Effective Software-Based Self-Test Strategies for On-Line Periodic Testing of Embedded Processors

Antonis Paschalis, *Associate Member, IEEE,* and Dimitris Gizopoulos, *Senior Member, IEEE*

***Abstract*—Software-based self-test (SBST) strategies are particularly useful for periodic testing of deeply embedded processors in low-cost embedded systems with respect to permanent and intermittent operational faults. Such strategies are well suited to embedded systems that do not require immediate detection of errors and cannot afford the well-known hardware, information, software, or time-redundancy mechanisms. In this paper, we first identify the stringent characteristics of a SBST program to be suitable for on-line periodic testing. Also, we study the probability for a SBST program to detect permanent and intermittent faults during on-line periodic testing. Then, we introduce a new SBST methodology with a new classification and test-priority scheme for processor components. After that, we analyze the self-test routine code styles for the three more effective test pattern generation (TPG) strategies in order to select the most effective self-test routine for on-line periodic testing of a component under test. Finally, we demonstrate the effectiveness of the proposed SBST methodology for on-line periodic testing by presenting experimental results for two pipeline reduced instruction set computers reduced instruction set processors of different architecture.**

***Index Terms*—Intermittent faults, on-line testing, periodic testing, processor testing, software-based self-test (SBST).**

I. Introduction

AFTER manufacturing testing, an embedded system is placed in its natural environment, where *operational faults* may appear. Operational faults in deep submicron technology are classified into the following three categories. *Permanent operational faults* are infinitely active at the same location and reflect irreversible physical changes. *Intermittent operational faults* appear repeatedly at the same location and cause errors in bursts only when they are active. These faults are induced by unstable or marginal hardware due to process variations and manufacturing residuals and are activated by environmental changes. In many cases, intermittent faults precede the occurrence of permanent faults. *Transient operational faults* appear irregularly at various locations and last short time. These faults are induced by neutron and alpha particles, power supply and interconnect noise, electromagnetic interference and electrostatic discharge.

On-line testing aims at detecting and/or correcting these operational faults by means of *concurrent* and *nonconcurrent* test strategies [1]. Concurrent on-line test strategies are used to detect all kinds of operational faults, while keeping the system in normal operation and are classified into the following four categories. *Hardware redundancy* strategies like duplication and comparison for fault detection and triple modular redundancy for error correction. *Information redundancy* strategies based on various coding schemes and self-checking design. Both hardware and information-redundancy strategies have low fault-detection latency, but impose large to huge hardware overhead [2]. However, when a large increase in silicon area is not acceptable, the other two categories of concurrent on-line test strategies are used. *Time redundancy* strategies based either on recomputing using shifted/swapped operands or recomputing using duplication and comparison. *Software redundancy* strategies like N-version programming and software signature monitoring [3]. Both time- and software-redundancy strategies have higher fault-detection latency (compared to the first two strategies) and impose large to huge performance overhead.

All concurrent on-line test strategies are high-cost solutions that trade off among hardware overhead, performance overhead and fault-detection latency. It is a common practice of the cost of on-line test strategies to be weighted against the cost of undetected faults.

In nonsafety-critical low-cost applications of embedded systems, there is no need for immediate detection of errors and, thus, no need for hardware, information, software, or time-redundancy mechanisms that increase significantly the system cost. In such embedded systems, detection of intermittent operational faults that cause errors in bursts only when they are active and may precede the occurrence of permanent faults, is much more important than detection of transient operational faults that appear once and last a short time. Therefore, on-line periodic testing is well suited to such embedded systems since it detects at low cost, not only permanent faults, but intermittent faults with very high probability. After fault detection, the system may reapply periodic testing several times to ensure that the fault is permanent or simply the system is restarted. On-line periodic testing is a nonconcurrent test strategy that trades off between fault-detection latency and performance overhead.

The problem of on-line periodic testing of deeply embedded processors in high-complexity low-cost nonsafety-critical embedded systems is becoming challenging for the following reason. *Hardware-based self-test* (HBST) techniques for on-line periodic testing, like built-in self-test (BIST), provide excellent test quality as they achieve at-speed testing with high fault coverage. However, in the case of high performance, low area and low power consumption embedded processors, the application of HBST techniques is limited and sometimes prohibited due to manual and extensive design changes, high hardware overhead, and high power consumption.

Manuscript received March 6, 2004; revised June 15, 2004. This paper was recommended by Guest Editor J. Figueras.

A. Paschalis is with the Department of Informatics and Telecommunications, University of Athens, 15784 Athens, Greece (e-mail: paschali@di.uoa.gr).

D. Gizopoulos is with the Department of Informatics, University of Piraeus, 18534 Piraeus, Greece (e-mail: dgizop@unipi.gr).

Digital Object Identifier 10.1109/TCAD.2004.839486

Recently, the use of low-cost *software-based self-test* (SBST) techniques for on-line periodic testing of embedded processors has been proposed in [4] as an effective alternative solution to HBST techniques. The SBST techniques utilize internal processor components to perform test application and response evaluation and are nonintrusive in nature as they use the processor-instruction set to perform self-testing. The key concept of SBST is the generation of an efficient self-test program that achieves high fault coverage without processor modifications. The processor executes periodically the SBST program residing in the memory system (e.g., in a flash memory) at its actual speed (at-speed testing) and very small area, performance or power consumption overheads are induced for the embedded system. In the case of on-line periodic testing, the SBST program must satisfy the following requirements: small memory footprint, small execution time and low power consumption [4]. In addition, there is a demand for low development cost of the SBST program particular necessary for low-cost applications of embedded systems.

SBST techniques are *functional* in nature that use random instruction sequences, operations and operands, have been proposed in [4]–[7]. Such techniques have low test development cost due to their high abstraction level, but they also achieve medium to high fault coverage with a large number of instruction sequences. Thus, the derived test program is large and requires excessive test-execution time. Also, long fault-simulation time is required for fault grading. Therefore, functional-based SBST techniques are not suitable to on-line periodic testing.

SBST techniques *structural* in nature, targeting processor components have been proposed in [8]–[10] as promising techniques for efficient testing of a processor deeply embedded in an embedded system. First, based on a divide-and-conquer approach, processor components and their corresponding component operations are identified. Then, for every component under test (CUT) within the processor and for every operation of the CUT, test patterns are generated targeting structural faults. After that, the test patterns are transformed to self-test routines (consisting of processor instruction sequences) which are used to apply test patterns to the inputs of the CUT and collect test responses from the outputs of the CUT. All self-test routines together constitute a test program with stringent requirements in code size, data size, and execution time in order to be suitable to on-line periodic testing of the embedded processor.

The test patterns are derived by following the three more effective test pattern generation (TPG) strategies. The first TPG strategy is based on deterministic automatic TPG (ATPG) and is usually applied to combinational components, where instruction-imposed constraint ATPG is feasible. The second TPG strategy is based on pseudorandom TPG and is applied to combinational components with irregular structure, where instruction-imposed constraints can be considered. Both TPG strategies are low gate-level strategies, since they require the knowledge of the gate-level structure of the embedded processor; the former to achieve ATPG and the latter to identify polynomial, seed and number of test patterns. The third TPG strategy is based on regular deterministic TPG [9], [10] that exploits the inherent regularity of the most critical to test processor components like arithmetic and logic components, shifters, comparators, multiplexers, registers and register files which usually constitute the vast majority of processor components. In many cases, acceptable fault coverage is derived after testing only these components. This TPG strategy is a high-level strategy, since the derived test patterns are independent on gate-level implementation and constitute test sets of constant or linear size.

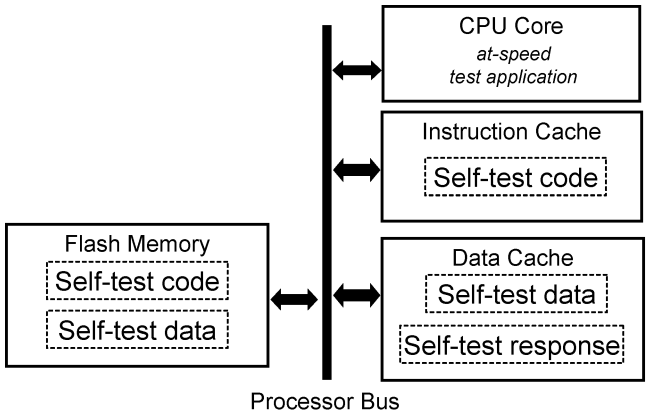

Fig. 1. Concept of SBST.

In this paper, we first identify the stringent characteristics of a SBST program for on-line periodic testing. Also, we study the probability for a SBST program to detect permanent and intermittent faults during on-line periodic testing. Then, we introduce a new SBST methodology for on-line periodic testing that includes a new classification and test priority scheme for processor components. This scheme is well suitable for the systematic selection of the convenient TPG strategy for test pattern derivation, as well as, the systematic transformation of test patterns to a self-test routine. After that, we analyze the self-test routine code styles for the three TPG strategies with respect to code size, data size, and execution-time characteristics. Such an analysis is needed to select the most effective self-test routine for on-line periodic testing according to specific processor component test pattern characteristics. Finally, we demonstrate the effectiveness of our SBST methodology for on-line periodic testing by presenting experimental results for two pipeline reduced instruction set computer (RISC) processors of different architecture.

II. On-Line Periodic Test-Program Characteristics

On-line periodic testing is performed in-field, while the processor operates at its normal operational environment. The processor executes the efficient SBST program residing in the memory system (e.g., in a flash memory) at its actual speed (at-speed testing) under the supervision of the operating system, as shown in Fig. 1.

Test-program execution may be initiated during system startup or shutdown, thus ensuring system normal operation with respect to permanent faults, but it imposes large fault detection latency. Alternatively, the operating system scheduler may identify idle cycles and issue test-program execution or test program may be executed at regular time intervals with the aid of programmable timers found in the system. In these cases,

the SBST program is another process that has to compete with user processes for system resources, CPU cycles and memory.

To alleviate the system's operation overhead, a SBST program should run in the minimum possible number of CPU clock cycles. An ideal period for test-program execution should be the *quantum time cycle*, assuming an operating system with round robin scheduling. Typical values of quantum times used in embedded applications are in the range of a few hundreds of milliseconds. Although it is possible to have test-program execution span over more than one quantum time, this will lead to further system operation overhead, due to larger context switch overheads and must be avoided in practice.

In case the operating-system scheduler identifies idle cycles and issues test-program execution, *fault-detection latency*, with respect to permanent and intermittent faults, depends on the test-program execution time. The test-program execution time must be as short as possible and less than a quantum time cycle in order to reduce fault-detection latency.

In case the test program is executed at regular time intervals with the aid of programmable timers (i.e., periodic testing), fault-detection latency depends on the time interval between two consecutive test-program executions, as well as, the test program execution time. The time interval is specified as a tradeoff between user–program performance and fault-detection latency with the capability of the system to detect both permanent and intermittent faults. In the next session, we study the probability for a SBST program to detect permanent and intermittent faults during on-line periodic testing. Also, the test-program execution time must be as short as possible and less than a quantum time cycle in order to reduce fault-detection latency.

Therefore, the main characteristic of an SBST program to be suitable for on-line periodic testing is *the shortest possible test execution time which must be less than a quantum time cycle*.

The test program execution time can be generally described by the following equation [11]:

$$\begin{aligned}&\text{CPU\_execution\_time}\\&\quad=\text{clock\_cycle\_time}\\&\qquad\times(\text{CPU\_clock\_cycles}+\text{pipeline\_stall\_cycles}\\&\qquad\quad+\text{memory\_stall\_cycles}).\end{aligned}$$

Based on this equation, we conclude that the number of instructions that constitute the SBST program must be as small as possible with careful use of instructions with large clocks per instruction (CPI). Also, the existence of pipeline stalls and memory stalls increase test program execution cycles and must be avoided when possible.

Pipeline stalls should be avoided by constructing test programs which do not cause unresolved data hazards. Control hazards are usually avoided in architectures that implement the *branch delay slot* resolution, like MIPS processors, by proper instruction placement in the delay slot. However, pipeline stalls are unavoidable when *branch prediction* is used to handle branch conditions.

Test programs with big memory footprint (code and data) take more time to run due to increased number of memory stalls. Additionally, such a test program may force user programs to be unloaded from cache memory. When the user program resumes, it will experience cache misses which will affect its performance. Memory stalls are reduced when test programs take advantage of *temporal* and *spatial* locality.

A common application for on-line periodic testing is mobile applications, where power consumption is of great importance. A study by Intel [12] shows that 33% of a notebook system power is consumed in the CPU with a two-level cache hierarchy system. In the CPU, about 20%–30% of power is consumed in the cache system and about 30% is consumed in clock circuitry. Considering the data transfers from external memory in case of a cache miss the power consumed in the overall memory system increases furthermore. The processor has to stall when a cache miss occurs. Extra energy has to be consumed in driving clock-trees and pulling up and down the external bus between the on-chip cache and external memory. Therefore, reduction of memory stalls also reduces power consumption during on-line periodic testing.

Consequently a SBST program for on-line periodic testing must have the following stringent characteristics:

- the shortest possible test execution time which must be less than a quantum time cycle;
- small number of executed instructions with careful use of instructions with large CPI;
- small code without unresolved data hazards and with as much as possible compact loops that take advantage of temporal locality and sequentially executed instructions that take advantage of spatial locality;
- small data structured in arrays that take advantage of spatial locality.

III. Detection of Permanent and Intermittent Faults During On-Line Periodic Testing

When an intermittent fault exists in an embedded processor, it may be active, causing errors in bursts or it may be inactive, allowing the system to operate correctly. In this case, the processor behaves as it repeats nonactive states (state 0) and active states (state 1), alternately. To describe the behavior of intermittent faults, we adopt the *two-state continuous-parameter Markov model* for intermittent faults presented in [15]. According to this model, the time period during which the processor stays at states 0 or 1 is exponentially distributed with mean $1/\lambda$ and $1/\mu$, respectively. For the case of intermittent faults, the transition probability $P_{01}(t)$ for going from state 0 at time t_0 to state 1 at time t_0+t is given by the

$$P_{01}(t)=\frac{\lambda}{(\lambda+\mu)}\cdot\left(1-e^{-(\lambda+\mu)t}\right). \tag{1}$$

For the case of permanent faults, the processor continuously remains at state 1 after going once from states 0 to 1, and thus $\mu=0$. In this case, the transition probability $P_{01}(t)$ for going from state 0 at time t_0 to state 1 at time t_0+t is given by the

$$P_{01}(t)=1-e^{-\lambda t}. \tag{2}$$

Let us consider that during on-line periodic testing, the SBST program with execution time D is scheduled for execution at regular time intervals T, where $T>0$ and $T\gg D$. During the

TABLE I
REPRESENTATIVE CASES OF MEAN FAULT DETECTION LATENCIES

Cases	λ (ms^{-1})	μ (ms^{-1})	1/λ (ms)	1/μ (ms)	L(T) (ms) for T = 100 ms	L(T) (ms) for T = 1,000 ms	L(T) (ms) for T = 10,000 ms
1	0.001	1	1,000	1	100,100	1,001,000	10,010,000
2	0.01	1	100	1	10,100	101,000	1,010,000
3	0.001	0	1,000	∞	1,051	1,582	10,000
4	0.01	0	100	∞	158	1,000	10,000
5	0.001	0.001	1,000	1,000	1,103	2,313	20,000
6	1	1	1	1	200	2,000	20,000

repeated time intervals T, user processes are executed while, during the repeated time intervals D, the SBST program is executed. Intermittent and permanent faults are investigated only during the execution of the SBST test program, that is, only during the specific time intervals D, which can be considered approximately as instances of time, since $T \gg D$ [16]. Therefore, if we assume that the embedded processor is at state 0 at time t_0 and an intermittent fault occurs, it will be detected at one of the forthcoming time instances $t_0+T, t_0+2T, \ldots, t_0+nT, \ldots$, and the mean fault-detection latency $L(T)$ is

$$L(T) = \sum_{j=0}^{\infty}(j+1)T\left[P_{00}(T)\right]^j P_{01}(T) = \frac{T}{P_{01}(T)}. \quad (3)$$

From relations (1) and (3), it is derived that the mean fault-detection latency $L(T)$ for intermittent faults is

$$L(T) = \frac{T(\lambda+\mu)}{\lambda\left(1-e^{-(\lambda+\mu)\mathrm{T}}\right)}. \quad (4)$$

Following the same reasoning, from relations (2) and (3), it is derived that the mean fault-detection latency $L(T)$ for permanent faults is

$$L(T) = \frac{T}{1-e^{-\lambda\mathrm{T}}}. \quad (5)$$

In Table I, we present some representative cases with specific values of the parameters λ and μ, as well as the time interval T between two consecutive test program executions in order to illustrate the meaning of relations (3) and (5). We have chosen the values of T to be much larger than the test program execution time D which is less than 1 ms, namely, 100, 1,000 (= 1 s) and 10 000 ms (= 10 s).

It is evident for all the cases that the test-program execution time must be as short as possible and less than a quantum time cycle in order to permit us to reduce the time interval T and, consequently, to reduce the fault-detection latency $L(T)$, since $L(T)$ decreases when T decreases.

In Case 1, we consider intermittent faults so that the processor stays at state 0 1000 times more than at state 1. The time that processor stays at state 0 and state 1 is exponentially distributed with mean values 1000 and 1 ms, respectively. In this case, the mean fault-detection latency $L(T)$ is $(\lambda+\mu)/\lambda$ times greater than T.

In Case 2, we consider intermittent faults so that the processor stays at state 0 100 times more than at state 1; that is, in Case 2 the processor is more faulty than in Case 1. The time that processor stays at states 0 and 1 is exponentially distributed with mean values 100 and 1 ms, respectively. In this case, the mean fault-detection latency $L(T)$ is also $(\lambda+\mu)/\lambda$ times greater than T. Comparing Cases 1 and 2, we conclude that in both cases the fault-detection latency $L(T)$ depends virtually linearly on the time interval T and it is smaller when the processor goes to the faulty state more frequently.

In Case 3, we consider permanent faults and we assume that the time the processor stays at state 0 is exponentially distributed with mean value 1000 ms, while $\mu = 0$. In this case, the mean fault-detection latency $L(T)$ is T divided by $(1-e^{-\lambda T})$.

In Case 4, we consider permanent faults and we assume that the time the processor stays at state 0 is exponentially distributed with mean value 100 ms, while $\mu = 0$; that is, in Case 4 the processor is more faulty than in Case 3. In this case, the mean fault-detection latency $L(T)$ also is T divided by $(1-e^{-\lambda T})$. Comparing Cases 3 and 4, we conclude that the fault-detection latency $L(T)$ depends on both the time interval T and the probability for the processor to be faulty at time T, $(1-e^{-\lambda \mathrm{T}})$, and it is smaller when the processor goes to the faulty state more frequently. Note that when the time interval T is much larger than the mean duration at state 0 $(1/\lambda)$, $L(T) = T$ since the processor has probability to be faulty at time T close on 1.

In Case 5, we consider intermittent faults so that the processor stays at state 0 the same time it stays at state 1. The time that processor stays either at state 0 or state 1 is exponentially distributed with mean value 1,000 ms. In this case, the mean fault-detection latency $L(T)$ is $2T$ divided by $(1-e^{-\lambda \mathrm{T}})$.

In Case 6, we consider intermittent faults so that the processor stays at state 0 the same time it stays at state 1. The time that processor stays either at state 0 or state 1 is exponentially distributed with mean value 1 ms, that is, in Case 6 the processor is more faulty than in Case 5. In this case, the mean fault-detection latency $L(T)$ is $2T$, since the time interval T is much larger than the mean duration at state 0 (1 ms) and the processor has probability to be faulty at time T close on 1. Comparing Cases 5 and 6, we conclude that the fault-detection latency $L(T)$ is smaller when the processor goes to the faulty state more frequently.

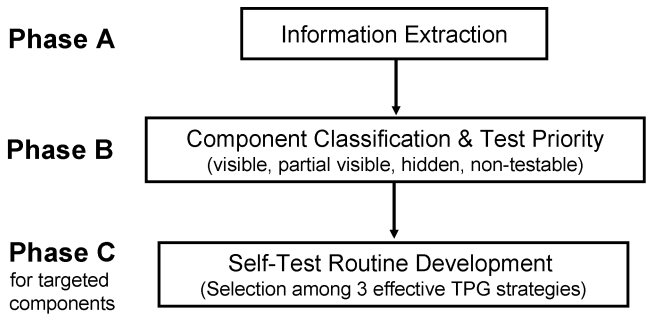

Fig. 2. SBST methodology for on-line periodic testing.

IV. SBST Methodology for On-Line Periodic Testing

The introduced here SBST methodology for on-line periodic testing of embedded processors consists of the three phases shown in Fig. 2, as follows.

Phase A: Identification of component operations and processor components *with the relevant multiplexers*, as well as, instructions that excite component operations and instructions (or instruction sequences) for controlling or observing processor registers.

Phase B: Classification of processor components in classes with the same properties from the aspect of Assembly language programmer and component prioritization for test development. This new classification scheme is well suitable for the systematic selection of a convenient TPG strategy for test pattern derivation, as well as, the systematic transformation of test patterns to a self-test routine.

Phase C: For the targeted components. Development of self-test routines based on specific self-test routine code styles for the three effective TPG strategies with respect to the stringent characteristics for on-line periodic testing. Selection of the most effective TPG strategy for on-line periodic testing.

These phases are explained in more details in the following sections.

A. Information Extraction

The starting point of our SBST methodology is the instruction format derived from the processor instruction set architecture (ISA) and the low register transfer level RTL description of the processor micro-operations derived from the RTL description of the processor.

Based on this information, we first identify the component operations and the processor components with specific inputs and outputs that carry out these component operations. At this stage, we identify possible multiplexers appearing in the inputs or the outputs of the processor components. Then, we map component/multiplexer inputs and outputs to internal temporary registers for multicycle datapaths or pipeline register fields for pipelined datapaths.

Then, we identify the instructions that carry out a specific operation and excite the pertinent processor component with the corresponding multiplexer(s) and register(s), if they exist.

Finally, we identify appropriate instruction(s) to control the values of processor component/multiplexer inputs or the corresponding registers; this is the controllability part of the methodology. Also, we identify instruction(s) to ensure propagation of the processor component/multiplexer outputs or the corresponding registers to processor primary outputs; this is the observability part of the methodology. Both the control and observe processes can be performed using single processor instructions or an instruction sequence.

B. Component Classification and Test Priority

From the information extracted in Phase A, we classify the processor components in the following four classes from the aspect of the Assembly language programmer:

Visible Components (VC): The components of the embedded processor with inputs and outputs *visible* to Assembly language programmer. For these components there is at least one instruction or instruction sequence that controls their inputs or the corresponding registers. Also, there is at least one instruction or instruction sequence that ensures propagation of their outputs or the corresponding registers to processor primary outputs.

These components are further classified in three subclasses according to the type of their inputs and outputs (data or addresses), as follows.

Data VC (D-VC): The inputs of these components receive data test patterns that can be stored: 1) in fields of an instruction format with immediate addressing mode; 2) in register file with an instruction with register addressing mode; 3) in data memory; or 4) in controllable data registers (i.e., data registers directly connected to register file or data memory). The outputs of these components produce data responses that can be stored: 1) in register file; 2) in data memory; or 3) in observable data registers (i.e., data registers directly connected to register file or data memory). Such components are *data processing components* (ALUs, shifters, multipliers, dividers, etc.) and *data storage components* (register file and special or temporary internal data registers). In this class, the *multiplexers* also belong at the inputs or the outputs of these components. The D-VCs have the highest test priority since they have the highest testability and dominate processor area. These components are well suitable for on-line periodic testing.

Address VC (A-VC): The inputs and outputs of these components receive addresses of memory system. The values of these addresses depend on the memory positions, where the instructions or the data will store. Thus, these components become visible with convenient storing of instructions or data in memory system. Such components usually appear inside the *instruction fetch unit* (e.g., the program counter) and the *data memory controller* (e.g., the memory address register). In this class, also belong special or temporary internal address registers, as well as, the *multiplexers* at the inputs or the outputs of these components. These components have low to medium testability, due to inherent address space limitations and are not suitable for on-line periodic testing, since they require a lot of distributed memory references and the derived self-test routines can not take advantage of temporal and spatial locality which reduces the cache miss overhead. A-VCs occupy a very small part of processor area and are partially tested with the addresses of the stored SBST program (code

and data) in memory system as a side-effect of testing the D-VCs.

Mixed (address-data) VC (M-VC): These components have inputs and/or outputs of both types (address or data) and become visible as mentioned above. For example, in this subclass belongs the *adder* used for PC-relative addressing implementation. These components have about the same characteristics with A-VCs.

Partially VC (PVC): These are the components of the embedded processor that generate control signals. Since the control outputs of these components affect the operation of VC, these components can be considered as *partially visible* to the Assembly language programmer. Such components are the processor control logic and small distributed controllers usually implemented as *finite-state machines*. These components have medium to high testability. To test such components we adopt simple high level functional tests like the application of all not already applied instruction opcodes for the case of testing the processor control logic, as well as, the application of instructions that achieve the most possible RTL code coverage for the case of testing a specific distributed finite-state machine. The PVCs occupy a small part of processor area and are usually suitable for on-line periodic testing.

Hidden Components (HC): The components of the embedded processor that are added in a processor architecture usually to increase its *performance*, but they are not visible to the assembly language programmer. These components consist of subcomponents which are further classified in three subclasses as follows.

Data visible hidden subcomponents (D-VHSC): These subcomponents receive data test patterns like the D-VCs and produce data responses that can be finally stored in register file or data memory. Such subcomponents are the *data fields and the immediate fields of pipeline registers* with the corresponding *multiplexers*. The D-VHSCs have high test priority, since they have high testability. These subcomponents are sufficiently tested since they receive plenty of data during testing of D-VCs. The data (or immediate) fields of pipeline registers are usually tested while the corresponding pipeline multiplexers can be easily tested. In any case they can be considered as well suitable for on-line periodic testing.

Address visible hidden subcomponents (A-VHSC): These subcomponents receive addresses of the memory system like the A-VCs. Such subcomponents are the *address fields of pipeline registers* with the corresponding *multiplexers*. The A-VHSCs have low to medium testability due to inherent address space limitations. These subcomponents are not suitable for on-line periodic testing like the A-VCs and are partially tested with the addresses of the stored SBST program (code and data) in memory system as a side-effect of testing the D-VC's. Note that in this category we also include the address fields of pipeline registers that store addresses of the register file with the corresponding multiplexers. These address fields of pipeline registers and the corresponding multiplexers are tested without any effort, since we exercise all registers of the register file during SBST.

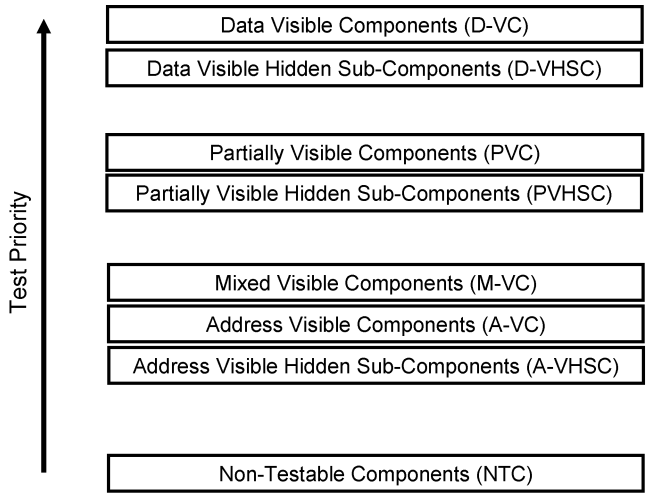

Fig. 3. Processor component test priority.

Partially visible hidden subcomponents (PVHSC): These subcomponents generate control signals that affect the operation of visible subcomponents inside a hidden component. Such subcomponents are the pipeline control units (e.g., forwarding unit, hazard detection unit) and the branch-prediction mechanism. The PVHSCs have medium to high testability and are tested in the same way as the PVCs. These components are usually suitable for on-line periodic testing.

Nontestable Components (NTC): The components of the embedded processor which can not be tested by executing only self-test programs developed by the Assembly language programmer. In the components are included, for example, registers, multiplexers, and control logic to handle interrupts and exceptions.

Based on the above-mentioned classification scheme of the processor components, we summarize the test priority of the processor component classes and subclasses as shown in Fig. 3. Components of the same class (or subclass) are further classified by size. We remark that we proceed to the next component in the same class (or subclass) or the next class (or subclass) only in case that the fault coverage is not acceptable. High test priority has the D-VCs and the D-VHSCs. Medium to high test priority has the PVCs and the PVHSCs. Low test priority in the sense that they are not suitable for on-line periodic testing and they will be targeted for SBST only in case that the fault coverage is not acceptable, have the A-VCs, the M-VCs, and the A-VHSCs. The NTCs can not be considered for SBST.

At this point, we remark that the data VC and the data visible hidden subcomponents dominates the processor area in various simple embedded processors (like AVR or 8051) or complex embedded processors (like MIPS) commonly used in low-cost embedded systems. Therefore, in practice testing of data VC or subcomponents usually leads to acceptable fault coverage since other processor components are partially tested as a side effect. This fault coverage can be slightly increased, if it is necessary, by further testing the PVCs and the PVHSCs that occupy a small processor area adopting the well known functional test techniques as mentioned above. In what follows we concentrate our discussion in the case of self-test routine development for the

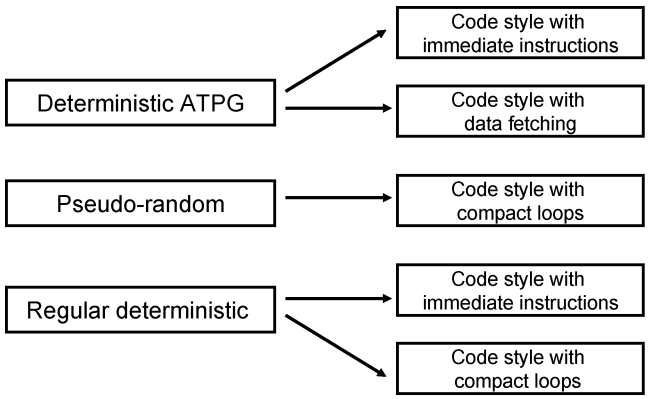

Fig. 4. Effective TPG strategies and the corresponding code styles.

most critical to test class of processor components, the data VC or subcomponents.

C. Self-Test Routine Development

The self-test routine development starts with test-pattern derivation and continues with the transformation of test patterns to self-test routines, which satisfies the test program requirements mentioned above for on-line periodic testing. Test patterns and the corresponding self-test routine code styles are derived according to the following three effective TPG strategies, as summarized in Fig. 4.

Deterministic ATPG-Based TPG Strategy: The first TPG strategy is based on deterministic ATPG and is usually applied to combinational data visible (sub)components, where instruction-imposed constraint ATPG provided by commercial tools is feasible. This strategy is a low gate-level strategy, since it requires the knowledge of the gate-level structure of the embedded processor.

The ATPG-based test patterns are transformed to effective self-test routines with two alternative ways: 1) the test patterns are transformed to instructions supporting immediate addressing or 2) the test patterns are stored in memory system and a loop-based self-test routine fetches these from the memory system and applies to the CUT.

Let us assume that the instruction *function* with register addressing carries out the specific operation function and excite the pertinent data visible (sub)component with the corresponding multiplexer(s) and register(s), if they exist. The self-test routine code style that generates a small number of n test patterns for data visible (sub)components with two inputs X and Y by using instructions with immediate addressing mode, is shown in Fig. 5 (following the MIPS Assembly language in all similar figures).

Test patterns are loaded in registers using the load immediate (`li`) pseudoinstruction, which the assembler decomposes to instructions `lui` and `ori` without transferring data from memory. After test pattern application, test responses are compacted by using a compaction routine, usually a software MISR routine with negligible aliasing to avoid transferring of data to memory that imposes data cache miss. At the end, the final signature is unloaded to data memory at the address `signature_address+signature_displacement` for error identification.

```
li $s0, pattern_X_1;                          # pattern X1
li $s1, pattern_Y_1;                          # pattern Y1
function $s2 $s0, $s1;                        # test application
jal compaction_routine_address;               # response compaction
        ......
li $s0, pattern_X_n;                          # pattern Xn
li $s1, pattern_Y_n;                          # pattern Yn
function $s2 $s0, $s1;                        # test application
jal compaction_routine_address;               # response compaction
li $s3, signature_address;
sw $s2, (signature_displacement) ($s3);       # signature storing
```

Fig. 5. ATPG-based code style with immediate instructions.

```
li $s3, first_pattern_address;
addi $s4, $zero, number_of_test_patterns
add $t0, $zero, $zero;
test_pattern_loop:
      lw $s0, 0($s3);                         # pattern X
      addiu $s3, $s3, 0x0004;                 #  assuming 32-bit data
      lw $s1, 0($s3);                         # pattern_Y
      addiu $s3, $s3, 0x0004;
      function $s2 $s0, $s1;                  # test application
      jal compaction_routine_address;         # response compaction
      addiu $t0, $t0, 0x0001;
      bne $s4, $t0, test_pattern_loop;
li $s5, signature_address;
sw $s2, (signature_displacement) ($s5);       # signature storing
```

Fig. 6. ATPG-based code style with data fetching.

This self-test routine has the following characteristics.

1) The code size depends linearly on the number of test patterns. If the number of test patterns is small enough, the code size is small as well.
2) Code without unresolved data hazards (assuming that the processor supports forwarding) with sequentially executed instructions that take advantage of spatial locality.
3) There is a high instruction-miss rate alleviating by the spatial locality and depending linearly on the number of test patterns.
4) The absence of data memory references and the use of only one store data memory reference result in zero data cache miss.

Alternatively, the self-test routine code style that generates a small number of n test patterns for data visible (sub)components with two inputs X and Y, by using data fetching from the memory system and a loop-based code, is shown in Fig. 6.

We assume that the first test pattern is stored in the data-memory address `first_pattern_address`, as well as the number of test patterns for both inputs is `number_of_test_patterns`. All test patterns first are loaded in the register file, then are applied to the CUT and afterwards test responses are compacted. At the end the final signature is unloaded to data memory for error identification.

This self-test routine has the following characteristics.

1) The code size is small and independent of the number of test patterns.
2) Code without unresolved data hazards (assuming that the processor supports forwarding) with compact loop that takes advantage of temporal locality.

```
li $s3, seed;
li $s4, polynomial;
addi $s5, $zero, number_of_test_patterns;
add $t0, $zero, $zero;
test_pattern_loop:
     # LFSR generation for $s0;            # generation of pattern X
     # LFSR generation for $s1;            # generation of pattern Y
     function $s2 $s0, $s1;                # test application
     addiu $t0, $t0, 0x0001;
     jal compaction_routine_address;       # response compaction
     bne $s5, $t0, test_pattern_loop;
li $s6, signature_address;
sw $s2, (signature_displacement) ($s6);    # signature storing
```

Fig. 7. Pseudorandom based code style.

3) There is low instruction-miss rate alleviating by the spatial locality and depending linearly on the number of test patterns.
4) There is high data-miss rate alleviating by the spatial locality and depending linearly on the number of test patterns.

Both alternative ways of effective self-test routines are used in practice for on-line periodic testing since they have short execution time, assuming that the number of ATPG-based test patterns is small. The former has a high instruction-miss rate, while the latter has a high data-miss rate. The selection is mainly based on test routine execution time and depends on the clock cycles per instruction (CPI) of the pertinent instructions and especially of the instruction that loads data from the data memory (like `lw`).

Pseudorandom-Based TPG Strategy: The second TPG strategy is based on pseudorandom TPG and is also usually applied to combinational data visible (sub)components with irregular structure, where instruction-imposed constraints can be taking into consideration. The pseudorandom test patterns are transformed to a loop-based software LFSR self-test routine. This strategy also is a low gate-level strategy, since it requires the knowledge of the gate-level structure of the embedded processor.

Let us assume an instruction *function* with register addressing that excite the pertinent data visible (sub) component as previously. The self-test routine code style that generates a large number of pseudorandom test patterns for data visible (sub)components with two inputs X and Y by using a loop-based code, is shown in Fig. 7.

We assume that the number of test patterns for both inputs is the `number_of_test_patterns`. Also, `seed` and `polynomial` are used by the software implemented LFSRs. All pseudorandom generated test patterns are applied to the CUT and afterwards test responses are compacted. At the end the final signature is unloaded to data memory for error identification.

This self-test routine has the following characteristics.

1) The code size is small and independent of the number of test patterns.
2) Code without unresolved data hazards (assuming that the processor supports forwarding) with compact loop takes advantage of temporal locality.
3) There is a low instruction-miss rate alleviating by the spatial locality and depending linearly on the number of test patterns.
4) The absence of data memory references and the use of only one store data memory reference result in zero data cache miss.

Pseudorandom-based self-test routines are used when a data visible (sub)component with irregular structure is considered for on-line periodic testing. Such test routines usually have large execution time, since processor components are random-pattern resistant and, thus, a large number of test patterns must be applied to reach acceptable fault coverage.

Regular Deterministic-Based TPG Strategy: The third TPG strategy is based on regular deterministic TPG [9], [10] that exploits the inherent regularity of the most critical to test processor data visible (sub)component like arithmetic and logic components, shifters, comparators, multiplexers, registers, and register files, which usually constitute the vast majority of processor components. In many cases, acceptable fault coverage is derived after testing only these components. This TPG strategy is a high-level strategy, since the derived test patterns are independent on gate-level implementation and constitute test sets of constant or linear size. The regular deterministic based test patterns are transformed to self-test routines with two alternative ways: 1) the test patterns are transformed to instructions supporting immediate addressing mode in case that the test set size is small enough or 2) the test patterns are transformed to a loop-based self-test routine with an initial value, a final value and a specific function to generate the regular transition from the one value to the other.

In case the number of regular deterministic test patterns is small enough, we use the processor instruction set with immediate addressing mode to generate and apply test patterns as it is shown in Fig. 5. This is also the case for the register file, where all registers of the register file must receive two patterns. In order to avoid stores in data memory the testing of register file is done in two phases, as follows. In the first phase, we test the one half of the register file by using registers of the other half for compaction, while in the second phase, we do the opposite.

Otherwise, let us assume an instruction *function* with register addressing that excite the pertinent data visible (sub)component as previous. The self-test routine code style that generates a small number of regular deterministic test patterns for data visible (sub)components with two inputs X and Y by using a loop based code, is shown in Fig. 8.

We have assumed that for every value of input X all values of input Y are applied to the CUT and, afterwards, test responses are compacted. In some cases, the final value is exactly the initial value and some instructions are deleted. At the end, the final signature is unloaded to data memory for error identification.

This self-test routine has the following characteristics.

1) The code size is small and independent of the number of test patterns.
2) Code without unresolved data hazards (assuming that the processor supports forwarding) with compact loop takes advantage of temporal locality.
3) There is a low instruction-miss rate alleviating by the spatial locality and depending linearly on the number of test patterns.

```
li $s0, initial_value_X;                    # initiate pattern X
li $s3, initial_value_Y;
li $s4, final_value_X;
li $s5, final_value_Y;
add $s1, $s3, $zero;                        # initiate pattern Y
test_pattern_loop:
     function $s2 $s0, $s1;                 # test application
     # generate next Y pattern in $s1;      # generation of next pattern Y
     jal compaction_routine_address;        # response compaction
     bne $s5, $s1, test_pattern_loop;
     add $s1, $s3, $zero;                   # initiate again pattern Y
     # generate next X pattern in $s0;      # generation of next pattern X
     bne $s4, $s0, test_pattern_loop;
li $s6, signature_address;
sw $s2, (signature_displacement) ($s6);     # signature storing
```

Fig. 8. Regular deterministic-based loop code style.

4) The absence of data memory references and the use of only one store data memory reference result in zero data cache miss.

We remark that in any case `nop` instructions are inserted accordingly when forwarding is not supported. Also, similar self-test routines can be derived without any effort if we consider an instruction *function_I* with immediate addressing to carry out the specific operation function.

TPG Strategy Selection for On-Line Periodic Testing: In general, the selection of a TPG strategy and a code style for on-line periodic testing of a specific processor data visible (sub)CUT is based on the following factors:

1) the maximum possible fault coverage that an effective TPG strategy can achieve for this CUT;
2) the minimum possible test-execution time of the corresponding self-test routine derived according to an effective TPG strategy with a specific code style;
3) the minimum possible self-test code/data size of the corresponding self-test routine derived according to an effective TPG strategy with a specific code style.

The second and the third factors depend on the ISA of the processor and thus the selection may vary among the processors. If the most important factor is the fault coverage we select the TPG strategy that achieves the maximum possible fault coverage. If there are two TPG strategies with similar fault coverage, we select the TPG strategy with the minimum possible test execution time, since it is the most important factor for on-line periodic testing. In any case, the three most effective TPG strategies, as mentioned above, are selected in turn as listed below:

1) The regular deterministic based TPG strategy is selected for a combinational or a sequential data visible (sub)CUT with inherent regularity, when the derived test-set size is small and leads to acceptable self-test code size and test execution time.
2) The deterministic ATPG based TPG strategy is selected for a combinational data visible (sub)CUT, when the derived test set size is small and leads to acceptable self-test code size (when the code style with immediate instructions is used) or self-test data size (when the code style with data fetching is used), and shorter test-execution time than the regular deterministic based TPG strategy.
3) The pseudorandom based TPG strategy is selected for a combinational data visible (sub)CUT with irregular structure, when the derived usually large test set size leads to smaller self-test code size and comparable test execution time with respect to the other TPG strategies.

V. Experimental Results

In this session, we demonstrate the effectiveness of the proposed SBST methodology for on-line periodic testing presenting experimental results for two RISC pipeline processor with different architecture:

1) A 32-bit embedded RISC processor core of MIPS architecture named "Plasma" that implements 3-stage pipeline with forwarding [13]. The Plasma core has been enhanced with a fast parallel multiplier [14], a serial divider, and a complex data-memory controller that supports load and store instructions of one byte, half-word or word, and implements a simple pipeline mechanism.
2) A 32-bit embedded RISC processor core of Concert'02 architecture named Jam that fully implements five-stage pipeline forwarding and pipeline hazard checking [17]. The Jam core has a serial multiplier and no divider at all, two simple memory controllers (termed memory access units), a more complex immediate extender to support four different instruction modes, and implements a much more complex pipeline mechanism.

The Mentor Graphics suite was used for VHDL synthesis, and functional and fault simulation (Leonardo, ModelSim, and FlexTest products, respectively).

A. Plasma Processor Core

The Plasma core was synthesized with area optimization at 26 080 gates targeting a 0.35-um technology library. The design runs at a clock frequency of 57 MHz. The CUTs with the highest priority for on-line periodic testing are the D-VCs (parallel multiplier, serial divider, register file, shifter and ALU), the PVC (control logic) and the memory controller which is 73% D-VC (memory data register and data multiplexers), 23% A-VC (memory address register), and 4% PVC (special control). The D-VCs dominate the processor area (92%).

TABLE II
COMPONENT-GATE COUNT AND CLASSIFICATION, SELF-TEST PROGRAM STATISTICS AND FAULT COVERAGE OF MIPS PLASMA CORE FOR ON-LINE PERIODIC TESTING

Component	Gate Count (gates)	Classification	Code Style	Size (words)	CPU Clock Cycles	Data Refer	FC (%)	Miss. FC (%)
Par. Mul. – Ser. Div.	11,601	D-VC	RegD (L + I)	68	6,848	2	96.3	1.8
Register File	9,905	D-VC	RegD (I)	278	1,302	1	97.8	0.7
Memory controller	1,119	73% D-VC	RegD (I)	113	357	80+1	90.3	0.3
Shifter	682	D-VC	AtpgD (I)	195	571	1	99.9	0.0
ALU	491	D-VC	RegD (L + I)	61	582	1	96.8	0.1
Control Logic	230	PVC	FT	85	245	1	89.3	0.1
Pipeline	885	HC					98.4	0.0
Remaining	1,167	39% D-VC		8			63.1	1.4
Total	**26,080**	**92% D-VC**		**808**	**9,905**	**87**	**95.6**	**4.4**

We have applied to all CUTs the most effective TPG strategies with low development cost and have achieved an acceptable high fault coverage of 95.6%. We have used regular deterministic code style (RegD) with loops (L) or immediate type instructions (I) for all D-VC's except the shifter where we use ATPG deterministic code style (AtpgD) with immediate type instructions (I). Also, we have used functional tests (FT) like the application of all not already applied instruction opcodes for control logic. For every self-test routine, a final signature is derived after compaction of all responses by using a shared software MISR routine of eight words. At the end of periodic testing seven signatures, one for every CUT, are unloaded to data memory for fault detection.

Component gate count and classification, self-test program statistics (code style, program size in words, CPU clock cycles, and data-memory references—loads and stores), along with the achieved single stuck-at fault coverage and the percentage of the processor overall fault coverage which is missing from each of the CUTs, are presented in specific columns of Table II, respectively.

The derived self-test program for on-line periodic testing has the required stringent characteristics:

1) a very small code of only 808 words without pipeline stalls that takes advantage of temporal locality and spatial locality;
2) a small number of only 87 memory data references that imposes a small number of data cache misses. The only CUT that requires 80 loads and stores for test application is the memory controller.
3) a very short CPU execution time of 9905 clock cycles. Assuming an average instruction/data cache miss rate of 5% and a miss penalty of 20 clock cycles, the derived test execution time is less than 11.000 CPU clock cycles or less than 0.2 ms, which is much less than a quantum time cycle.

B. Jam Processor Core

The Jam core was synthesized with area optimization at 43 208 gates targeting a 0.35-um technology library. The design runs at a clock frequency of 41.8 MHz. The CUTs with the highest priority for on-line periodic testing are the D-VCs (register file, integer unit, and immediate extender), the PVC (control logic), and the memory access unit (MAU) 2 that accesses the data memory and is 64% D-VC. The integer unit implements the following integer operations: multiplication, addition, subtraction, bitwise OR, bitwise AND, bitwise XOR, and shift. All integer operations apart from multiplication take one cycle, while multiplication takes 33 cycles (a serial multiplier is implemented). The 55% of pipeline registers belong to D-VHSC and are fully testable, while the remaining part of pipeline register mainly includes address fields for instruction memory and register file. The address fields for register file are fully testable, while the address fields for instruction memory are partially testable due to restrictions on instruction memory address space. The pipeline structure of the Jam core is implemented at the top level of the VHDL design hierarchy. It is therefore difficult to identify components that implement the pipeline mechanism other than the pipeline registers. The rest of the pipeline logic (control logic for hazard detection and multiplexers for forwarding) is not accounted as separate components, but as part of the remaining logic. The D-VCs (with pipeline subcomponents) dominate the processor area (86%).

We have applied to all CUTs the most effective TPG strategies with low development cost and we have achieved an acceptable high fault coverage of 94%. We have used regular deterministic code style (RegD) with loops (L) or immediate type instructions (I) for all D-VCs, except the shifter where we use ATPG deterministic code style (AtpgD) with immediate type instructions (I). Also, we have used functional tests (FT) for control logic and pipeline logic (control and multiplexers). For every self-test routine a final signature is derived after compaction of all responses by using a shared software MISR routine of eight words. At the end of periodic testing seven signatures, one for every CUT, are unloaded to data memory for fault detection. For the integer unit we receive three signatures, one for the ALU, one for the shifter, and one for the serial multiplier.

TABLE III
COMPONENT GATE COUNT AND CLASSIFICATION, SELF-TEST PROGRAM STATISTICS AND FAULT COVERAGE OF THE JAM CORE FOR ON-LINE PERIODIC TESTING

Component	Gate Count (gates)	Classification	Code Style	Size (words)	CPU Clock Cycles	Data Refer	FC (%)	Miss. FC (%)
Register File	22,917	D-VC	RegD (I)	483	2,101	1	98.1	1.0
Integer Unit	5,698	D-VC	Reg/AtpgD(I+L)	276	5,560	3	98.9	0.2
Immediate Extender	269	D-VC	RegD (I)	37	126	1	98.5	0.0
MAU 1 (instructions)	576	A-VC, PVC					69.4	0.4
MAU 2 (data)	576	64% D-VC	RegD (I)	45	280	20+1	81.7	0.2
Control Logic	388	PVC	FT	215	809	1	81.2	0.2
Pipeline Registers	3,771	55% D-VHSC					89.7	0.9
Remaining	9,013	HC		8			85.0	3.1
Total	**43,208**	**86% D-VC**		**1,064**	**8,876**	**27**	**94.0**	**6.0**

Component gate count and classification, self-test program statistics (code style, program size in words, CPU clock cycles, and data memory references—loads and stores), along with the achieved single stuck-at fault coverage and the percentage of the processor overall fault coverage which is missing from each of the CUTs are presented in specific columns of Table III, respectively.

The derived self-test program for on-line periodic testing has the required stringent characteristics:

1) a very small code of only 1064 words without pipeline stalls that takes advantage of temporal locality and spatial locality;
2) a small number of only 27 memory data references that imposes a small number of data cache misses. The only CUT that requires 20 loads and stores for test application is the MAU 2.
3) a very short CPU execution time of 8876 clock cycles. Assuming an average instruction/data cache miss rate of 5% and a miss penalty of 20 clock cycles, the derived test execution time is about 10.000 CPU clock cycles or about 0.24 ms, which is much less than a quantum time cycle.

VI. CONCLUSION

The application of SBST to on-line periodic testing of embedded processor cores, trades off between fault-detection latency and performance overhead, detects both permanent and intermittent faults, and is well suitable for low-cost nonsafety-critical embedded systems that do not require immediate fault detection.

In this paper, we have shown that the SBST methodology for low-cost embedded processors introduced here results in very effective SBST strategies for on-line periodic testing. Both types of permanent and intermittent faults with acceptable high fault coverage are detected by a small embedded test program with test execution time much less than a quantum time cycle. SBST for on-line periodic testing can be applied to improve reliability of low-cost nonsafety-critical embedded systems based on embedded processors, where high cost hardware, information, software, or time redundancy mechanisms can not be applied.

REFERENCES

[1] H. Al-Assad, B. T. Murray, and J. P. Hayes, "Online BIST for embedded systems," *IEEE Des. Test Comput.*, vol. 15, no. 4, pp. 17–24, Oct.–Dec. 1998.
[2] M. Nicolaidis and Y. Zorian, "On-line testing for VLSI—a compendium of approaches," *J. Electron. Testing: Theory Applicat.*, vol. 12, no. 1–2, pp. 7–20, 1998.
[3] N. Oh and E. J. McCluskey, "Error detection by selective procedure call duplication for low energy consumption," *IEEE Trans. Reliab.*, vol. 51, no. 4, pp. 392–402, Dec. 2002.
[4] G. Xenoulis, D. Gizopoulos, N. Kranitis, and A. Paschalis, "Low-cost on-line software-based self-testing for embedded processor cores," in *Proc. IEEE Int. On-Line Testing Symp.*, 2003, pp. 149–154.
[5] J. Shen and J. Abraham, "Native mode functional test generation for processors with applications to self-test and design validation," in *Proc. IEEE Int. Test Conf.*, 1998, pp. 990–999.
[6] K. Batcher and C. Papachristou, "Instruction randomization self test for processor cores," in *Proc. VLSI Test Symp.*, 1999, pp. 34–40.
[7] P. Parvathala, K. Maneparambil, and W. Lindsay, "FRITS—A microprocessor functional BIST method," in *Proc. IEEE Int. Test Conf.*, 2002, pp. 590–598.
[8] L. Chen and S. Dey, "Software-based self-testing methodology for processor cores," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. 20, no. 3, pp. 369–380, Mar. 2001.
[9] N. Kranitis, D. Gizopoulos, A. Paschalis, and Y. Zorian, "Instruction-based self-testing of processor cores," in *Proc. IEEE VLSI Test Symp.*, 2002, pp. 223–228.
[10] N. Kranitis, G. Xenoulis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Application and analysis of RT-level software-based self-testing for embedded processor cores," in *Proc. IEEE Int. Test Conf.*, 2003, pp. 431–440.
[11] J. Hennessy and D. Patterson, *Computer Architecture A Quantitative Approach*. San Francisco, CA: Morgan Kaufman, 1996.
[12] Intel Corporation, "Mobile Power Guidelines 2000,", Dec. 11, 1998.
[13] Plasma CPU Model. [Online]. Available: http://www.opencores.org/projects/mips
[14] J. Phil and E. Sand. Arithmetic Module Generator for High Performance VLSI Designs. [Online]. Available: http://www.fysel.ntnu.no/modgen
[15] S. Y. H. Su, I. Koren, and Y. K. Malaiya, "A continuous-parameter Markov model and detection procedures for intermittent faults," *IEEE Trans. Computers*, vol. c-27, no. 6, pp. 567–570, June 1978.
[16] T. Nakagawa and K. Yasui, "Optimal testing-policies for intermittent faults," *IEEE Trans. Reliab.*, vol. 38, no. 5, pp. 577–580, Dec. 1989.
[17] J. E. Thelin, A. Lindstrom, and M. Nordseth, *Concert'02 Architecture Specification and Implementation*. Goeteborg, Sweden: Chalmers Univ. Technology, 2002.

Antonis Paschalis (M'97–A'97) received the B.Sc. degree in physics, the M.Sc. degree in electronics and computers, and the Ph.D. degree in computers, from the University of Athens, Athens, Greece.

He is Associate Professor in the Department of Informatics and Telecommunications, University of Athens. He was previously a Senior Researcher with the Institute of Informatics and Telecommunications, National Research Centre "Demokritos," Athens, Greece. He has published over 100 papers and holds a U.S. patent. His current research interests are in logic design and architecture, very large scale integration testing, processor testing, and hardware fault-tolerance.

Prof. Paschalis is a Member of the Editorial Board of *JETTA* and is the Vice Chair of the Communications Group of the IEEE Computer Society Test Technology Technical Council. He has participated in several organizing and program committees of international events in the area of design and test.

Dimitris Gizopoulos (S'93–M'97–SM'03) received the received the B.S. degree in computer engineering from the University of Patras, Patras, Greece, and the Ph.D. degree from the University of Athens, Athens, Greece.

He is Assistant Professor in the Department of Informatics, University of Piraeus, Piraeus, Greece. He is the author of more than 60 technical papers in transactions, journals, book chapters, and conferences, and is the author of a book. He is also a co-inventor of a U.S. patent. His research interests include processor testing, design-for-testability, self-testing, on-line testing, and fault tolerance of digital circuits.

Prof. Gizopoulos is a Member of the Editorial Board of the *IEEE Design and Test of Computers* magazine. He was the guest editor of several IEEE special issue publications. He is a Member of the Steering, Organizing, and Program Committees of several test technology technical events, the Executive Committee of the IEEE Computer Society Test Technology Technical Council, and a Golden Core Member of the IEEE Computer Society.