<td>Simple ISA, 17 instructions</td>
<td>No pipeline</td>
<td>2,000</td>
<td>VHDL</td>
</tr>
<tr>
<td>GL85</td>
<td>8-bit CISC</td>
<td>Intel 8085 ISA, 80 instructions</td>
<td>No pipeline</td>
<td>8,000</td>
<td>VHDL</td>
</tr>
<tr>
<td>i8051</td>
<td>8-bit CISC</td>
<td>Intel 8051 ISA, 44 instructions</td>
<td>No pipeline</td>
<td>12,000</td>
<td>VHDL</td>
</tr>
<tr>
<td>Plasma</td>
<td>32-bit RISC</td>
<td>MIPS-I ISA, 56 instructions</td>
<td>3-stage pipeline Serial port, interrupt controller, timer</td>
<td>26,000</td>
<td>VHDL</td>
</tr>
<tr>
<td>miniMIPS</td>
<td>32-bit RISC</td>
<td>MIPS-I ISA, 51 instructions</td>
<td>5-stage pipeline Hazard detection and forwarding mechanism System coprocessor</td>
<td>33,000</td>
<td>VHDL</td>
</tr>
<tr>
<td>Leon2</td>
<td>32-bit RISC</td>
<td>Sparc V8</td>
<td>5-stage pipeline Debug unit, interrupt controller, timer AMBA bus</td>
<td>35,000</td>
<td>VHDL</td>
</tr>
<tr>
<td>OpenRISC 1200</td>
<td>32-bit RISC</td>
<td>OpenRISC 1000 ISA, 52 instructions</td>
<td>5-stage pipeline Basic DSP capabilities Virtual-memory support</td>
<td>36,000</td>
<td>Verilog</td>
</tr>
<tr>
<td>OpenSparc T1</td>
<td>64-bit RISC</td>
<td>Sparc V9, 196 instructions</td>
<td>8 cores, 4 threads per core Crossbar interconnection Shared floating-point unit Stream-processing unit</td>
<td>273,000 (per core)</td>
<td>Verilog</td>
</tr>
</tbody>
</table>

Functional vs. structural

At the root level of the tree-based taxonomy (see Figure 3), we separate methods that exploit only functional information (e.g., those about the instruction set architecture [ISA]) for the generation process from those that also use structural information (e.g., gate-level or RTL descriptions). This classification is based on the type of processor description used, not the targeted fault model. Hence, methods in the first category, for example, do not necessarily exploit functional-fault models during the generation process. Clearly, functional approaches are particularly interesting because they can be applied even when structural information is not available. On the other hand, the lack of access to this information prevents these approaches from providing any kind of structural-fault coverage results.

Functional approaches

We can divide functional-based strategies into two subclasses. The first corresponds to methods that rely mainly on code randomizers (possibly oriented with suitable constraints) to obtain test programs. The second consists of methods that adopt a feedback-based strategy, meaning they evaluate generated test programs according to suitable metrics (often computed through simulation) and try to progressively improve them.

Shen and Abraham have used a tool named Vertis, which can generate both test and verification programs on the basis of only the processor’s ISA. Vertis generates many different instruction sequences for every instruction tested. This leads to very large test programs. The test program generated for the GL85 microprocessor following this approach is compared to the patterns generated automatically by two ATPG tools. The GL85 program achieves 90.20% stuck-at fault coverage (far higher than that of the ATPG tools)—proving for the first time the efficacy of SBST. In the Vertis environment, either the tool generates pseudorandom instruction sequences with random data, or the user specifies the type of test instruction sequences and determines the heuristics for assigning values to instruction operands to achieve good results. In more complex processors, this is obviously a nontrivial task.

Parvathala, Maneparambil, and Lindsay have presented an automated functional self-test method called Frits (Functional Random Instruction Testing at Speed), which generates random instruction sequences with pseudorandom data. In this paper, the basic requirements for the application of a cache-resident approach have been determined. According to these requirements, the processor must incorporate a cache-load mechanism for the test-program download, and the loaded test program must produce neither cache misses nor bus cycles, so that the address, data, and control buses don’t have to be controlled or observed during test execution. The authors have reported some results on Intel’s Pentium 4.

Figure 3. Taxonomy of routine development styles for SBST.
and Itanium processors for test programs automatically generated by the Frits tool. With the Pentium 4, the test programs achieve 70% stuck-at fault coverage for the entire chip, and enhancing these tests via manually generated tests increases the fault coverage by 5%. With the Itanium processor, the test programs target only the integer and floating-point execution units, and Frits achieves 85% fault coverage.

Bayraktaroglu, Hunt, and Watkins have investigated the potential of using SBST for the manufacturing flow of future Sun processor generations. In this case, the test program comes from a random program generator, and the paper describes the workflow to apply software-based techniques on complex processor cores that exploit on-chip caches. Interesting results are obtained with an UltraSparc T1 processor core. The approaches of Parvathala, Maneparambil, and Lindsay (Intel) and Bayraktaroglu, Hunt, and Watkins (Sun) provide strong evidence of the usefulness of SBST in the manufacturing flow of industrial processor designs.

Unlike the previously described functional methods, the method proposed by Corno et al. uses information feedback to improve test program quality. This approach is based on an evolutionary algorithm and can evolve small test programs and capture target corner cases for design validation. An evolutionary algorithm is a population-based optimizer that attempts to mimic Darwin's theory of evolution to iteratively refine the population of individuals (or test programs). The approach's effectiveness is demonstrated by comparing it with a pure instruction randomizer—with both systems working on an RTL description of the Leon2 processor. The proposed approach can seize three intricate corner cases that the purely random method cannot while saturating the addressed code coverage metrics. Additionally, the developed validation programs are not only more effective, but are smaller as well.

Structural approaches

We subdivide structural methods into two major subcategories. Methods in the first group are called hierarchical because they adopt a hierarchical approach. Such methods focus on a processor's modules one at a time, generating stimuli for each module and then extending those stimuli to the processor level. The second group, called RTL, includes methods in which the test program generation process exploits structural RTL information along with ISA information to generate instruction sequence templates for justifying and propagating faults of the module under test. These templates are then adjusted according to the module's testability requirements.

The test generation process of hierarchical methods includes several steps. The process begins by partitioning the entire processor description into modules and focusing on each module individually. Then, for every considered module, low-level (e.g., gate-level) information is exploited to generate a set of test vectors based on different test generation methods (e.g., ATPG). In most cases, the essence of the process lies in efficiently extracting the behavior of the circuitry surrounding the module under consideration to reduce the effort required to generate test vectors that comply with constraints stemming from the rest of the processor. Finally, the most promising vectors are transformed into instruction sequences that can apply suitable stimuli to each module and propagate fault effects to visible locations (registers or memory).

This group of methods can further be categorized according to the method used to generate the stimuli for the targeted processor module. In particular, one subgroup of hierarchical generation methods is based on precomputed test sets: such methods first generate a set of stimuli for each processor module under consideration, and then exploit some functional processor mechanism (e.g., on the basis of processor instructions) to give the module the precomputed stimuli and to propagate the results to observable locations. Methods by Gurumurthy, Vasudevan, and Abraham and Lingappan and Jha belong to this group.

Gurumurthy, Vasudevan, and Abraham have described a technique for instruction sequence generation targeting hard-to-detect faults that test programs with random instruction sequences (such as those generated by the randomization-based approaches discussed earlier) fail to detect. For every processor module, such faults are considered, and an ATPG tool is used to generate precomputed test sequences for each one. This method uses a bounded model checker technique to identify the processor instructions that can apply the precomputed stimuli to the module inputs. To guarantee fault observability, propagation requirements are expressed as a Boolean-difference problem. A bounded model checker is then used again to extract an instruction sequence that can propagate the effects of the considered fault to a primary output.
The entire process produces the instruction sequences that guarantee the controllability and observability of each considered fault (except when the bounded model checker times out and does not produce results). On an OpenRISC 1200 processor core, the proposed methodology increases overall fault coverage from 64% (saturated coverage value of a test program with random instruction sequences and random data) to 82% for stuck-at faults.

Lingappan and Jha have proposed a generation method based on a satisfiability-based ATPG to develop a test framework that targets stuck-at faults. In this method, an assignment decision diagram (ADD) is first extracted from the microarchitectural description of the processor. This ADD representation facilitates justifying and propagating the test stimuli that can be generated for each module: potential justification or propagation paths from the I/O ports of the embedded module to the primary I/O ports of the processor are derived as a set of Boolean implications. Next, the modules surrounding the module under test are abstracted by their equivalent I/O transparency rules. Then, a satisfiability solver produces valid test sequences that guarantee the detection of the faults targeted by the precomputed test vectors. Finally, the obtained results are further improved by introducing some DFT enhancements derived from the ADD. The very simple Parwan processor core serves as a benchmark for this methodology, achieving about 96% fault coverage, with a DFT area overhead of only 3.1% compared to the original processor size.

A second subgroup of hierarchical generation methods is based on constrained test generation. Such methods feed an ATPG tool with a processor core described using different abstraction levels: the module under consideration is described at the structural level (e.g., gate level), whereas the rest of the processor is described at a higher level, still enforcing existing constraints between the considered module and the rest of the processor. Thus, the ATPG tool theoretically handles a circuit of far lower complexity than the original one. For example, using different techniques, the methods presented by Chen et al. and Wen, Wang, and Cheng extract information about the circuit surrounding the module under consideration.

In the approach proposed by Chen et al., the processor is first partitioned into a set of modules. Then, test program templates are derived for every module. To select the most suitable templates for each module, their observability and controllability properties are analyzed. Next, constrained test generation is performed, which exploits the selected templates. To perform this module-level test generation, the ATPG tool focuses only on the actual module under consideration. The surrounding logic is mapped as a few virtual constraint circuits that model the upstream and downstream circuitry, thus capturing the behavior of the rest of the processor core. Finally, module-level test patterns are translated into instruction test sequences. Results gathered on the execution unit of the Xtensa processor core—a five-stage, pipelined, configurable RISC processor—show 95.2% fault coverage for the functionally testable stuck-at fault list.

Wen, Wang, and Cheng have proposed a module-oriented approach for generating test programs that follows a simulation-based methodology involving two steps: simulation and test program generation. The first step (simulation) applies a set of random patterns to the processor and records the considered module’s I/O behavior, to derive learned models that replace the circuitry before and after the module under evaluation. Different learning mechanisms are proposed, depending on the characteristics of the considered module’s surrounding blocks. In the second step (test program generation), an ATPG tool exploits the learned models to justify and propagate test pattern generation for the faults within the module under test. Experimental results show that the method can obtain about 93% stuck-at fault coverage on the controller and ALU modules of an OpenRISC 1200 processor.

As we discussed earlier, some SBST methods based on structural information mainly exploit RTL information during the generation process. This information can be complemented by other information concerning the processor behavior (e.g., involving the ISA) to generate instruction sequences that control and observe the processor’s modules. The test instruction sequences are adjusted, and the instruction operands are selected according to the target module’s required test pattern set.

RTL methods can be further divided into three subcategories, which differ according to the approach taken to generate the test patterns for each module in the processor:

- Deterministic algorithms benefit from the knowledge of the specific function the module performs and from its regularity. The approaches presented
by Kranitis et al. and Gizopoulos et al. fall into this category.\textsuperscript{12,13} 

- **ATPG algorithms** generate test stimuli starting from a gate-level description of the module. The methods proposed by Chen et al. and Krstic et al. belong to this category.\textsuperscript{15,16} 

- **Pseudorandom methods** randomly generate sequences of instructions and/or data to be internally elaborated by the processor. The methods proposed by Rizk, Papachristou, and Wolff and Krstic et al. fall into this category.\textsuperscript{14,16} 

Kranitis et al. have proposed a component-based divide-and-conquer approach.\textsuperscript{12} Program generation is based only on the processor ISA and its RTL description. The methodology is organized into three phases: identification of processor components and component operations, classification of processor components and prioritization for test development, and development of self-test routines for processor components. For the last phase, deterministic routines are developed according to guidelines introduced to tackle specific components. The authors have successfully applied this approach to a few pipelined processor cores and provide results concerning stuck-at faults: a Plasma (MIPS-like) processor core obtains an overall fault coverage of about 95% for three different processor synthesis processes, and a five-stage MIPS R3000 compatible processor core reaches about 93% of the overall fault coverage. The authors have also compared results obtained with an approach using deterministic data with those from ATPG-based and pseudorandom data. 

Gizopoulos et al. have enhanced the previous methodology to test the pipelined logic of more sophisticated pipelined processors.\textsuperscript{13} They first identify the testability hotspots of the pipelining logic, applying existing SBST programs (generated according to the methodology by Kranitis et al.\textsuperscript{12} and targeting the processor's functional components) to two fully pipelined RISC processor models: miniMIPS and OpenRISC 1200. The proposed automated methodology complements any other SBST generation approach that targets functional components. It analyzes the data dependencies of an existing SBST program and considers the basic parameters of the pipelined architecture (number of stages, forwarding paths, etc.) and the memory hierarchy system (virtual- and physical-memory regions) to generate an enhanced SBST program that comprehensively tests the pipelined logic. The experimental results on the two processors show that the enhanced SBST program achieves significant fault coverage improvements for the pipelined logic (19% improvement on average) and for the total processors (12% improvement on average).

Rizk, Papachristou, and Wolff have applied a mixed methodology using SBST and low-overhead BIST to test embedded digital-signal processing (DSP) core components.\textsuperscript{14} Test routines are generated on the basis of each instruction's ability to control and observe each DSP component. These test routines are executed within a loop, with different random numbers delivered by appropriate hardware-inserted random generators, to enhance the final fault coverage. As the results show, the method is suitable for functional, but not for control, components. Results on an industry-based pipelined DSP achieve about 98% stuck-at fault coverage.

Chen et al. have described a two-step test-program generation methodology that uses multiple-level abstractions of the processor under consideration, including the ISA description, RTL model, architectural model, and synthesized gate-level netlist.\textsuperscript{15} In the first phase, processor classification identifies ISA registers, logic and control blocks, pipelined registers, and pipeline-related control logic. In the second phase, for every identified component, suitable test routines are developed on the basis of different abstraction levels. Register and processor control parts are mainly tackled by exploiting the RTL and architectural processor descriptions, whereas fundamental-logic chunks (e.g., the ALU) are addressed via the processor gate-level synthesized version and a constrained ATPG tool. Experimental results are gathered on a compatible ARM4 processor core developed by the authors. The method attains 93.74% fault coverage for processor stuck-at faults.

A method proposed by Krstic et al. targets both stuck-at and path-delay faults.\textsuperscript{16} For stuck-at faults, they use a divide-and-conquer approach, deriving a set of spatial and temporal constraints from the microprocessor ISA for every microprocessor component. Spatial constraints are produced in the form of Boolean equations, which can be derived by observing the processor instructions and can correctly activate the component under consideration. Temporal constraints are generated by analyzing the sequences of instructions required to apply test patterns to the processor component, observing the results after their
elaboration. Once the constraint set is built, an ATPG tool or random generator fed with this set handles test pattern generation. Thus, from this viewpoint, the method belongs to both the second and third subcategories (i.e., ATPG algorithms and pseudorandom test generation) of the RTL methods discussed thus far. Finally, for the considered component, the set of test programs is derived from the generated patterns. Experimental results on the simple Parwan processor achieve a fault coverage of 92.42%.

For path-delay faults, the test program generation process is similar: First, a set of spatial and temporal constraints is produced. Then, a path classification algorithm is run that exploits the constraint set to eliminate functionally untestable paths from the fault universe. Afterward, an ATPG tool handles test vector generation for every path labeled as functionally testable. Finally, a set of test programs is synthesized following the test vectors generated in the previous step. Experimental results on the Parwan and DLX processors show that, for testable path-delay faults, the proposed approach achieves 98.8% and 96.3% fault coverage respectively.

Comparison

The approaches we’ve discussed follow different methods for developing effective test programs. Table 2 presents the advantages and limitations of the four key categories of our classification: functional approaches, hierarchical structural approaches using precomputed stimuli, hierarchical structural approaches based on constrained test generation, and RTL structural approaches. Summarizing the previous approaches in Table 2, we also highlight the achievements in this research field and the open research issues.

Functional approaches can be easily integrated in any processor design flow because they are based only on the ISA and don’t require sophisticated test development or experienced test engineers. Their basic limitation is that they cannot achieve high structural-fault coverage because they don’t consider the processor structure. Also, they might require manual intervention in some steps of the test generation process (i.e., to specify test instruction sequences and operands, bias the test-program generation tool, or convert existing legacy test programs to cache-resident tests); hence, automation in most of these approaches is limited. An exception is the method proposed by Corno et al., in which manual intervention is far less because the test generation process is guided by an evolutionary tool and high-level coverage metrics.

Hierarchical structural approaches based on precomputed test stimuli can be highly automated. Also, they can be combined with one of the functional approaches to target hard-to-test faults that escape functional approaches, to achieve higher fault coverage. Their main advantage is that they exploit mature formal-verification engines to automatically resolve computationally intensive test-program generation problems. However, formal-verification methods (such as bounded model-checking and satisfiability-based methods) could be computationally prohibitive for complex processor architectures (i.e., with deep pipelines, dynamic instruction scheduling, advanced speculative mechanisms, or multithreading technology). Thus, the effectiveness of formal-verification methods for test generation

<table>
<thead>
<tr>
<th>Approaches</th>
<th>Advantages</th>
<th>Limitations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional</td>
<td>Readily applicable to any design flow</td>
<td>Hard to achieve high fault coverage because of lack of structural information</td>
</tr>
<tr>
<td>Structural, hierarchical, precomputed stimuli</td>
<td>Target hard-to-test faults that escape functional approaches</td>
<td>Formal-verification methods are computationally prohibitive for complex circuits</td>
</tr>
<tr>
<td>Structural, hierarchical, constrained test generation</td>
<td>Reduce test generation complexity</td>
<td>Test efficiency depends on manually generated instruction templates and simulation-based learned models</td>
</tr>
<tr>
<td>Structural, RTL</td>
<td>Applicable to complex architectures</td>
<td>Require knowledge of the RTL structure to generate test programs</td>
</tr>
<tr>
<td></td>
<td>Efficient testing of large functional modules</td>
<td>Less effective on control logic</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Limited automation</td>
</tr>
</tbody>
</table>
must still be demonstrated on more advanced architectures.

Hierarchical structural approaches based on constrained test generation avoid the exponential growth of test generation complexity by using simulation-based learning to describe the circuit at higher abstraction levels. However, their test efficiency depends on the quality of the instruction templates and learned models, which cannot be guaranteed. In fact, instruction templates must be written manually, and learned models are based on statistical results obtained through time-consuming simulations. Thus, these approaches should be evaluated in more-complex architectures to prove the effectiveness of learned models and constrained test generation.

Finally, RTL-based structural approaches can handle complex architectures because they can isolate and generate efficient test programs for large functional units regardless of the processor's architectural complexity. However, they are characterized by a low automation level because they require significant human effort to acquire in-depth knowledge of processor architecture and to analyze in detail the data path and control modules. An exception is the approach by Krstic et al., which combines an RTL method with constrained test generation to support automation. Although RTL approaches have been proved efficient for testing functional units, they must be evaluated for testing the control logic of advanced-performance mechanisms (e.g., speculative mechanisms and multithreading technology).

Our comparison concludes with quantitative information. Table 3 presents, in chronological order, the fault simulation results of the approaches we’ve discussed thus far. The information presented is a rough indication of these approaches’ feasibility and effectiveness in the adopted processor models. Some of the approaches have already been applied in industrial test flows—for example, in the manufacturing-test process of Intel’s Pentium 4 and Itanium processors and Sun’s UltraSparc T1 processor. For others, the real applicability must still be evaluated in more-complex processors in terms of development cost (human effort and computational resources) and test efficiency (coverage for different fault models). But, SBST is certainly a mature technique today for testing simple microprocessors and microcontrollers for the most traditional fault models (e.g., stuck-at faults). Some techniques have proved effective with more-sophisticated pipelined architectures as well. However, the application of SBST to advanced architectures with dynamic execution and multiprocessing and multithreading technology is still mainly a research topic (which we will discuss later).

<table>
<thead>
<tr>
<th>Approach</th>
<th>Year</th>
<th>Microprocessor model</th>
<th>Fault coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shen and Abraham</td>
<td>1998</td>
<td>GL85</td>
<td>90%</td>
</tr>
<tr>
<td>Krstic et al.</td>
<td>2002</td>
<td>Parwan</td>
<td>92%</td>
</tr>
<tr>
<td>Parvathaia, Maneparambil, and Lindsay</td>
<td>2002</td>
<td>Pentium 4</td>
<td>75%</td>
</tr>
<tr>
<td>and Lindsay</td>
<td></td>
<td>Itanium (integer and FP units only)</td>
<td>85%</td>
</tr>
<tr>
<td>Chen et al.</td>
<td>2003</td>
<td>Xtensa (ALU only)</td>
<td>95%</td>
</tr>
<tr>
<td>Corno et al.</td>
<td>2004</td>
<td>Leon2</td>
<td></td>
</tr>
<tr>
<td>Rizk, Papachristou, and Wolf</td>
<td>2004</td>
<td>DSP core (functional units only)</td>
<td>98%</td>
</tr>
<tr>
<td>Kranitis et al.</td>
<td>2005</td>
<td>Plasma, MIPS R3000</td>
<td>95%</td>
</tr>
<tr>
<td>Bayraktaroglu, Hunt, and Watkins</td>
<td>2006</td>
<td>UltraSparc T1</td>
<td>Not reported</td>
</tr>
<tr>
<td>Gurumurthy, Vasudevan, and Abraham</td>
<td>2006</td>
<td>OpenRISC 1200</td>
<td>82%</td>
</tr>
<tr>
<td>Wen, Wang, and Cheng</td>
<td>2006</td>
<td>OpenRISC 1200 (controller and ALU only)</td>
<td>93%</td>
</tr>
<tr>
<td>Lingappan and Jha</td>
<td>2007</td>
<td>Parwan</td>
<td>96%</td>
</tr>
<tr>
<td>Chen et al.</td>
<td>2007</td>
<td>ARM4 core</td>
<td>94%</td>
</tr>
<tr>
<td>Gizopoulos et al.</td>
<td>2008</td>
<td>MiNiMIPS, OpenRISC</td>
<td>93%</td>
</tr>
</tbody>
</table>

* FP: floating point.
** For details about the microprocessor models, see the sidebar, “Typical Open-Source Microprocessor Benchmark Models.”

Table 3. Reported fault simulation results (stuck-at faults).
Other key SBST issues

Several research teams have addressed other important issues of the SBST concept, such as enhancement of SBST testability using DFT techniques, extension of SBST to other fault models, power-constrained SBST, and using SBST to test other components in a typical SoC.

Design for software-based testability

To improve SBST efficiency on a processor core, some approaches propose using instruction-level DFT, either to facilitate the automatic generation of test instruction sequences or to increase the controllability and observability of the processor’s internal nodes during self-test program execution.\(^\text{17-19}\)

Batcher and Papachristou have proposed adding test hardware for better randomization of self-test programs to achieve higher fault coverage.\(^\text{17}\) The extra hardware either modifies the stored test software to create pseudorandom instruction sequences or feeds the test software with pseudorandom seeds to randomize the instruction operands. The instruction-level DFT methodology proposed by Lai and Cheng adds new test-specific instructions to improve processor testability and to reduce test program size and execution time.\(^\text{18}\) The DFT method proposed by Nakazato et al. adds observation points to the processor to reduce error masking.\(^\text{19}\)

SBST for delay faults

With technology improvements and frequency increases, testing for delay faults is becoming increasingly important, and some proposals extend SBST techniques to address path-delay faults. In addition to the method proposed by Krstic et al.,\(^\text{16}\) the approaches of Christou et al., Singh et al., and Gurumurthy et al. belong to this group.\(^\text{20-22}\)

Christou et al. have presented a test generation strategy that exploits different descriptions of the processor under consideration.\(^\text{20}\) The proposed strategy uses the gate-level description to select a set of path-delay faults; for this set, all necessary path excitation requirements are extracted via binary decision diagram (BDD) analysis. This strategy also uses the RTL description effectively to identify test programs that can reproduce the conditions necessary to activate a given set of target path-delay faults (excitation) and to make the fault effects visible at the processor outputs (propagation). For the generation process, an evolutionary algorithm is employed that relies on RTL simulation. Results on an 8051 microcontroller reach about 91% fault coverage on functionally excitable path-delay faults.

Singh et al. have proposed a methodology for path-delay fault testing of pipelined processors based on a pipelined-instruction execution graph, which is derived from the ISA and the RTL description of the processor. The graph is used for architectural constraint extraction and path classification. Test vectors are generated executing constrained ATPG under the extracted constraints.\(^\text{21}\) Experimental results on vPro and DLX processors show that this approach achieves 100% fault efficiency for functionally testable paths.

Gurumurthy et al. have described a test generation method for path-delay faults that uses a delay-test-annotated ATPG tool based on the Podem (Path-Oriented Decision-Making) algorithm to generate delay tests locally, a satisfiability-based verification engine to map the tests to processor instructions, and a feedback mechanism to accelerate the process.\(^\text{22}\) Experimental results on modules of an OpenRISC 1200 show that the proposed approach achieves 96% path-delay fault coverage for functionally testable paths.

SBST for performance faults in speculative units

To increase instruction-level parallelism, computer architects incorporate speculative mechanisms, such as branch prediction and data value prediction. A potential fault in a speculative unit might not be manifested, because its correcting mechanism masks the erroneous speculation. Such faults don’t affect program behavior but could impose performance degradation; thus they are called performance faults. Hatzimihail et al. have shown that a significant portion of stuck-at faults in a branch prediction unit escape in a conventional SBST approach, and they’ve proposed enhancing the approach using performance-monitoring hardware.\(^\text{23}\) The enhanced SBST approach accesses performance counters to read speculation-related data carrying the fault effects before the erroneous speculation is masked. Experimental results on two branch prediction schemes of a miniMIPS core show that the enhanced SBST program increases stuck-at fault coverage of the branch prediction unit from 80% to 97%.

SBST with power constraints

Compared to DFT-based testing strategies, SBST techniques are less prone to power-related problems.
(e.g., due to overheating during test application) because testing exploits normal processor functionality. However, some testing approaches require high switching activity and thus increased power consumption, to stress the circuit's critical paths and screen-timing defects. In such cases, test stimuli that maximize power must be found—for example, to check the processor's correct behavior in these extreme conditions. In other cases, such as online testing, lowering power consumption during SBST application time is a strong requirement.

Zhou and Wunderlich have proposed a method to achieve high fault coverage while also addressing power consumption. Experimental results on a 32-bit RISC processor show a reduction in energy (about 40%) and average power consumption (about 14%) over the original test program without affecting fault coverage.

Merentitis et al. have proposed a low-energy-oriented SBST approach suitable for online testing. This approach uses compiler-like techniques such as loop unrolling, instruction substitution, and register renaming. Experimental results on two RISC processor cores demonstrate that as much as 35% power can be saved by applying the proposed approach, without harming fault coverage.

SBST for peripheral cores in SoCs

In a typical SoC, SBST can also be exploited to test nonprogrammable SoC cores, such as peripheral-communication controllers.

Apostolakis et al. have proposed a hybrid SBST method for testing communication peripherals in a typical SoC architecture. One technique builds a fully automated test-program generation flow around an evolutionary algorithm, which iteratively generates test programs for peripherals embedded in a SoC, to maximize high-level coverage metrics. The second technique introduces a different test generation flow, which deterministically precomputes stimuli and maps them into processor instructions that can directly excite peripheral cores. The authors propose a combined approach that exploits the deterministically generated test routines from the second technique to guide the automatic test-program generation of the first technique. The effectiveness of this methodology is experimentally demonstrated on an OpenRISC-1200-based SoC containing three popular communication interfaces—UART (Universal Asynchronous Receiver-Transmitter), HDLC (High-Level Data Link Control), and Ethernet—reaching more than 90% fault coverage in all cases.

SBST during the chip life cycle

Research teams have also studied the potential of SBST in different phases of the microprocessor chip-testing process.

Software-based diagnosis

Chen and Dey and Bernardi et al. have studied software-based diagnosis (albeit lacking physical-design details) to see if it can provide valuable information regarding candidate failure locations during fault diagnosis. The main objective of each approach is to generate a final diagnostic set with high fault coverage, comprising many small test programs with high diagnostic resolution. Chen and Dey have performed manual development of diagnostic programs. Pass/fail simulations are performed to build a tree-based fault dictionary for the evaluation of the diagnostic resolution of the test programs. Bernardi et al. have proposed automated manipulation of existing test programs, which are split into a very large set of small independent fragments, named spores.

Bernardi et al. have demonstrated the effectiveness of the approach with experiments on an i8051 microcontroller. The final diagnostic programs achieve very high diagnostic resolution (the fraction of all covered fault pairs that are distinguishable), more than 99.85% for two different test programs. When comparing their approach with that of Chen and Dey, Bernard et al. have found that their approach reaches a higher level of diagnostic capability for the divider of the i8051 core—uniquely classifying more faults.

Online periodic SBST

Concurrent online test approaches are commonly based on redundancy techniques. For noncritical, low-cost embedded applications in which redundancy overhead is cost prohibitive and detection latency requirements are relaxed, an online periodic (nonconcurrent) test strategy provides an adequate reliability level.

Paschalis and Gizopoulos have studied SBST feasibility for online periodic testing and investigated the best trade-off between fault detection latency and system performance degradation. They propose executing an online periodic SBST strategy in which the self-test programs are grouped in a separate system process. Such a strategy can be performed in
one of two ways: the operating system can identify idle periods and execute the test program, or the test program can be executed at regular time intervals with the aid of programmable timers. In the former, fault detection latency depends on test-program execution time; in the latter, it depends on both the test period and test-program execution time. Therefore, the main characteristic of an SBST program suitable for online periodic testing is the shortest possible test execution time, which must be less than a time quantum. Paschalidis and Gizopoulos have proposed a test-program generation methodology which produces test programs that meet these requirements while also achieving high fault coverage. They’ve demonstrated the effectiveness of their methodology on two RISC processor models.

Online periodic SBST execution is even more challenging when applied during normal operation of systems with hard real-time constraints. Gizopoulos has presented a self-test program scheduling approach that maximizes effective self-test utilization without affecting the system’s real-time requirements.

Recent trends in SBST research

The most recent architectural advances dictate the integration of multiple cores and hardware threads on a single die to deliver high computing power by exploiting thread-level execution parallelism. Following these advances, a recent trend in SBST research is to scale SBST techniques in multiprocessor and multithreading architectures. For example, in a recent International Test Conference (ITC) 2008 panel (Panel 6, “The University DFT Tool Showdown,” organized by Sun Microsystems), our research teams (University of Piraeus and Politecnico di Torino) presented research activities on the application of the SBST paradigm to real, industrial multithreading processors.

The three main challenges regarding scaling the SBST paradigm to multiprocessor and multithreaded architectures are

- use of proven SBST techniques for uniprocessor architectures to individually test all processor cores and all thread-level execution pipelines,
- exploitation of core- and thread-level execution parallelism to speed up self-test program runtime, and
- efficient testing of interoperability logic at the thread and core levels (i.e., control logic for thread switching, intercore communication, and cache coherency).

Two recent approaches focus on the second challenge and thus aim to reduce the test application time by exploiting core- or thread-level parallelism. Apostolakis et al. have proposed porting an SBST approach from the unicom case to a bus-based symmetric multiprocessor (SMP) architecture. Their self-test scheduling algorithm at the core level significantly minimizes the time overheads caused by data cache coherency invalidations and intense bus contention among cores. They’ve demonstrated their methodology in two-, four-, and eight-core versions of a multiprocessor based on an OpenRISC 1200 core. Experimental results show that the methodology achieves more than 91% total stuck-at fault coverage for all multicore, while reducing test application time by more than 24% compared to the fastest alternative.

In another study, Apostolakis et al. have applied SBST to the chip-multithreading architecture of Sun’s OpenSparc T1, which integrates eight CPU cores, each supporting four hardware threads. Their focus is on reducing test execution time in every physical core of the chip by exploiting thread-level parallelism. Their methodology exploits thread-waiting intervals due to cache misses and long-latency operations. Experimental results show that this methodology reduces test execution time for each core by a factor of 3.3 compared to the test execution time of the test code’s single-threaded version.

Guzey, Wang, and Bhadra have proposed a simulation trace-based methodology that reduces the development complexity of functional testbenches for both functional verification and postsilicon performance validation. Their automatic constraint extraction tool, which use data-mining algorithms to analyze simulation data, can be easily integrated with an existing commercial test generation framework. They’ve demonstrated the scalability of this methodology in three units of an OpenSparc T1.

The SBST approaches and results discussed in this article show that SBST has already reached a significant level of maturity. Recent research efforts are focused on developing effective techniques and methodologies to apply the SBST approach to more complex and emerging architectures. These include microprocessors with dynamic instruction execution, multithreading and multicore technology, and GPUs. Research efforts are also focusing on applying SBST
to problems such as diagnosis, silicon debug and validation, online testing, stress testing, and speed binning. Many of these SBST research efforts are expected to be intensified in the near future.

References

Microprocessor Test and Validation


**Mihalis Psarakis** is a lecturer in the Department of Informatics at the University of Piraeus in Greece. His research interests include microprocessor and SoC testing, and fault-tolerant design of embedded systems. He has a PhD in computer science from the University of Athens. He is a member of the IEEE Computer Society and the IEEE.

**Dimitris Gizopoulos** is an associate professor in the Department of Informatics at the University of Piraeus. He is also an associate editor of *IEEE Design & Test, IEEE Transactions on Computers*, and *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*. His research interests include dependable computer architectures, and test and fault tolerance of microprocessors and multiprocessors. He has a PhD in computer science from the University of Athens. He is a Golden Core Member of the IEEE Computer Society, a senior member of the IEEE, and a member of the ACM.

**Ernesto Sanchez** is an assistant professor in the Department of Control and Computer Engineering at Politecnico di Torino. His research focuses on microprocessor testing. He has a PhD in computer engineering from Politecnico di Torino. He is a member of the IEEE.

**Matteo Sonza Reorda** is a full professor in the Department of Control and Computer Engineering at Politecnico di Torino. His research interests include testing and fault-tolerant design of electronic circuits and systems. He has a PhD in computer engineering from Politecnico di Torino. He is a senior member of the IEEE.

Direct questions and comments about this article to Dimitris Gizopoulos, University of Piraeus, Department of Informatics, 80 Koraoli & Dimitriou St., 18534, Piraeus, Greece; dgizop@unipi.gr.
RAISE YOUR STANDARDS
Software Development

Get Certified

“The CSDA establishes a core set of competencies for entry-level software development practitioners.”
Tori Wenger
Sr. Manager
Rockwell Collins

The CSDA certification is intended for entry-level software development practitioners and is based upon the 15 Knowledge Areas (KAs) of the internationally-recognized SoftWare Engineering Body Of Knowledge (SWEBoK) Guide.

Key benefits of CSDA Certification:

1. **Practitioners**: Demonstrate your knowledge of established software development practices, and become productive more quickly.
2. **Employers**: Shorten the training cycle for new practitioners and establish a baseline for software development practices for your organization.
3. **Industry Recognition**: The CSDA was developed by the IEEE Computer Society, an international leader in the software engineering profession, in conjunction with key academic and industry leaders.

To learn more about our programs and how they can help your organization, visit us at www.computer.org/getcertified

May/June 2010