A Processor and Cache Online Self-Testing Methodology for OS-Managed Platform

Ching-Wen Lin and Chung-Ho Chen, *Member, IEEE*

***Abstract*— Software-based self-test (SBST) is an effective method to detect operational faults of a processor system. We propose an architectural approach to support high fault-coverage online SBST: Processor Shield, which tackles the difficult-to-test issues raised due to the protection of an operating system. The processor shield, including a software framework and design for testing hardware, creates an online self-testing environment without influencing other processes and on-bus devices even if the SBST fails. We present a case study that demonstrates SBST executions under Linux kernel on an ARMv5-compatible processor system. For CPU testing, the stuck-at fault coverage is over 99% while the transition fault coverage is higher than 93%. For cache control logic testing, the stuck-at fault coverage is over 99% while the transition fault coverage is higher than 95%. For RAM module testing, the fault coverage is nearly 100%. Cache SBSTs finish in a context-switch interval of less than 4 ms while CPU SBST finishes in less than 8 ms for 1-GHz clock. The hardware overhead of the processor shield is only 0.494% of the whole processor area. We also present an SBST-dynamic voltage and frequency scaling application that calibrates the dynamic minimal guardbands and helps achieving lower power consumption and mitigating transistor-aging effect.**

***Index Terms*— Design for testing (DFT), guardband, online testing, software-based self-test (SBST), transistor aging.**

I. Introduction

RELIABILITY of an SoC chip will be degraded by operational faults during chip lifetime. Operational faults can be classified into three categories: permanent faults, intermittent faults, and transient faults [1]. Permanent faults infinitely generate irreversible faulty effects at the same location. Although at first the manufacturing test can identify known-good dies, permanent faults still appear and violate the chip normal operations.

A permanent fault, which may eventually crash the system, significantly hammers system reliability. Our aim is to capture permanent faults of an embedded processor even if the system still performs its normal functions. We analyze the stuck-at fault coverage for the Linux kernel booting process on an ARMv5 processor implemented in Verilog. During Linux booting, all the bus input–output signals are recorded. This record along with the processor netlist is used to perform fault simulation by Synopsys TetraMAX [35]. Even though Linux kernel booting requires nearly two billion instructions, including all the supported instruction types, the processor fault coverage is only 90% of the stuck-at faults. Observing this test result, there exists a substantial 10% of possible latent faults that are not tested during the booting process.

In order to detect permanent faults of a processor system, an effective method is to use the software-based self-test (SBST) to capture faults by executing a test program through the processor itself. Test programs are typically dedicated to detect structural faults, including stuck-at fault, transition fault, path delay fault, etc. These test programs for a processor core can be classified into two forms according to their developing methods: deterministic or random.

A deterministic test program can use fewer instructions to quickly detect faults in the target components [2]–[9]. For instance, an approach is described in [5] focusing on the branch prediction unit test. In [8], a method is proposed to test the forwarding unit in a pipelined processor. There are also several methods proposed to test the whole processor core [10]–[15].

Another way to develop deterministic programs is to use automatic generation tools [16]–[19]. The programs generated by the tools can typically achieve high fault-coverage for manufacturing testing. However, this places some strict constrains for testing environment and leads to many challenges for online testing. For instance, in an online test scenario, the work proposed in [19] can achieve 98% of fault efficiency (excluding the un-testable faults) for the miniMIPS core. The authors built the memory access model with the constrained interaction between the processor and the system bus. For an online test, these constraints are typically difficult to meet as there are unpredictable memory latencies and bus contentions resulting from other bus masters in an SoC design.

A test program can be generated by a random instruction generator, which includes all the supported instructions [12], [13], [20]. This method can produce special instruction sequences that are hard to appear in deterministic programming. Hence, for the desired high fault-coverage, an effective method is to combine deterministic and random programming methods to generate the test programs [12], [13]. In this paper, we employ this hybrid method to generate the SBST programs [12].

For RAM cell testing, March algorithms are the most popular methods [21], [22]. March algorithm is a sequence of March elements composed of pre-defined read/write operations, which are called March read/write, applied to

Manuscript received November 6, 2016; revised February 24, 2017; accepted April 7, 2017. Date of publication May 10, 2017; date of current version July 24, 2017. This work was supported by the Ministry of Science and Technology, Taiwan, under Grant MOST 103-2221-E-006-266-MY3.

The authors are with the Institute of Computer and Communication Engineering, Department of Electrical Engineering, National Cheng Kung University, Tainan City 70101, Taiwan (e-mail: kliolin@mail.ee.ncku.edu.tw; chchen@mail.ncku.edu.tw).

Digital Object Identifier 10.1109/TVLSI.2017.2698506

the test target memory cells by special addressing orders, including ascending, descending, and either. In the word-oriented March algorithm [23], the values, which are used in the March read/write, are called March data backgrounds (March DBs) or their complementary data backgrounds (March CDBs). To test a cache system, March algorithm has to be performed on the target RAM modules, including virtual tag RAM, physical tag RAM, and data RAM.

A high fault-coverage online SBST program requires the capability to test all of the functions of the target processor. However, to run such an SBST program under an operating system (OS) without violating normal system behaviors is quite challenging.

In order to test the processor on a platform governed by an OS, the first system related issue is system memory mapping, including virtual address translation and physical memory layout or allocation. An OS manages the virtual memory, and a test process has to acquire the required memory space from the OS. The physical memory layout decides where a physical address is allocated onto the main memory, the memory-mapped I/O (MMIO), or the reserved space. This mapping relation is typically recorded in the hardware or platform configuration. We call the address, which is required for testing but its access is limited by system memory allocation or protection, shielded address. In order to achieve high fault-coverage, accessing all the required addresses, including shielded addresses, is an essential operation.

The second system related issue is interrupt testing and its service routine. A preemptive OS may suspend SBST processes to serve a higher priority interrupt. This changes the expected SBST instruction flow and compromises SBST test results. Disabling interrupt is an intuitive way to prevent SBST processes from being affected. However, interrupt handling circuits, for instance, switching to a privilege mode, invoking shadow register use, and preserving the interrupted status, must be tested for high fault-coverage. In order to test interrupt logic and circuit, an interrupt signal is raised, and then the processor branches the service routine prepared by the SBST process.

The third system related issue is the need to block faulty actions, which may disturb contexts of other processes or on-bus devices during online testing. When capturing a fault, a responsible online test strategy records the test report in a non-volatile memory, e.g., flash, and notifies the system manager, instead of crashing the system. For online testing, the major faulty actions are faulty memory accesses that may result in a system crash or generating irreversible damages to the on-bus devices. Therefore, preventing a faulty access coming from the processor or cache controller is a crucial step when performing an online test.

In this paper, we propose an efficient method to create an environment for high fault-coverage SBST testing, including CPU and caches, on an SoC platform governed by an OS. This environment can overcome the above system related issues without modifying the internal design of the processor or changing the target instruction set architecture (ISA). The proposed method can seamlessly switch the SBST process and the kernel process.

Our method is called processor shield, including a software framework and design for testing (DFT) hardware. The processor shield technology can support tests for CPU and caches respectively. The software framework constructs a non-preemptive privileged process to request the required resources from the OS. The framework performs the SBST programs and then returns the control to the OS as if the test were never executed. The DFT hardware blocks the faulty accesses and redirects shielded accesses to the available memory space prepared by the software framework.

We present a case study that performs SBST programs to test the processor core and cache system under Linux kernel on an ARMv5-compatible processor system. For CPU online testing, the achieved stuck-at fault coverage is 99.01% while the transition fault coverage is 93.25%. For cache control logic testing, the stuck-at fault coverage is higher than 99%, and the transition fault coverage is over 95%. For data cache RAM modules, the fault coverage is 100%. For instruction cache RAM modules, the fault coverage is 99.99%. The DFT hardware overhead is 0.494% of the whole processor area. The proposed processor shield design can effectively support SBST programs to execute under Linux kernel and successfully achieve the expected high fault coverages without interfering with other processes and on-bus devices.

The rest of this paper is organized as follows. Section II discusses the related work. Section III discusses the online SBST challenges for processor core and caches. Section IV describes the proposed processor shield. Section V demonstrates a case study and the experimental result. Section VI presents several fault-coverage enhancement methods and an SBST-dynamic voltage and frequency scaling application that can calibrate the required minimal guardband. Section VII presents a conclusion.

II. Related Work

Several online SBST methods have been proposed to deal with various problems on different platforms [1], [24]–[28]. Some works perform SBSTs without hardware support, and others leverage hardware to achieve their purposes. Table I shows the comparison of the proposed processor shield methodology and relevant online SBST techniques.

Bernardi *et al.* [24] have proposed an SBST development flow for automotive microcontroller. As the automotive design is a safety-critical system, all the self-test routines must be preempted at any time. They separated the processor design into many sub-modules and tested one sub-module at a time in order to meet the time constraint specification. Their test programs did not test the shielded addresses, and this work was not allowed to use absolute addresses during testing. As a result, the achieved fault coverage was less than 88% for the 32-bit power architecture.

Skitsas *et al.* [25] demonstrated a fine-grained selective testing method in a multicore system. They proposed that only the functional module which is recently stressed needs to be tested. They used many performance counters that counted the activation times of all function units. A function unit is tested when the corresponding counter value reaches a threshold. Their method can successfully run under an OS.

TABLE I
COMPARISON OF RELEVANT ONLINE TESTING METHODS

	Proposed Processor Shield		[24]	[25]	[1]	[26]	[27]	[28]
Target module	Core	Cache	Core	Function unit	Core	Core	Core	Cache
Target architecture	ARMv5 Architecture		32-bit Power Arch.	UltraSPARC III	32-bit MIPS	OpenSPARC T1	miniMIPS	OpenSPARC T1
Fault model	Stuck-at & Transition	RAM fault	Stuck-at	Stuck-at	Stuck-at	Stuck-at & Path-delay	Stuck-at	RAM fault
Fault coverage	Very high (99%)	99.99% / 100% (ICache / DCache)	Low (87%)	Low (88%)	High (95%)	Very high (99%)	Moderate (93%)	100%
Instruction count	High (1.1M)	Low (6K)	Low (55K)	Low (1.5k)	Low (800)	Moderate (600K)	Low (400~1K)	Low (1k)
Running test under an OS	Yes (Linux)		Yes (real time OS)	Yes	Not mentioned	Yes	Not mentioned	Not mentioned
Special ISA support	No		No	No	No	Yes	No	Yes
CPU circuit modification	No		No	Yes	No	Yes	No	No
Hardware insertion	Yes		No	Yes	No	Yes	Yes	No
Hardware overhead	Low (0.49%)		No	Low	No	High (5.8%) scan-chain	Low (0.5~1.6%)	No
Shielded address testing	Yes		No	Not mentioned	No	Yes	Yes	Yes
Interrupt testing	Yes		Yes	Not mentioned	No	Yes	Not mentioned	

However, testing for interrupt and shielded addresses was not mentioned in their research. The achieved fault coverage was relative low, i.e., 88% for UltraSPARC III.

Paschalis and Gizopoulos [1] proposed a periodic online testing method for embedded processors. They classified the component units according to the relationship of the data paths. During testing, the faults in control-path-bound units can be additionally observed when testing the data-path-bound devices. Their proposed test program features small code size and short execution time. However, interrupt and exception handler circuits were regarded as non-testable units. They did not discuss whether the same high fault-coverage was achieved when executing an SBST under an OS.

Constantinides *et al.* [26] presented an online testing method that utilized an enhanced ISA with special testing instructions. A DFT hardware was combined with scan chains to perform fault detection and isolation. Structural test patterns were applied to the DFT hardware by a software routine using the special testing instructions. Intrusive processor modifications were implemented to enable structural testing through the scan-chain infrastructure. Therefore, their design achieved very high fault coverage, i.e., 99% for OpenSPARC T1. However, this also leads to 5.8% area overhead, special instructions for testing, and the modification of the processor internal structure.

Bernardi *et al.* [27] proposed a hardware module, MIcroprocessor Hardware Self-Test unit (MIHST), which is used to support online SBST. In order to reduce code size, they encoded the SBST program. The encoded program is stored in the MIHST unit that dynamically decodes the program for testing. The MIHST unit can fully control the processor execution flow during testing and redirect all memory accesses to MIHST itself. However, this method blocks the normal operations of the system bus during testing. In addition, they proposed an online testing scenario using the MIHST unit, but they did not discuss interrupt testing and the impact on fault coverage when performing an SBST under an OS.

Theodorou *et al.* [28] proposed the direct cache access (DCA) instructions that can directly read/write all cache RAM cells, including tag and data RAM. The DCA instructions perform March sequence by reading/writing March DBs/CDBs from/to the target cache RAM cells. However, these DCA instructions are not fully supported in common ISAs, especially the instructions to directly read/write tag RAM. How to test the control logics of caches was not mentioned in their work.

The above prior works used techniques including DFT test assisted hardware, pure SBST test, or changing the target ISA. For some works, including [1], [24], [25], the authors did not discuss how to test the shielded addresses. The untested shielded addresses reduce the fault coverages of fetch unit and the load/store unit. For faulty effect isolation, pure SBST methods rely on memory management unit (MMU) or memory protection unit (MPU) to block the faulty memory accesses. However, for the faulty actions that do not query MMU, e.g., writing back a dirty line from a physical tagged cache, there is no way to prevent this from occurring.

Consequently, a DFT hardware is an efficient method to block all faulty operations during online testing. For DFT methods, including [26], [27], shielded address testing can be tested through the support of the DFT. However, there exists a tradeoff between the DFT hardware overheads and the fault coverage achieved. Our method can leverage fewer hardware areas to achieve very high fault coverage and prevent the system from being affected at the same time.

III. ONLINE SBST CHALLENGE FOR CPU AND CACHE

In the introduction, we have addressed three system related issues when conducting online SBST testing. In this section, we further present the challenges due to these

system related issues when developing and executing a native SBST program.

A. Challenge for CPU Core Online SBST

SBST programs typically bear the same requirements in order to achieve a high fault coverage, regardless of the development methods. The first is that all supported instructions, including their addressing modes, must be executed more than once. The processor mode needs to be changed to perform privileged instructions.

The second is interrupt handling circuit testing. Before testing interrupt circuits, an SBST process needs to build an interrupt vector table and service routines for testing. During interrupt testing, the system interrupt vector table must be redirected to the prepared one. For precise interrupt, an interrupt has to be triggered at the deterministic time with respect to the SBST instruction sequence. Hence, for online testing, an SBST process must both test the precise interrupt and maintain the expected test instruction sequence without being switched to other processes.

A processor core online SBST typically needs to obtain the access right for almost all of the memory space managed by an OS. Therefore, the third requirement is to access all the needed addresses, including shielded addresses, without violating the memory protection scheme of an OS [40].

B. Challenge for Cache System Online SBST

The cache SBSTs, introduced in the previous work [28]–[32], can be divided into four types of device testing: data RAM, virtual tag RAM, physical tag RAM, and control logic. For RAM cells, March algorithms are the most popular methods to capture various types of faults [21], [22]. A control logic SBST development is based on the cache control functions and the cache read/write policies.

1) Cache RAM Cell Testing: In order to test cache RAM cells, an important issue is to acquire the memory regions for the March read/write operations. For all tag RAM cells, including virtual and physical, the accessed memory addresses must conform to the March DBs/CDBs since the March DB is the address rather than the value. On an automatic test equipment (ATE), there is no problem in accessing the required addresses. However, a problem occurs when performing a cache SBST on a platform hosted by an OS.

This problem comes from the physical memory allocation, and we call it physical memory limitation. In order to test the physical tag RAM, the March algorithm needs to access the specified physical pages that conform to the March DBs/CDBs. These pages are required when testing the physical tag RAM so we call them PATAG (Physical Address TAG) pages. There are two limitations in performing March read/write accesses in the PATAG pages, and these limitations come from an OS and hardware platform.

The first limitation comes from the memory protection scheme of an OS. To acquire a specific physical memory page typically requires alteration of the current system state, for example, releasing of the requested page from the owner process. Specifically, if the required PATAG page is dispatched to another process by the OS, the SBST process has to either copy and protect this page or request the OS to release it. These two actions cause additional system overhead due to the copy function or the swap out routine. In addition, if the PATAG page is released by the OS, a page fault may occur when the original owner process resumes its operation.

The second limitation is the physical memory allocation on the target platform. When an SBST process obtains the right to access the PATAG pages, however, these physical pages might belong to the shielded addresses, which are not allocated in the main memory. Accessing these shielded addresses directly disturbs the testing execution flow, and the results cannot be used to examine the test.

Through the address redirection function, the proposed processor shield design can effectively tackle these two limitations without interfering with other processes.

2) Cache Controller Testing: Many cache control functions are typically dependent on the memory attributes such as the cache read/write policy specified in the page table. However, the OS-dispatched page table typically does not contain all the required memory attributes for testing. A feasible method is either to modify the page descriptors by system calls or to build a manual page table for cache controller testing [41].

IV. Proposed Processor Shield Design

In this paper, we propose an architecture level methodology called processor shield, which includes a software framework and DFT hardware. Before executing a native SBST program, the software framework has to initialize the testing environment. We call a native SBST program, which is developed by the methods introduced in the previous work [10]–[15], a testing kernel. The processor shield DFT hardware performs address redirection function, and all the memory accesses, including the shielded accesses, can be redirected to the memory regions acquired from the OS by the SBST process. This can also prevent other processes and MMIOs from being affected by the testing kernels or faulty actions. The DFT hardware employs a multiple-input signature register (MISR) to compress the processor output values [33], and includes a time-out counter. After a testing kernel finishes or is timed-out by DFT time-out counter, the OS can take over the control as if the testing kernel were never executed.

A. Processor Shield Software Framework

Constructing an SBST program and used as a system call, is a feasible method to obtain the privileged right to perform online testing, and we call this system call an SBST service routine. As shown in Fig. 1, an SBST service routine can be divided into three major parts: testing environment initialization, SBST body function, and result examination. Since the SBST service routine gets the privilege right to use all the supported instructions, we can directly deal with the first CPU SBST challenge discussed in Section III-A.

In testing initialization, an SBST service routine acquires several free pages from the OS. For instance, get_free_pages Linux system call allocates a continuous physical memory space and returns a virtual address pointer to the caller. These free pages are read-writable, and they are divided into three regions: code region, data region, and backup region. The code

region is used to store the interrupt vector table, interrupt service routines for testing, and SBST body function. The data region is used to store the required testing data and testing results. The processor state backup is stored in the backup region. Note that all the pages in the code region have to be executable before branching to the SBST body function stored in the code region.

The physical addresses of the above three regions are passed to the DFT hardware as configuration arguments. The relevant physical addresses can be obtained from the system calls. For instance, virt_to_phys Linux system call can return the corresponding physical address for a given virtual address.

Once an SBST service routine has prepared the memory regions and notified the DFT hardware of their physical addresses, the SBST service routine stores the current processor state, including all register contents, processor status, MMU status, and page table base pointer to the backup region. When restoring processor state from the backup region, this enables the SBST service routine to return to the state before entering the SBST body function.

Before calling the SBST body function, the cache must be flushed to make sure that the testing results are not compromised by these unrelated cache contents. Then, the SBST service routine branches to the SBST body function stored in the code region. The SBST body function first enables the DFT hardware and disables the interrupt, which makes the SBST body function non-preemptive. Then, there are different requirements for testing CPU and caches respectively. We will introduce them separately as follows.

For a high fault-coverage CPU SBST, the right to access almost all of the memory space is required. Disabling the MMU is an effective method, allowing the CPU to directly use logical addresses to fetch instructions and data without any restriction. However, disabling the MMU makes the following address access unpredictable. The proposed DFT hardware can maintain the expected execution flow when disabling the MMU since it can redirect all instruction fetches to the code region. Note that the disabling of the MMU is performed after the DFT hardware is activated; otherwise, this may result in a system crash.

For cache testing, the physical memory limitations mentioned in Section III-B need to be overcome. We propose an efficient method to access any required page without influencing the current system state. This method can be used to test all shielded addresses. First, an SBST service routine prepares a new page table, which contains the page descriptors of PATAG pages and other required pages. Then, the page table base pointer is changed from the OS-dispatched one to the prepared one by SBST body function. In this way, the processor can access all the required pages, including shielded pages, without causing any page fault.

However, the shielded pages cannot be directly accessed; nonetheless, this problem can be resolved by the DFT. For instance, the DFT hardware redirects all the memory accesses in PATAG pages (shielded pages) to the code or data region depending on the tested target cache. Therefore, all physical page numbers can be directly read/written from/to the physical tag RAM without adding any special instruction to the ISA. In this way, when testing shielded addresses, interference with the current system state can be minimized since the SBST service routine only acquires free pages without performing any memory protection operation introduced in Section III-B.

After the initialization stage, the SBST service routine branches to SBST body function allocated in the code region. Since the physical address mapping in the DFT is configured in the initialization stage, the address redirection function can be activated once the interrupt is disabled by the DFT. After configuring the MMU (disable or change page table base), the DFT time-out counter and the MISR can be enabled. The MISR enabling sequence will be described in Section IV-B4. A testing kernel can be executed after the MISR is enabled.

A testing kernel either finishes the test normally or is time-out by the DFT time-out counter, not a system timer. In both cases, the SBST body function enters the recovery stage. In the recovery stage, first, the DFT time-out counter is disabled and the MISR stops any further update. Then, the processor returns to the state before entering the SBST body function by recovering the backup register contents, processor status, MMU status, and the page table base pointer. After finishing processor state recovery, the address redirection is disabled and the interrupt is enabled.

Later, the SBST service routine can examine the test results stored in the data region and MISR. The test results in the MISR can be read by the coprocessor instruction or MMIO read while the test results in the memory can be directly fetched by load instructions. Note that the MISR can only indicate whether the test passes or not. During testing caches, the testing kernel can write the fault information into the data region. Therefore, the test results in the data region can be used to diagnose the fault location. Before the SBST service routine exits, all the pages acquired from the OS are released.

As shown in Fig. 1, the SBST body function is non-preemptive in order to obtain the expected testing sequence, and this non-preemptive environment is controlled by the DFT hardware. During executing the SBST body function, the on-bus system interrupt controller still performs its normal functions, but the interrupt signal is blocked by the DFT hardware until the test finishes.

If there are critical interrupts that must be serviced immediately, this means that the system interrupt cannot be blocked by the DFT hardware. In this case, the initialization stage has to back up the system interrupt controller settings. Then, it reconfigures the interrupt controller so that only the critical interrupts will be serviced while the rest are disabled. The prepared interrupt test ISR needs to check the interrupt source originated from the DFT hardware or the interrupt controller. If it is triggered by the interrupt controller, the current test aborts immediately and the SBST service routine enters the recovery stage. In this case, the interrupt controller settings are also restored in the recovery stage.

Note that the system interrupt is not served by the interrupt test ISR. After the processor state and the interrupt controller are recovered, the control is returned to the OS and this critical interrupt will be served by the OS. The interrupt response time is architecture dependent, and it mainly comes from the recovery stage. For the case of ARMv5 platform shown

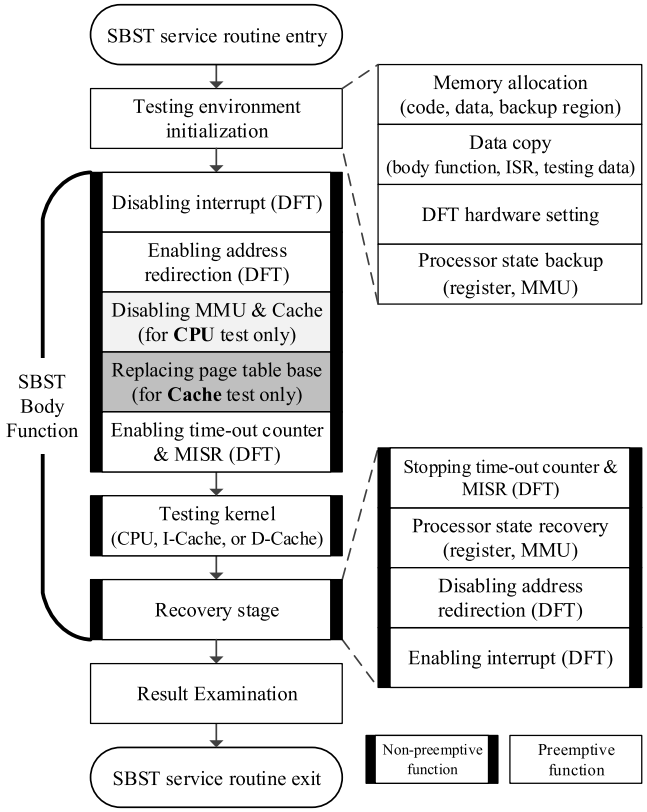

Fig. 1. SBST service routine execution flow.

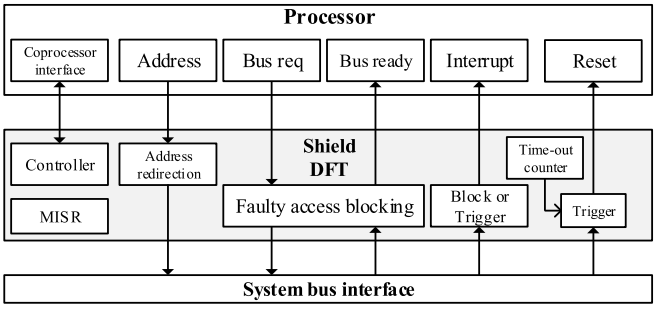

Fig. 2. Processor shield DFT hardware architecture.

in Section V, the response time is about 650 ns for 1-GHz processor.

Since the proposed methodology uses an MISR to compress the test results, each testing kernel must be atomically executed in order to obtain a valid test signature. So it is required to rerun an aborted test. One way to reduce the occurrences of incomplete tests is to divide the testing kernel into smaller modules that can be performed separately.

B. Processor Shield DFT Hardware

As shown in Fig. 2, the processor shield DFT hardware is inserted between the processor and system bus, and it works like a bus wrapper. The DFT hardware is largely dependent on the target bus architecture, not the processor architecture, so it can be easily reused to test other processors that use the same bus architecture.

If the target processor is a multi-core design, the DFT placed in the system bus may not work efficiently. In this case, inserting the DFT into the internal private core bus can provide the required protection for the entire system.

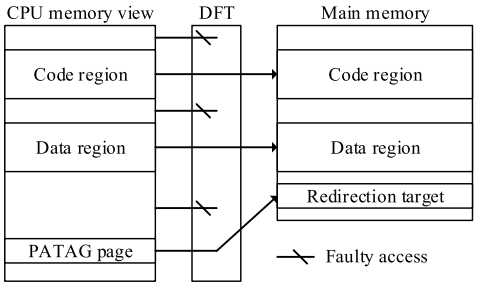

Fig. 3. Address redirection from CPU memory view to main memory.

The DFT hardware can be constructed as either an MMIO or a coprocessor. An MMIO design is a universal method. Other processors can access the DFT control through the same bus interface. A coprocessor design is a relative low-cost method, which delivers configuration arguments to DFT hardware through a coprocessor interface. A coprocessor instruction triggers the corresponding DFT function. The DFT implementation is a tradeoff between portability and overhead. In this paper, we employ a coprocessor design to implement the DFT.

The DFT hardware integrates five functions: 1) address redirection; 2) external signal control; 3) result compression; 4) time-out counter; and 5) faulty effect isolation.

1) Address Redirection for CPU Test: During CPU testing, the DFT hardware redirects all instruction fetches and data accesses to the code and data region respectively. The software framework delivers the physical base addresses, sizes of the code and data region to the DFT hardware as configuration arguments.

When a testing kernel is generated by a random instruction generator, the DFT design performs a random-to-sequential conversion [12]. The DFT hardware can generate sequential instruction addresses allocated in the code region, and then the generated addresses are used to fetch instructions rather than program counter (PC) of the processor core.

In this way, all the required addresses can be accessed by the SBST, and the third CPU SBST challenge presented in Section III-A can be resolved without influencing other processes and on-bus devices.

2) Address Redirection for Cache Test: For cache testing, the address redirection function redirects physical addresses from the processor to the main memory, as shown in Fig. 3. The SBST service routine forward two physical page numbers to the DFT hardware: the PATAG page and its redirection target page. These two page numbers can be dynamically replaced for different testing requirements. All addresses in the PATAG page are redirected to the redirection target page by the DFT hardware.

The addresses in the code and data region can be used to directly access the system bus without being modified by the DFT hardware. Other physical addresses, which are neither in PATAG pages nor in the code/data region, are regarded as faulty addresses. The DFT hardware discards the faulty addresses to prevent the system from being crashed.

When a faulty access comes to the DFT hardware, the DFT directly replies a bus access success signal to the processor without requesting the system bus. If the test cannot finish due to those discarded accesses, the DFT time-out counter resets the processor in order to break off the test.

This redirection function can effectively overcome the cache SBST challenges discussed in Section III-B1, including the memory access rights and the physical memory layout.

3) External Signal Control: The most common external signals are the interrupt and reset. An interrupt signal is typically triggered by an on-bus interrupt controller. A processor can be integrated into various platforms, which have their own interrupt controllers. Once an interrupt is triggered by the system interrupt controller, the portability of SBST programs is degraded due to the customized controls. As an alternative, utilizing the DFT hardware to trigger/block the interrupt is an efficient method to perform testing without changing configurations of the system interrupt controller. In addition, the DFT hardware can effectively perform precise interrupt testing as an interrupt signal is directly triggered by executing the DFT interrupt control instruction. In this way, the second CPU SBST challenge, i.e., precise interrupt testing, discussed in Section III-A can be resolved.

The processor reset circuit is typically regarded as an uncontrollable unit. In order to test the reset circuit, we construct the processor reset control in the DFT hardware. The DFT processor reset is not an overall system reset; all other devices can still perform their normal operations during processor reset circuit testing. Note that the system does not enter the complete reset sequence. Only the processor and caches are reset so there is no need to load the bootstrap from the flash memory or reconfigure the on-bus devices. Before performing a DFT processor reset, the testing kernel must store a unique value in the data region. The reset ISR can use this value to distinguish whether the reset signal is triggered by the testing kernel or by the DFT time-out counter, which does not store the value.

If additional external signals are required to activate the hardware circuits, the DFT design can also support testing kernels to test these circuits. For instance, the ARM processor contains two external signals, interrupt (IRQ) and fast interrupt (FIQ). The priority of FIQ is higher than IRQ. The DFT can implement three control functions, including individual IRQ trigger, FIQ trigger, and concurrent trigger. Not only interrupt-handling circuits are tested, but also priority related circuits are activated for testing. In this way, all the required external signals can be implemented and controlled by the DFT hardware.

4) Result Compression: To collect and compress test results, one way is to employ a multiple-input signature register (MISR) on the processor output side. However, there may be uncertain waiting cycles depending on bus contentions and memory latencies, and these waiting cycles interfere with the MISR result. Therefore, the MISR has to exclude all values in the waiting cycles and only compress the rest. Note that only the waiting cycles that come from the platform can be excluded, e.g., bus access and memory latency. If the processor generates waiting cycles by itself, e.g., pipeline bubbles, all the processor outputs in these cycles are compressed by the MISR. Therefore, all effects generated by performance related faults could be detected by the MISR.

During online testing, in order to obtain a signature that matches the golden one, all inputs of the MISR must be the same as running the golden test. The PC value on the address bus is one of the MISR inputs; therefore, the PC sequence should remain the same in every test iteration when the MISR is enabled. Meeting this requirement is challenging as a testing kernel is allocated in the code region that is dynamically acquired from the OS.

In order to obtain the same fixed PC sequence as running the golden test, our method is to change the PC to the same starting address of the golden test. At the same time, the address redirection function of the DFT hardware is activated so the changed PC can be redirected to the code region again. Note that the PC addresses that MISR observes are those before performing the redirection.

After the fixed PC sequence is obtained, the SBST body function resets all registers, drains the CPU pipeline with no-op, enables the MISR, and executes the testing kernel. In this way, if the processor passes the test, the observed signature will match the golden one.

5) Time-Out Counter: A time-out counter is required as operational faults may change the expected instruction flow or lead to an infinite loop. A simple cycle counter, which counts the total execution cycles, is insufficient due to the unpredictable waiting cycles from bus contentions and memory latencies. Like the MISR, the proposed time-out cycle counter excludes all these waiting cycles and counts only the rest.

When executing a testing kernel, including processor core and cache tests, the time-out function is always enabled. If time-out is triggered, the DFT hardware resets the processor core through the DFT reset control. This reset signal forces the processor core branching to the reset service routine, which is constructed by the software framework.

If the fault does not crash the system, at the end of the reset service routine, the processor will branch to the recovery stage and report a fault to the caller. Otherwise, the test may lead to a system panic and the user will need to manually reboot the whole system. In either case, the SBST has indicated a fault occurring in the processor system.

6) Fault Effect Isolation: In order to record the fault report and notify the user, the DFT hardware has to block the faulty actions to prevent the system from crashing. Faulty effects observed by the DFT hardware appear in the form of erroneous addresses, incorrect written data or corrupted bus controls.

The address redirection function can resolve the erroneous address issue. For CPU testing, all addresses, including erroneous addresses, are redirected to the code or data region. During cache testing, the DFT hardware can identify and discard the access with an erroneous address.

When there comes an access with corrupted bus control, the DFT hardware directly discards it. At the same time, the MISR compresses the value of this faulty access.

During testing, the DFT hardware cannot block the processor to write the incorrect data since there is no way to

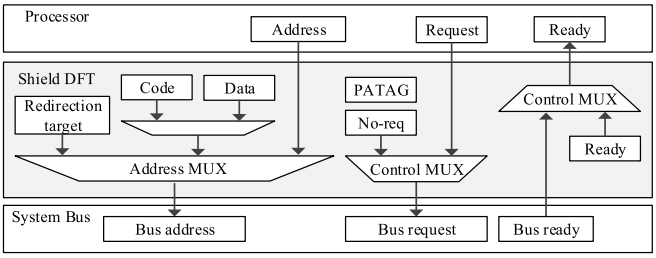

Fig. 4. Detailed function block: address redirection & faulty access blocking.

know the correct values. Nonetheless, these incorrect data do not affect other processes or on-bus devices since the DFT performs the address redirection. All the written values, including incorrect values, only can be written to the data region acquired from the OS. Therefore, we can make sure that all the other processes or on-bus devices are protected by the DFT hardware during testing. Although the DFT hardware cannot block the memory accesses, which use incorrect values, the MISR can capture these faulty values to examine the test.

If there are faulty effects triggered before enabling the DFT hardware, the DFT hardware cannot protect the system. In this scenario, the system state is unreliable before testing. When there are faults detected by the SBST service routines, how to repair the system is the next challenge. The common methods can use the checkpoint recovery or hot spare designs, but this issue is beyond the scope of this paper.

7) Detailed Function Block: The major functions of the DFT hardware are address redirection and faulty access blocking. These two functions protect the whole system without being crashed by faulty effects. As shown in Fig. 4, the detailed function block is majorly constructed by multiplexors and registers. When a memory access comes from the processor, the DFT hardware checks the current state, which may be core testing, cache testing, or normal operation. If the current state is normal, the address, request, and ready control of the processor are directly connected to the system bus. When the current state is in core testing, the page number of the output address is replaced by the page number of the code/data address configured in the DFT. During performing CPU random testing, the whole instruction fetch address is replaced by the code address generated by a random-to-sequential convertor. The request and ready signal are directly connected to the system bus.

When testing caches, the DFT compares the processor address and the PATAG register in the DFT. If the processor address is in the PATAG region, the page number is replaced by the redirection target stored in the DFT; otherwise, the processor address is directly connected to the system bus. During cache testing, the addresses, which are neither in the PATAG page nor in the code/data region, are regarded as faulty addresses, as shown in Fig. 3. All faulty accesses are blocked by the two control multiplexors, which are used to select the request or ready signals, as shown in Fig. 4. For a faulty processor address, the DFT directly replies a ready signal to the processor without requesting the system bus.

C. Comparison Between Processor Shield and Conventional SBST

In the following, we compare the proposed processor shield and conventional methods for resolving the online SBST challenges. The faulty effect isolation is the first feature of the processor shield. Conventional SBSTs typically rely on the MMU/MPU to block the faulty memory accesses, but this has three disadvantages. First, it cannot block faulty accesses that do not query the MMU. Second, it cannot work when testing the MMU itself. Third, the bus-control related faults cannot be blocked. These three disadvantages can be easily tackled down by the processor shield.

For CPU testing, the required external signals can be freely generated by the DFT, but conventional SBSTs encounter two problems. First, the reset circuits are often uncontrollable. The second is the precise interrupt test. The conventional SBST has to configure the interrupt controller and send an interrupt in a specific time. Hence, locking the bus when configuring the interrupt controller is typically required to make sure the triggering time for precise interrupt. However, locking the bus is forbidden in the advance bus, e.g., AMBA AXI4. In this case, precise interrupt is difficult to implement.

For physically tagged cache testing, the required physical memory space leads to two issues for the conventional SBSTs. First, if the memory space is dispatched to other processes, copying memory is a common method to protect these memory regions. This method results in longer execution time and requires additional memory space for backup. Second, conventional SBSTs may not be able to test the entire processor visible memory space due to the smaller size DRAM installed. If the required testing addresses are located in the reserved memory space (no DRAM mapped), the processor cannot access them. Cache faults in the reserved region are then not covered. Although these faults may have no impact on the existing system, the cache has become unreliable. The processor shield can test all the required testing addresses for caches.

V. Case Study and Experimental Results

In this section, we present a case study that preforms SBSTs under Linux kernel on an ARMv5-compatible pipelined processor, which has a 16-KB direct-mapped instruction cache and a 16-KB direct-mapped data cache. This processor has been fabricated in 90-nm technology and used to run Linux 2.6.33 to verify the design. The instruction cache is a virtual-index and virtual-tag architecture. The data cache is a virtual-index and virtual-tag architecture, which also contains a physical tag associated with each cache line. This physical tag is used to write back a dirty line without querying the MMU.

We implement a Verilog and SystemC co-simulation platform to verify the proposed processor shield design. In this platform, processor core, MMU, cache system, and DFT hardware are implemented in Verilog. The platform components, including system bus, main memory, interrupt controller, timer, and other peripherals are implemented in SystemC.

The processor design, including core, cache control logic, and processor shield DFT hardware, is synthesized

TABLE II
HARDWARE SYNTHESIS RESULTS

	CPU Core	I-cache logic	D-cache logic	DFT	Cache RAM*	Total
Area (um^2)	72,980	8,998	17,971	2,344	372,391	474,684
%	15.374	1.896	3.786	0.494	78.450	100

**The area of cache RAM cells, including the instruction and data cache, is estimated by CACTI [36].*

TABLE III
LOGIC CIRCUIT POWER ANALYSIS: PROCESSOR AND DFT (MILLIWATT)

		CPU	I-cache Logic	D-cache Logic	DFT	Total (mW)
Dynamic power	Core test	59.46 (78.31%)	3.46 (4.56%)	9.044 (11.91%)	3.968 (5.23%)	75.932 (100%)
	ICache test	41.94 (69.22%)	5.894 (9.73%)	8.862 (14.63%)	3.894 (6.42%)	60.59 (100%)
	DCache test	38.62 (66.04%)	3.446 (5.89%)	12.624 (21.59%)	3.794 (6.48%)	58.484 (100%)
	Normal program*	45.311 (70.76%)	4.931 (7.71%)	10.671 (16.66%)	3.121 (4.87%)	64.034 (100%)
Leakage power		226 (67.29%)	29.7 (8.84%)	74.2 (22.09%)	5.974 (1.78%)	335.874 (100%)

**This program is the "qsort" benchmark from MiBench.*

by TSMC 40-nm technology library, and the highest frequency is set to 1 GHz. The synthesis results are shown in Table II, and the area overhead of the processor shield is 0.494% with respect to the whole processor, including caches. The DFT circuits increase the latency between the processor bus and the system bus by 0.1 ns, which majorly comes from the multiplexors.

We use the processor gate-level netlist to execute the test programs, including core test, I-cache test, D-cache test, and normal program. During program execution, the timing related information is recorded. This record and the processor netlist are used to perform the time-based power analysis by Synopsys PrimeTime [42]. Table III shows the power analysis for the logic part of the circuits of the processor, excluding cache RAMs. The total power consumption of the DFT is about 2.4% of the total logic circuit power, including leakage and dynamic power.

For CPU testing, we develop two testing kernels, including deterministic and hybrid based on our prior works [11], [12]. To test the data cache, including RAM modules and control logics, we use the methods proposed in [31] and [41]. For the data RAM of instruction cache testing, we employ the method proposed in [32]. Furthermore, we develop the SBSTs for the control logic and tag RAM module of the instruction cache according to the method demonstrated in [31]. All the developed SBST programs are regarded as testing kernels, and we build an SBST service routine for each of the testing kernels.

Except the applied methods mentioned above, some previous works are also proposed to achieve the high fault coverage for cache RAM and controller testing. For instance, Perez *et al.* [43] demonstrated a method to construct SBST programs based on the cache accesses and read/write policies. The basic concept is that all the cache functions are performed on every cache line during testing. This is similar to the methods in [31] and [41]. All the methods can be used to design the testing kernel for cache controller testing. Our proposed processor shield focuses on how to enable testing kernels to be executed effectively on the platform hosted by an OS.

TABLE IV
FAULT COVERAGES FOR CPU FUNCTIONAL MODULES

	Fault count		Fault coverage (%)			
			Deterministic		Hybrid	
	SA.	Tran.	SA.	Tran.	SA.	Tran.
Fetch	3,951	3,854	91.32	68.11	98.79	91.13
Decoder	3,763	3,671	93.38	58.32	99.18	92.29
Register	74,907	73,050	94.67	59.31	98.09	93.22
ALU	73,358	71,452	96.02	62.61	98.14	93.56
MEM	17,521	17,187	95.12	63.61	97.89	92.67
Others	876	862	96.23	71.35	99.32	95.01
Total	174,376	170,076	95.18	61.37	98.14	93.25

SA.: stuck-at fault model; Tran.: transition fault model

In the following, we demonstrate the experimental results of the SBST programs assisted by the processor shield. The test results are shown in the subsections for CPU, cache RAM modules, and cache control logics. Finally, we profile the executions of SBST service routines under Linux.

A. Processor Core Test Results

When an SBST service routine is activated under Linux kernel, we capture all the input–output signals of the whole processor every cycle. The captured signals are translated into the test pattern format, and then the test patterns and the netlist of the processor are used to perform the fault simulation by Synopsys TetraMAX [35]. Table IV shows the fault count and fault coverage for each module of the processor core.

The deterministic program can achieve 95% stuck-at fault coverage, but only 61% of the transition fault coverage can be obtained. As a result, adding an effective random generated program, i.e., a hybrid SBST, is useful to capture the remaining stuck-at and transition faults.

The processor shield can assist such a hybrid SBST program to execute under Linux. For the hybrid SBST, not only 98% of the stuck-at fault coverage can be achieved but also 93% of the transition fault coverage is obtained. The fault coverage of the fetch unit is significantly enhanced since the processor shield enables the processor to directly access the whole logical memory space. The fault coverage of the decoder is obviously enhanced due to the massive random instructions executed.

We compare the test results to the related work in [15]. Gizopoulos *et al.* [15] proposed a systematic method to develop the SBST program for pipelined processor. They focus on the propagation of the address-related information and the hazard detection. Since their basic testing program is a loop-based code, the enhanced program unrolls the loop and adds pipeline propagation controls. The achieved stuck-at fault coverage is 95.08% while the transition fault coverage is 92.02% for miniMIPS core.

Our deterministic method performs function module tests in serial so the triggering times of each fault are less than their work. Therefore, the stuck-at fault coverages are very close, but the transition fault coverage is lower than their work.

TABLE V
RAM MODULE FAULT COVERAGES REPORTED BY RAMSES

	Data Cache			Instruction cache	
	Virtual tag	Physical tag	Data	Virtual tag	Data RAM
SAF	100%	100%	100%	100%	100%
TF	100%	100%	100%	100%	100%
AF (inter)	100%	100%	100%	100%	100%
AF (intra)	100%	100%	100%	100%	100%
CFst (inter)	100%	100%	100%	100%	100%
CFst (intra)	100%	100%	100%	100%	98.83%
CFin (inter)	100%	100%	100%	100%	100%
CFin (intra)	100%	100%	100%	100%	100%
CFid (inter)	100%	100%	100%	100%	100%
CFid (intra)	100%	100%	100%	100%	98.05%
Total	100%			99.99%	

The hybrid test can significantly improve the transition fault coverage.

B. Cache RAM Module Test Results

A complex March algorithm can detect diverse fault types due to its various March read/write sequences. It is a challenge to develop an SBST program to implement the required sequences of various March algorithms. According to the methods demonstrated in [31] and [41], we implement the March read and write respectively for different RAM modules, including tag RAM, physical tag RAM, and data RAM. In our approach, a March element is composed of the implemented March read/write to avoid the unexpected RAM accesses appeared on the target RAM module. In this way, for complex March algorithm, it will require more execution time, but not compromising the expected fault coverage.

We employ the March C- algorithm which can effectively capture stuck-at-fault (SAF), transition fault (TF), address decoder fault (AF), state coupling fault (CFst), inversion coupling fault (CFin), and idempotent coupling fault (CFid). The memory accesses are recorded during the simulation, and then they are applied to RAMSES simulator [34]. Table V shows the test results reported by RAMSES simulator.

The proposed online SBST executed in the Linux-governed processor-cache system can obtain 100% of the fault coverage for data cache testing. The fault coverage of the instruction cache test does not achieve 100% in intra-word faults of CFst and CFid. The reason is that the chosen instructions, which are regarded as the March DBs, are limited by the target ISA without generating ambiguous test results [32]. The final fault coverage of the instruction cache is over 99.99%.

C. Cache Control Logic Test Results

The design complexity of the cache control logic is dependent on cache control functions. The data cache controller provides eight control functions: enable/disable, invalidate, clear, preload, write-back, write-through, write-allocate, and write-around policy. The instruction cache controller only provides three control functions: enable/disable, invalidate, and preload. Table VI shows the test results for each module of the cache control logics. As testing the processor core, we also feed the netlist of cache control logics and the stimuli to Synopsys TetraMAX [35] to obtain these fault coverages, including the stuck-at and transition fault model. Since the cache control logics are more regular and controllable, the fault coverages of cache control logics are higher than the CPU.

TABLE VI
TEST RESULTS FOR CACHE CONTROL LOGICS

Instruction Cache				
	Stuck-at fault		Transition fault	
Module	Fault count	F.C.	Fault count	F.C.
Controller	480	96.88%	456	94.08%
Multiplexer	677	99.41%	677	98.67%
Valid unit	18,114	99.34%	16,046	96.93%
Others	1,642	98.66%	1,610	95.34%
Total	20,913	99.23%	18,989	95.77%
Data Cache				
	Stuck-at fault		Transition fault	
Module	Fault count	F.C.	Fault count	F.C.
Controller	1,462	93.02%	1,438	90.13%
Multiplexer	1,132	99.12%	1,132	97.26%
Valid unit	18,114	99.35%	16,046	94.33%
Dirty unit	35,800	99.31%	31,692	95.73%
Others	1,738	96.95%	1,702	94.30%
Total	58,246	99.09%	52,010	95.13%

F.C.: fault coverage

TABLE VII
SBST TESTING KERNEL MEMORY USAGES

Testing kernel	Static code size	Dynamic memory usage
Processor Core		code / data / backup
Deterministic	3.82 KB	4KB / 8KB / 4KB
Hybrid	3343.17 KB	4MB / 4MB / 4KB
Instruction Cache		code / data / backup / March
Data RAM	11.32 KB	16KB / 4KB / 4KB / 32KB
Tag RAM	3.41 KB	4KB / 4KB / 4KB / 32KB
Logic device	20.03 KB	24KB / 4KB / 4KB / 32KB
Data Cache		code / data / backup / March
Data RAM	1.69 KB	4KB / 4KB / 4KB / 32KB
Virtual Tag RAM	3.89 KB	4KB / 4KB / 4KB / 32KB
Physical Tag RAM	6.42 KB	8KB / 4KB / 4KB / 32KB
Logic device	24.11 KB	28KB / 4KB / 4KB / 32KB

D. SBST Service Routine Statistics

Table VII shows the static code size and dynamic memory usage of testing kernels respectively. During executing a testing kernel, the dynamic memory usage is the required memory pages, including code region, data region, backup region, and March region. The March region is used to perform the March algorithm for cache RAM testing. During testing physical tag RAM, the March region contains the PATAG pages. For ARM architecture, the page size is typically 4 KB in Linux, and get_free_pages system call can provide two to the power of n pages. This system call can obtain a continuous 4-MB physical memory space.

During executing the SBST service routine, the data region stores the shared data, which are used in three stages, including initialization, SBST body function, and result examination. When testing CPU, only the required input patterns are stored in the data region, and the size is about 900 B. When testing caches, the shared data include program flow control variables, March data backgrounds, testing results, and values for logic device testing. The size of the shared data ranges from 0.8 to 2.1 KB.

TABLE VIII
SBST SERVICE ROUTINE EXECUTION TIMES (NANOSECOND)

Service Routine	Initial-ization	Body function	Result	Total	Overall
Processor Core					
Deter-ministic	13,432 (35.10%)	17,447 (45.59%)	7,392 (19.31%)	38,271 (100%)	7,826,794
Hybrid	2,112,838 (26.99%)	5,547,637 (70.89%)	166,319 (2.12%)	7,826,794 (100%)	
Instruction cache					
Data RAM	20,314 (1.17%)	1,691,714 (97.77%)	18,376 (1.06%)	1,730,404 (100%)	2,427,049
Tag RAM	16,712 (8.40%)	164,471 (82.67%)	17,773 (8.93%)	198,956 (100%)	
Logic device	24,964 (5.02%)	453,394 (91.1%)	19,331 (3.88%)	497,689 (100%)	
Data cache					
Data RAM	14,597 (3.48%)	386,432 (92.19%)	18,122 (4.33%)	419,151 (100%)	1,628,547
Virtual Tag RAM	16,323 (7.89%)	172,543 (83.45%)	17,897 (8.66%)	206,763 (100%)	
Physical Tag RAM	17,895 (7.66%)	197,331 (84.44%)	18,471 (7.9%)	233,697 (100%)	
Logic device	28,401 (3.69%)	721,523 (93.84%)	19,012 (2.47%)	768,936 (100%)	

Table VIII shows the execution time breakdown for each of the SBST service routines. The SBST execution time is measured after the OS has performed the context switching and gives the control to the SBST process until the end of the SBST process. Since the entire SBST process is preemptive except the body function, only the time executing the SBST program is counted as the SBST execution time. In Table VIII, one row represents an SBST service routine, and each of the SBST service routines can be executed independently.

The memory allocation and data copy dominate the execution time of the initialization stage. The get_free_pages system call is the major function of memory allocation, and it requires 1500–80 000 ns depending on the requested numbers of pages. The data copy execution time is dependent on the static code size and the required testing data size. For the hybrid method, the initialization stage needs over 2 ms. Since the code size of the hybrid testing kernel is larger than 3 MB, the SBST service routine has to acquire the maximum number of free pages, i.e., 4 MB, and copy the program into the code region.

Since the ARMv5 processor uses virtual-tag virtual-index cache architecture, the cache contents are flushed by the Linux during context switching. This context switching overhead depends on the volume of the dirty cache data; for instance, to flush 16 KB dirty data in our experimental system would require about 10 000 ns. This overhead time is not included in the execution time shown in Table VIII.

The processor state backup is another architecture dependent task. For the target ARMv5 architecture, the processor state backup contains 43 32-bit registers, including the register file, processor status register, and the MMU controls. This backup operation requires 432 ns, which is already counted in the initialization execution time shown in Table VIII.

Since the SBST body function is non-preemptive, the execution time of this part represents the longest uninterruptable period. The hybrid method for core testing requires 5.6 ms to execute the testing kernel without being interrupted. Note that a commonly used context-switch interval is 4 ms in Linux. This means non-preemptive SBST testing kernels can be scheduled as general processes, and there is almost no impact to the system operations.

TABLE IX
CPU STUCK-AT FAULT COVERAGES WITH ENHANCEMENT METHODS

	Original (hybrid program)		UFF random program generation	Flip-flop replacement	
	Fault count	F.C. (%)	F.C. (%)	Replaced FF count	F.C. (%)
Fetch	3,951	98.79	98.80	34	99.67
Dec.	3,763	99.18	99.19	5	99.31
Reg.	74,907	98.09	98.65	302	99.05
ALU	73,358	98.14	98.62	166	98.85
MEM	17,521	97.89	98.32	184	99.36
Others	876	99.32	99.32	2	99.54
Total	174,376	98.14	98.62	693	99.01

F.C.: fault coverage

After the testing kernel, the SBST body function enters the recovery stage. This stage is also an architecture dependent procedure. For the target ARMv5, the 43 32-bit values in the backup region are restored into the corresponding registers. This recovery operation requires 446 ns, which is included in the body function execution time shown in Table VIII.

The result examination stage includes MISR signature check and releases of the acquired memories. The release system call, i.e., free_pages Linux system call, requires 1000–100 000 ns depending on the released number of pages. This system call dominates the execution time of the result examination stage. For cache testing, the execution times of the result examination are almost the same since their memory usages are very close.

The rightmost column in Table VIII shows the overall execution times to test CPU core, I-Cache, and D-Cache, respectively. When testing data RAM of the instruction cache, one March read value, i.e., one instruction, typically needs more than five instructions to examine the test result. This is the reason why the required time of I-Cache testing is longer than the D-Cache testing.

VI. DISCUSSION

In this section, we discuss fault coverage enhancement and related issues: interrupt controller online testing, and the cooperation between the SBST and DVFS.

A. CPU Fault Coverage Enhancement

We use two methods to improve the CPU stuck-at fault coverage: uncovered fault first (UFF) random program generation and flip-flop replacement.

In our previous work [12], the developed random program generator is a basic block-based generator. The generated random program is composed of three types of basic blocks: in-order, fully random, and mode change. This generator builds a random program that includes about 800-k instructions. The random program along with the deterministic program in [11] has achieved the fault coverage of 98.14% for stuck-at fault. This is the original method shown in Table IX.

The idea of the UFF random program generation is to generate effective random programs to test the uncovered faults iteratively. We explore the following procedure to improve

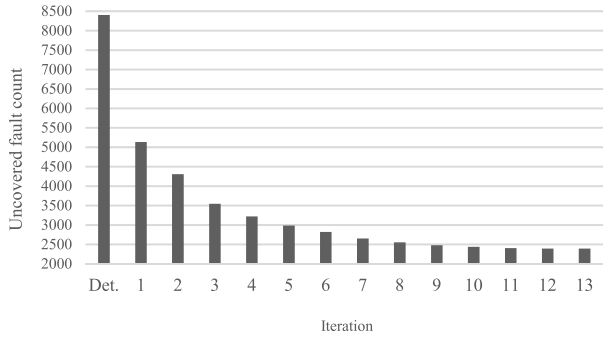

Fig. 5. Uncovered fault count for each best UFF random program.

the fault coverage. First, the faults tested by the deterministic program are removed from the whole fault set. The remained is called the uncovered fault set. The random program generator first produces about 200 smaller test programs, each containing about 100-k instructions for fault simulation. The test program having the best fault coverage for the uncovered fault set is chosen as a testing kernel. Then, the tested faults are removed from the original uncovered fault set and we repeat the above procedure several times to get more testing kernels.

As shown in Fig. 5, the first UFF random program can test more than 3000 uncovered faults, which are not tested by the deterministic SBST program. The 11th UFF random program can test 33 more faults. However, the following UFF random programs cover less than 10 faults, a more dramatic change occurring here. Therefore, we can choose the first 11 UFF random programs for SBST testing, totally including 1.1 million instructions. All the tested faults are the union of the covered faults of all UFF random programs and the deterministic program. The achieved fault coverage has increased to 98.62% for stuck-at fault, as shown in Table IX.

Note that these testing kernels can be executed independently with their own golden MISR signatures. Once all the testing kernels are performed, the expected fault coverage can be obtained without executing them in order. For each testing kernel, the non-preemptive execution time is less than 1 ms.

The second method is the flip-flop replacement. From the uncovered fault set, we discover that many faults on the reset pin of flip-flops are untested. We further observe that almost all these flip-flops are located in the data paths. During testing, when a CPU active-low reset is triggered, the flip-flops on control paths or data paths are reset at the same time. If there is a reset stuck-at-one fault in the data path flip-flop, this flip-flop will still keep the original value without being reset. This reset fault is not observable since the flip-flop data cannot be propagated to the next stage. When a fetched instruction comes, the new value overwrites the original one. Therefore, the reset signals of the flip-flops in the data paths are untestable and become redundant faults.

We replace all the flip-flops in the data paths with ones without the reset input. Note that the flip-flops in the control paths still keep their reset pins. The post-synthesis simulation can verify the correctness of the replacement. The replaced flip-flop count and fault coverage is shown in the rightmost two columns in Table IX. Based on the above two improvement methods, we have achieved 99.01% fault coverage of stuck-at faults for the target ARMv5 processor.

B. Interrupt Controller Online Testing

The functions of the interrupt controller directly change the processor execution flow. In order to test the interrupt controller, the processor has to enter the testing mode to execute test programs. In general, the interrupt controller circuits can be divided into three parts: external inputs, priority comparison, and processor notification. The test controllability is directly dependent on the external input signals. However, to control these input signals is challenging.

If the interrupt controller is embedded into the processor, the proposed processor shield can directly control the input signals and generate the required scenarios for testing. In our simulation platform, the interrupt controller is constructed as a behavioral standalone peripheral; the SBST testing of the interrupt controller itself is not investigated in this paper.

C. SBST and DVFS: Required Guardband Calibration

The proposed processor shield methodology not only constructs the required testing environment but also isolates all the faulty effects to prevent the system from crashing. Hence, each of the SBST testing kernels can be repeated even if the test fails. This inspires us to integrate the processor shield with the dynamic voltage and frequency scaling (DVFS) technique to calibrate the required guardbands, which is used to accommodate transistor aging.

Transistor aging, e.g., NBTI aging [37], typically causes increase of the threshold voltage for operation and degrades the highest working frequency, which may result in 20% speed degradation [38]. To accommodate aging effects, a conventional solution is to use guardband, which reduces the working frequency or increases the operating voltage. Designers usually incorporate one-time worst case guardband (OWG) at the beginning of chip lifetime. This guardband is assumed for the worst case aging effects at the end of chip lifetime. In order to obtain the expected performance, OWG typically increases the operating voltage. However, this increases the power consumption and accelerates the aging effect at the same time.

The guardbands can be reduced in the beginning of chip lifetime, and the minimal required guardbands can be dynamically recomputed without using OWG. In this paper, we propose using the online self-test which can provide feedback to the DVFS system to calibrate the minimal required guardbands.

In the proposed method, the SBST service routine cooperates with the DVFS system to regulate a voltage and frequency pair based on a previous workable setting. For instance, the Linux system provides an operating performance point (OPP) library, which records the workable voltage and frequency pairs for each device. The DVFS system can query the OPP library to set the operating voltage and frequency.

As shown in Fig. 6, the DVFS system sets the frequency to a specified value and programs the voltage to the expected lowest value from the OPP record. If an SBST fails, this SBST is iterated with a higher voltage. When the processor passes the test, the current voltage is the new workable lowest.

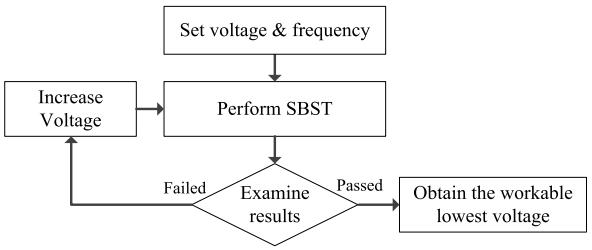

Fig. 6. Flow to get the workable lowest voltage for a specified frequency.

This voltage and frequency pair is used to update the record in the OPP library. The workable lowest voltage for any frequency can be acquired by executing the above loop. In this way, the guardbands can be calibrated for processor core, instruction cache, and data cache, respectively.

In order to perform SBSTs for guardband calibration, the SBST programs may be executed several times under different operating conditions, including voltage and frequency. Therefore, all faulty actions must be blocked without influencing system operations. The proposed processor shield design can effectively construct the testing environment where the SBST programs can be executed many times even if the SBST fails.

A similar concept has been proposed in [39], but the authors used the inserted hardware devices to collect real-time information for guardband calibration. This needs to modify the internal architecture of the processor. Our proposed method can collect an accurate amount of the accumulated aging effects without changing the processor design through the method shown in Fig. 6.

VII. Conclusion

This paper has addressed system related issues in the processor online testing, including redirecting shielded address, controlling interrupt signal, isolating faulty effect. We propose the processor shield, which efficiently creates the required testing environment to tackle the system related issues. The processor shield design is an architectural methodology, including the software framework and the DFT hardware.

The software framework requests the required resources and privilege right from the OS. The DFT redirection function enables the test program to access any required physical address without violating memory protection scheme, including the shielded one. The DFT can trigger the interrupt signal at the deterministic time with respect to the instruction sequence of the test program for precise interrupt testing. The DFT hardware also can isolate all faulty actions. The software framework recovers the processor state even if the SBST fails. A fault report can be obtained after testing, and the user will be notified, instead of crashing the system. The SBST service routines can completely run under an OS without affecting other processes and on-bus devices. Our method can seamlessly switch between the SBST service routine and the kernel process.

We present a case study that demonstrates the SBST service routine executions under Linux on an ARMv5-compatible processor system. For CPU testing, the stuck-at fault coverage is over 99% while the transition fault coverage is higher than 93%. For cache control logic testing, the stuck-at fault coverage is over 99% while the transition fault coverage is higher than 95%. For RAM module testing, the fault coverage is nearly 100%. For online testing, these results show that the expected high fault-coverage can be achieved through the processor shield design. The hardware overhead is 0.494% compared to the whole processor area. In addition, we have discussed an SBST-DVFS application that calibrates the dynamic minimal guardbands to reduce operating power consumption and mitigate transistor-aging effect.

References

[1] A. Paschalis and D. Gizopoulos, "Effective software-based self-test strategies for on-line periodic testing of embedded processors," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 24, no. 1, pp. 88–99, Jan. 2005.

[2] F. Corno, E. Sánchez, M. S. Reorda, and G. Squillero, "Automatic test program generation: A case study," *IEEE Des. Test Comput.*, vol. 21, no. 2, pp. 102–109, Mar. 2004.

[3] M. Scholzel, T. Koal, and H. T. Vierhaus, "Systematic generation of diagnostic software-based self-test routines for processor components," in *Proc. IEEE Eur. Test Symp. (ETS)*, May 2014, pp. 1–6.

[4] P. Singh, D. L. Landis, and V. Narayanan, "Test generation for precise interrupts on out-of-order microprocessors," in *Proc. IEEE Int. Workshop Microprocessor Test Verification (MTV)*, Dec. 2009, pp. 79–82.

[5] E. Sánchez and M. S. Reorda, "On the functional test of branch prediction units," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 23, no. 9, pp. 1675–1688, Sep. 2015.

[6] D. Sabena, M. S. Reorda, and L. Sterpone, "A new SBST algorithm for testing the register file of VLIW processors," in *Proc. Des. Autom. Test Eur. Conf. Exhibit. (DATE)*, Mar. 2012, pp. 412–417.

[7] P. Bernardi *et al.*, "On-line software-based self-test of the address calculation unit in RISC processors," in *Proc. IEEE Eur. Test Symp. (ETS)*, May 2012, pp. 1–6.

[8] P. Bernardi *et al.*, "On the functional test of the register forwarding and pipeline interlocking unit in pipelined processors," in *Proc. IEEE Int. Workshop Microprocessor Test Verification (MTV)*, Dec. 2013, pp. 52–57.

[9] P. Bernardi *et al.*, "On the in-field functional testing of decode units in pipelined RISC processors," in *Proc. IEEE Int. Symp. Defect Fault Tolerance Nanotechnol. Syst. (DFT)*, Oct. 2014, pp. 299–304.

[10] N. Kranitis, A. Paschalis, D. Gizopoulos, and G. Xenoulis, "Software-based self-testing of embedded processors," *IEEE Trans. Comput.*, vol. 54, no. 4, pp. 461–475, Apr. 2005.

[11] C. H. Chen, C. K. Wei, T. H. Lu, and H. W. Gao, "Software-based self-testing with multiple-level abstractions for soft processor cores," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 5, pp. 505–517, May 2007.

[12] T.-H. Lu, C.-H. Chen, and K.-J. Lee, "Effective hybrid test program development for software-based self-testing of pipeline processor cores," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 19, no. 3, pp. 516–520, Mar. 2011.

[13] N. Kranitis, A. Merentitis, G. Theodorou, A. Paschalis, and D. Gizopoulos, "Hybrid-SBST methodology for efficient testing of processor cores," *IEEE Des. Test Comput.*, vol. 25, no. 1, pp. 64–75, Jan./Feb. 2008.

[14] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Effective software self-test methodology for processor cores," in *Proc. Des. Autom. Test Eur. (DATE)*, Mar. 2002, pp. 592–597.

[15] D. Gizopoulos *et al.*, "Systematic software-based self-test for pipelined processors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 16, no. 11, pp. 1441–1453, Nov. 2008.

[16] L. Chen, S. Ravi, A. Raghunathan, and S. Dey, "A scalable software-based self-test methodology for programmable processors," in *Proc. IEEE/ACM Des. Autom. Conf. (DAC)*, Jun. 2003, pp. 548–553.

[17] L. Lingappan and N. K. Jha, "Satisfiability-based automatic test program generation and design for testability for microprocessors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 15, no. 5, pp. 518–530, May 2007.

[18] V. Singh, M. Inoue, K. K. Saluja, and H. Fujiwara, "Instruction-based self-testing of delay faults in pipelined processors," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 14, no. 11, pp. 1203–1215, Nov. 2006.
[19] A. Riefert, R. Cantoro, M. Sauer, M. S. Reorda, and B. Becker, "A flexible framework for the automatic generation of SBST programs," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 24, no. 10, pp. 3055–3066, Oct. 2016.
[20] K. Batcher and C. Papachristou, "Instruction randomization self test for processor cores," in *Proc. IEEE VLSI Test Symp. (VTS)*, Apr. 1999, pp. 34–40.
[21] S. M. Al-Harbi, F. Noor, and F. M. Al-Turjman, "March DSS: A new diagnostic march test for all memory simple static faults," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 26, no. 9, pp. 1713–1720, Sep. 2007.
[22] S. Hamdioui, A. J. van de Goor, and M. Rodgers, "March SS: A test for all static simple RAM faults," in *Proc. IEEE Int. Workshop Memory Technol., Des. Test (MTDT)*, Jul. 2002, pp. 95–100.
[23] A. J. van de Goor, I. B. S. Tlili, and S. Hamdioui, "Converting March tests for bit-oriented memories into tests for word-oriented memories," in *Proc. Int. Workshop Memory Technol., Des., Test. (MTDT)*, Aug. 1998, pp. 46–52.
[24] P. Bernardi, R. Cantoro, S. De Luca, E. Sánchez, and A. Sansonetti, "Development flow for on-line core self-test of automotive microcontrollers," *IEEE Trans. Comput.*, vol. 65, no. 3, pp. 744–754, Mar. 2016.
[25] M. A. Skitsas, A. Nicopoulos, and M. K. Michael, "DaemonGuard: Enabling O/S-orchestrated fine-grained software-based selective-testing in multi-/many-core microprocessors," *IEEE Trans. Comput.*, vol. 65, no. 5, pp. 1453–1466, May 2016.
[26] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "A flexible software-based framework for online detection of hardware defects," *IEEE Trans. Comput.*, vol. 58, no. 8, pp. 1063–1079, Aug. 2009.
[27] P. Bernardi, L. M. Ciganda, E. Sánchez, and M. S. Reorda, "MIHST: A hardware technique for embedded microprocessor functional on-line self-test," *IEEE Trans. Comput.*, vol. 63, no. 11, pp. 2760–2771, Nov. 2014.
[28] G. Theodorou, N. Kranitis, A. Paschalis, and D. Gizopoulos, "Software-based self-test for small caches in microprocessors," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 33, no. 12, pp. 1991–2004, Dec. 2014.
[29] J. Sosnowski, "In system testing of cache memories," in *Proc. Int. Test Conf. (ITC)*, Oct. 1995, pp. 384–393.
[30] S. M. Al-Harbi and S. K. Gupta, "A methodology for transforming memory tests for in-system testing of direct-mapped cache tags," in *Proc. IEEE VLSI Test Symp. (VTS)*, Apr. 1998, pp. 394–400.
[31] Y.-C. Lin, Y.-Y. Tsai, K.-J. Lee, C.-W. Yen, and C.-H. Chen, "A software based test methodology for direct mapped data cache," in *Proc. Asian Test Symp. (ATS)*, Nov. 2008, pp. 363–368.
[32] C.-W. Lin and C.-H. Chen, "Unambiguous I-cache testing using software-based self-testing methodology," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, Jun. 2014, pp. 1756–1759.
[33] D. K. Pradhan and S. K. Gupta, "A new framework for designing and analyzing BIST techniques and zero aliasing compression," *IEEE Trans. Comput.*, vol. 40, no. 6, pp. 743–763, Jun. 1991.
[34] C.-F. Wu, C.-T. Huang, and C.-W. Wu, "RAMSES: A fast memory fault simulator," in *Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Syst. (DFT)*, Nov. 1999, pp. 165–173.
[35] (Feb. 2017). *TetraMAX Version L-2016.03*. Synopsys Inc., San Jose, CA, USA. [Online]. Available: http://www.synopsys.com/
[36] (Feb. 2017). *Cacti Version 5.3*. HP Inc., Palo Alto, CA, USA. [Online] Available: http://www.hpl.hp.com/research/cacti/
[37] G. Chen *et al.*, "Dynamic NBTI of PMOS transistors and its impact on device lifetime," in *Proc. IEEE Int. Reliab. Phys. Symp. (IRPS)*, Mar. 2003, pp. 196–202.
[38] J. Abella, X. Vera, and A. Gonzalez, "Penelope: The NBTI-aware processor," in *Proc. IEEE Int. Symp. Microarchitecture (MICRO)*, Dec. 2007, pp. 85–96.
[39] E. Mintarno *et al.*, "Self-tuning for maximized lifetime energy-efficiency in the presence of circuit aging," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 30, no. 5, pp. 760–773, May 2011.
[40] C.-W. Lin and C.-H. Chen, "A processor shield for software-based on-line self-test," in *Proc. IEEE Asia–Pacific Conf. Circuits Syst. (APCCAS)*, Oct. 2016, pp. 149–152.
[41] C.-W. Lin and C.-H. Chen, "Processor shield for L1 data cache software-based on-line self-testing," in *Proc. Asia South Pacific Des. Autom. Conf. (ASPDAC)*, Jan. 2017, pp. 420–425.
[42] (Feb. 2017). *PrimeTime Version M-2016.12*. Synopsys Inc., San Jose, CA, USA. [Online]. Available: http://www.synopsys.com/
[43] W. J. Perez, J. Velasco, D. Ravotto, E. Sánchez, and M. S. Reorda, "A hybrid approach to the test of cache memory controllers embedded in SoCs," in *Proc. IEEE Int. On-Line Test. Symp. (IOLTS)*, Jul. 2008, pp. 143–148.

Ching-Wen Lin received the B.S. and M.S. degrees in electrical engineering from National Cheng Kung University, Taiwan, in 2006 and 2008, respectively. He is currently working toward the PhD degree from National Cheng Kung University. His research interests include computer architecture, online testing, and software-based self-testing.

Chung-Ho Chen received the MSEE degree in electrical engineering from the University of Missouri-Rolla, Rolla, in 1989 and the PhD degree in electrical engineering from the University of Washington, Seattle, in 1993. He then joined the Department of Electronic Engineering, National Yunlin University of Science and Technology. In 1999, he joined the Department of Electrical Engineering, National Cheng-Kung University, Tainan, Taiwan, where he is currently a professor. His research areas include advanced computer architecture, graphics processing, and full system ESL simulation systems. He is a member of the IEEE.