A Processor Shield for Software-Based On-Line Self-Test

Ching-Wen Lin and Chung-Ho Chen
Institute of Computer and Communication Engineering, National Cheng Kung University, Tainan, Taiwan, R.O.C.
kliolin@mail.ee.ncku.edu.tw and chchen@mail.ncku.edu.tw

***Abstract*—Software-based processor self-test typically ignores system related testing issues such as interrupt, memory-mapped IOs, especially for on-line testing. We propose an architectural support for processor SBST testing: Processor Shield, which can tackle the difficult-to-test issues during on-line SBST. We develop an execution flow to control the processor shield and run the SBST program without interfering other processes and on-bus devices. Finally, we present a case study that executes the SBST program under Linux kernel on an ARMv5-compatible processor system. Our method can successfully switch the test process and the kernel process and achieve the expected high processor fault coverage. The hardware overhead of the processor shield is 2.6% compared to the logic part of the processor.**

Keywords—SBST; on-line test; fault coverage; processor testing

I. Introduction

When an embedded processor comes up to operations, the operational faults may occur and degrade the system reliability. Although the manufacturing test can separate the good dies from the defective ones, the operational faults may appear and violate the normal functions during the system lifetime. On-line testing is an effective way to detect operational faults through non-concurrent or concurrent test methods [1]. The concurrent test can be classified into four strategies. *Hardware redundancy strategies* use hardware duplication and comparison to detect the faults. *Information redundancy strategies* employ coding schemes, such as Error-Correcting Code (ECC), to detect and correct the faults. *Time redundancy strategies* typically use a re-computing scheme to compare the results of both computations. *Software redundancy strategies* use N-version source codes and the software signature monitor for error detections.

Those concurrent on-line test methods introduce significant overheads in terms of hardware resources and system performance loss. Therefore, non-concurrent on-line tests become attractive ways for low-cost and non-safety-critical embedded applications. The most common non-concurrent on-line testing method is the functional test, which relies on the processor to execute the program to test the operational faults. In a conventional functional test, the processor can execute a test program, which in turn collects the fault syndromes for specified devices. However, there is an ambiguity here: is the detected fault coming from the specified design under test (DUT) or from the processor? Therefore, there exists a profound need that performing a comprehensive on-line test for the processor itself before testing other devices is necessary.

Despite the success of traditional manufacturing test methods, including scan-based test and Built-In Self-Test (BIST), it is difficult to perform on-line self-testing through them without an ATE machine. Software-Based Self-Test (SBST) is a feasible way to implement the on-line self-test for a processor. SBST for a processor can perform at-speed testing and critical path testing with no design modification and no additional power consumption.

For an embedded processor on-line SBST, the system memory mapping is a restriction for accessing various addresses that may be mapped onto either memory-mapped I/O (MMIO) devices or the reserved memory space. The read/write operation from the processor often affects the function of an MMIO, and also accessing a reserved memory space typically incurs an exception that changes the program control flow. To test only the processor and avoid test result ambiguity, we need to isolate the system devices in the platform, i.e., shield the processor from the MMIOs when performing processor SBST testing. To obtain high fault coverage, the processor SBST program cannot place any limit on the address to test. We call the access, whose address is the reserved space, MMIO, or memory space of other processes, *shielded SBST access*. An on-line SBST has to redirect all *shielded SBST accesses*, which come from the test program or possibly faulty effects, to available memory space; otherwise, the test may result in a system crash.

An interrupt triggers the crucial CPU functions, for instance, invoking shadow register use, switching to a privilege mode, and preserving the interrupted status. A preemptive operating system (OS) may suspend a SBST process to serve a higher priority interrupt. However, to guarantee obtaining the expected test result, the test procedure must be a non-preemptive process during on-line SBST. Therefore, all unexpected external stimuli have to be isolated from the processor so the expected SBST test results can be obtained. In addition, the interrupt triggering time has to be deterministic with respect to the instruction sequence of the test program.

A high fault-coverage processor SBST requires to access various addresses in order to deliver fault syndromes. In a paged virtual memory system, the operating system has to generate vast number of pages for the SBST process, but this may be impossible or illegal in the operating system. Therefore, during testing a processor core, disabling the MMU is a desirable test strategy. While the MMU is disabled during on-line SBST, the processor directly uses a virtual address to access the memory. This makes the following program flow unpredictable. Even if the test can be completed, how to return to the operating system becomes another challenge.

To tackle the system related testing issues, we propose an architectural support for the processor on-line SBST: **processor shield,** which can provide a shielded SBST environment for the

This work is supported by the Ministry of Science and Technology, Taiwan, R.O.C., under Grant 104-2218-E-006-028- and 105-3011-E-006-001-

processor to perform on-line SBST without interfering the other system devices and processes. The processor shield can isolate the external stimuli and faulty effects during an on-line test. The processor shield can also keep the expected program flow of the test code while the MMU is disabled. In our design, the SBST process can return the control to the OS as if the test program were never executed. We present a case study that executes the SBST program under Linux kernel on an ARMv5-compatible processor system. Our method can successfully switch the test process and the kernel process and achieve the expected high processor fault coverage, depending on the SBST program used. The hardware overhead of the processor shield is about 2.6% compared to the logic part of the processor.

The rest of this paper is organized as follows. Section II describes the background knowledge and related work. Section III presents our proposed processor shield design. Section IV provides the case study and the experimental results. Finally, we make a conclusion for this paper.

II. Background and Related Work

Paschalis and Gizopoulos have proposed a periodic on-line testing for embedded processors [1]. They classify target components according to the relationship of the data path. During testing data-path-bound components, some faults of the control-path-bound components can be covered. For testing MIPS, they get a small test code size (808 instructions) and an acceptable fault coverage (95%) using their test policy. A small test program means that the test can be executed more times during a time interval so this can increase the probability to capture the intermittent faults. However, system related issues are ignored in their work, including *shielded SBST access* redirection, the interrupt test, the MMU related issues, and a non-preemptive test.

Bernardi et al. have proposed a hardware module, MIcroprocessor Hardware Self-Test unit (MIHST), for performing on-line SBST [2]. They use a loop unrolling method to unroll loops in their test program, and they encode the unrolled loops to reduce the code size. The MIHST hardware decodes them for the processor during the execution time. Besides the decoding, MIHST also takes over the bus control during the on-line test so they can redirect all memory accesses to MIHST, including *shielded SBST accesses*. They use a wishbone bus as an example to show how to take over the bus control. However, they ignore the complexity of a modern bus system. For example, in the AMBA bus, they have to change the arbiter and the decoder to take over the bus control. Besides the bus issue, they also ignore the interrupt test, the MMU related issues, and a non-preemptive test. In their work, they get a relative low fault coverage: 92.67% for miniMIPS.

An effective method for building a high coverage processor SBST program is to combine a deterministic program and a random generated code [3]. During a random test, a test shell is proposed to generate the sequential instruction/data addresses to access the system bus, and the addresses generated by the processor are ignored. In the standalone platform, the test shell can generate any address of the memory; however, the address generation may lead to a system crash due to the modified system context during an on-line test. Our work in this paper proposes an SBST execution strategy that can switch the test process and the kernel process seamlessly for on-line processor testing.

III. Proposed Processor Shield Design

We propose an architectural support for processor SBST: processor shield, that deals with the hard-to-test issues in the on-line system. The processor shield isolates the external stimuli and the processor faulty effects during the on-line test phase. The cache testing and MMU testing are beyond the scope of this paper so our procedure disables the cache and MMU during testing. As shown in Fig. 1, the processor shield is inserted between the processor core and the system bus, and it works like a bus wrapper.

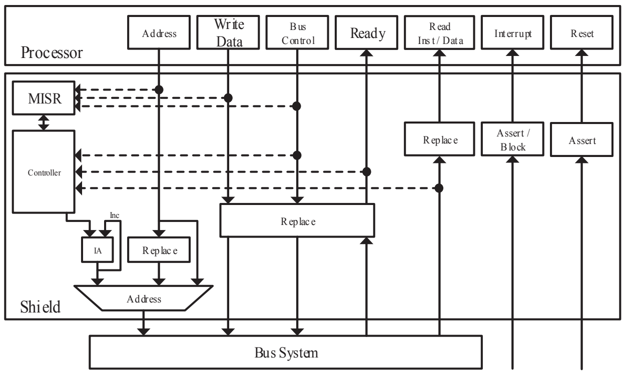


Fig. 1 Processor Shield Architecture Diagram

The processor shield provides four major functions: (1) address redirection function, (2) external signal control, (3) result compression, and (4) time-out and return function. All these functions can be turned on or off independently. For system security, the processor shield can only be accessed when the processor is in the privilege mode.

The processor shield can be implemented as either an MMIO or a coprocessor. An MMIO design requires a bus interface and system memory mapping. To implement the processor shield like a coprocessor is a relative low-cost method. A specified coprocessor instruction triggers the corresponding function.

A. Address Redirection Function

The SBST process requires three private memory regions for the test program, test data, and backup, respectively. Our SBST process stores its context; including all register contents, the processor status, and the MMU status, to the backup region before the test program is actually executed. Therefore, as the SBST process finishes its execution or a program time-out occurs, the processor recovers its context, which is stored before entering the test program, and returns the control to the OS.

In Fig. 2, we show the pseudo code that sends the physical memory attributes to the processor shield, including the base physical address and the size of the test program region acquired from the operating system. First, the SBST process calls the system call (get_free_pages) to get continuous physical pages for one of the memory regions, and a virtual address pointer is returned. Second, the SBST process calls the system call (virt_to_phys) to get the corresponding physical address. Then, the two inline assembly codes send the physical address and the size to the processor shield through the coprocessor interface.

```
va=get_free_pages(num); // acquire continuous pages from OS
pa=virt_to_phys(va); // get base physical address, pa,
asm (“Send_Program_Region_addr(pa)”); // send pa to shield
asm (“Send_Program_Region(num)”); // send no. of pages to shield
```

Fig. 2 Pseudo code: sending phsical memory attributes to the processor shield

After the processor shield records the physical memory attributes, the SBST process jumps to the test program region and enables the redirection. Then, the SBST process disable the MMU, setup the time-out counter, drain the CPU pipeline with no-ops, and lastly enable the MISR for test signature recording.

When executing the SBST body, all instruction fetches and data accesses are redirected to the corresponding region by the processor shield, including the *shielded SBST accesses*. The processor shield now plays the role of the MMU that performs the virtual address to physical address translation. As a result, the SBST test program is safely executed without interfering the contexts of other processes. After the test program is finished, the return routine, which jumps from the SBST process to the OS, will be described in the section III-D.

For a random generated SBST program, the processor shield works like a random-to-sequential address converter. The processor shield generates the sequential instruction addresses regardless of what addresses the processor generates, the same approach used in our previous work [3].

B. External Signal Control

An embedded processor can be plugged into various systems, which have their own interrupt controllers. Using the system interrupt controller to send the interrupt signal degrades the portability of the SBST program due to the customized functions. As an alternative, we use the processor shield to send the interrupt signal, and this simplifies the test program design and increases the test code portability. During testing, the processor shield also blocks the system interrupt so the SBST process becomes a non-preemptive process.

C. Result Compression

A feasible method to compress the test results is to employ a Multiple-Input Signature Register (MISR) on the output bus of the processor [4]. During an on-line SBST, the bus contention and the memory latency make the execution cycles uncertain. Therefore, MISR can’t capture the output values of the processor bus every cycle. In our design, there are only two situations that trigger the MISR to ignore the current output signals of the processor bus; otherwise, the MISR captures the bus signals every cycle. The first is that when the processor requests the system bus and waits for the granted signal from the arbiter. Since the processor is stalled for waiting the arbitration, the MISR ignores these output values. The second is that when the processor gets the right to access the system bus and waits for the ready response from the memory. The MISR also ignores these bus outputs because the waiting cycles caused by the memory latency are unpredictable.

D. Time-out and Return Function

Since a faulty effect may change the expected program flow, leading to an infinite loop, or even enabling MMU/cache, a time-out counter is needed. For an on-line SBST, a simple clock cycle counter to count total cycles is not suitable because the bus contention and the memory latency produce unpredictable waiting cycles. Instead, we implement an instruction counter that counts the fetched instructions and an idle clock cycle counter to count idle cycles between two instruction fetches. When one of the counters timeouts, the processor shield resets the processor. This is not a system reset, and all the other devices remain unaffected. This reset forces the processor to jump to the reset service routine; however, the processor shield still redirects the instruction fetch to the test program region where a service routine serves this reset and returns the control to the OS. The SBST process just reports the test result to the OS and the user.

E. On-line SBST System Call Execution Flow

We package the SBST body into a system call, and the on-line SBST system call execution flow is shown in Fig. 3. Only the OS can run the test procedure, and the user can call the system call to perform the SBST. TABLE I shows the processor shield control instructions, and the mnemonics used in Fig. 3.

TABLE I PROCESSOR SHIELD CONTROL INSTRUCTION LIST

Target Control	Mnemonics	Function Description
External Signal Control	ESC0	Enable/Disable system interrupt isolation
	ESC1	Trigger an interrupt signal
Lock Bus	LB	Lock/Unlock the system bus
Address Redirection Control	ARC0	Set the test program region size
	ARC1	Set the test program region starting address
	ARC2	Set the test data region size
	ARC3	Set the test data region starting address
	ARC4	Set the backup region size
	ARC5	Set the backup region starting address
	ARC6	Disable all the address redirection
	ARC7	Enable the test program address redirection
	ARC8	Enable the test data address redirection
	ARC9	Enable the backup address redirection
	ARC10	Start/Stop a random generated SBST
Time-out Control	TC0	Set instruction counter & enable idle cycle counter
	TC1	Disable all time-out counter
MISR Control	MC0	Enable & Reset MISR
	MC1	Write MISR signature to test data region

When a SBST system call is called, the first functional block isolates the required components for testing from the system, including the processor core, the system bus, the memory unit, and the process itself. This makes other system devices isolated from the SBST process. The SBST process disables the cache, blocks the system interrupt, and locks the system bus. At this moment, the SBST process becomes a non-preemptive process.

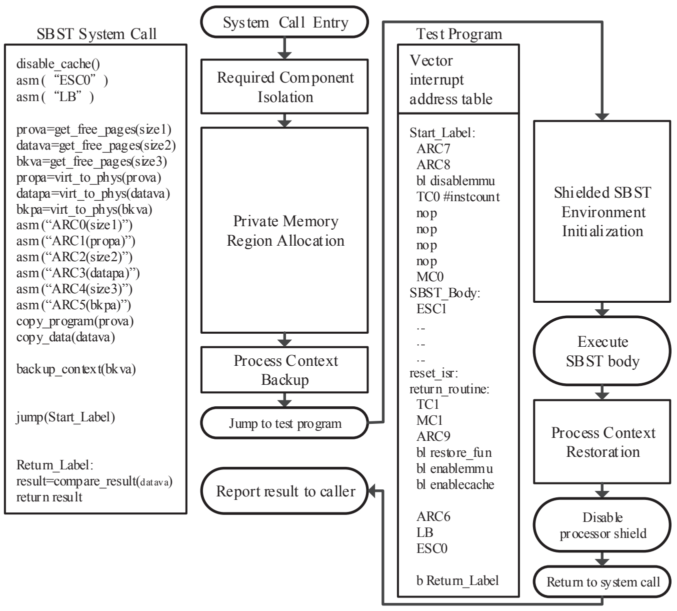
The second functional block is the allocation of private memory regions. In this block, the SBST process acquires three private memory regions from the OS and sends the corresponding physical memory attributes to the processor shield. These memory regions are used for testing and backup. The SBST process copies the test program, which includes the initialization code, SBST body, and the return function, to the test program region. The last function of this block is to copy the test data, which is required for the body, to the test data region.

The SBST process stores its own context, including all contents of registers, the processor status, and the MMU status, to the backup region, and then the SBST process jumps to the test program region, which is prepared previously.

The following function is the initialization of the shielded SBST environment. For the SBST body, this block generates a safe execution environment, which redirects all the instruction fetches and data accesses to the corresponding private memory region allocated by the second function block. All the other system devices and the other processes remain unaffected even if the processor fails the test. The initialization block enables the address redirection, disables the MMU, setups the time-out counter, drains the CPU pipeline with no-ops, and lastly enables the MISR for test signature recording. After performing the above operations, a shielded SBST environment is prepared for the SBST body.

In the SBST body, since all instruction fetches and data accesses are redirected, the SBST body is able to perform the desired access sequence with any address and thus achieve the expected fault coverage, depending on the SBST program used.

When the SBST body is finished, the context restoration function takes over the control of the process. Even if the test reaches a time-out, the context restoration function is still the next functional block. After the MISR signature is stored into the test data region through the processor shield control instruction, the SBST process restores the context, which is stored in the backup region. Then, the MMU is enabled, and the SBST process releases the processor by disabling the processor shield. Finally, the SBST process reports the test result to the test caller.



```
disable_cache()
asm( "ESC0" )
asm( "LB" )

prova=get_free_pages(size1)
datava=get_free_pages(size2)
bkva=get_free_pages(size3)
propa=virt_to_phys(prova)
datapa=virt_to_phys(datava)
bkpa=virt_to_phys(bkva)
asm("ARC0(size1)")
asm("ARC1(propa)")
asm("ARC2(size2)")
asm("ARC3(datapa)")
asm("ARC4(size3)")
asm("ARC5(bkpa)")
copy_program(prova)
copy_data(datava)

backup_context(bkva)


jump(Start_Label)


Return_Label:
result=compare_result(datava)
return result
```

```
Vector
interrupt
address table

Start_Label:
  ARC7
  ARC8
  bl disablemmu
  TC0 #instcount
  nop
  nop
  nop
  nop
  MC0
SBST_Body:
  ESC1
  ..
  ..
  ..
reset_isr:
return_routine:
  TC1
  MC1
  ARC9
  bl restore_fun
  bl enablemmu
  bl enablecache

  ARC6
  LB
  ESC0

  b Return_Label
```

Fig. 3 Proposed On-line SBST System Call Exectuion Flow

IV. CASE STUDY: ARMV5 CORE AND LINUX KERNEL

In this section, we present a case study that executes the SBST program under Linux kernel on an ARMv5-compatible processor system. The target processor is implemented as a five-stage pipeline architecture. We implement 18 processor shield control instructions shown in TABLE I . We also synthesize the processor shield and the ARMv5 processor core in TSMC 40nm process, and the processor frequency is set at 1GHz. TABLE II shows hardware synthesis results. The total hardware overhead is 2.6% compared to the logic part of ARMv5. The additional address latency between the processor bus and the system bus is 0.09ns due to the insertion of the processor shield.

TABLE II ARMV5 PROCESSOR AND PROCESSOR SHIELD DESIGNS

	ARMv5 core	Processor Shield
Area (um^2)	70,880	1,830
Frequency	1 GHz	
Additional address latency		0.09 ns

We build two SBST bodies and construct the corresponding system call. One is a deterministic program which has shorter execution time, and the development method is based on our previous work [5]. Another one is a hybrid program which contains the deterministic and random-generated codes. The method to generate the random code is also proposed in our work [3]. TABLE III shows the instruction count, cycle count, and overall fault coverage. Our processor shield design successfully runs the different types of SBST bodies without crashing the operating system, and the test program gets the expected high fault coverage.

TABLE III ARMV5 PROCESSOR SBST RESULTS

	Deterministic [5]	Hybrid [3]
Instruction Count	2,631	876,132
Cycle Count	16,605	5,282,678
Fault Coverage	94.02%	98.05%

V. CONCLUSION

This paper has addressed the system related issues during on-line processor SBST, including redirecting *shielded SBST accesses*, performing the test non-preemptively, disabling the MMU, and returning the control to the operating system. We propose a low-cost DFT called processor shield along with the required system routines to implement a seamless SBST execution model in the Linux system. We package the SBST bodies into system calls according to the proposed execution flow that successfully runs the test program in the Linux system without affecting other processes. We demonstrate the methodology using a case study that executes the SBST program under the Linux kernel on an ARMv5-compatible processor system. Our method can successfully switch the test process and the kernel process and achieve the expected processor fault coverage. The hardware overhead of the processor shield is about 2.6% compared to the logic part of the processor.

REFERENCES

[1] A. Paschalis, and D. Gizopoulos, "Effective Software-Based Self-Test Strategies for On-Line Periodic Testing of Embedded Processors," *IEEE Trans. Computer-Aided Design Integrated Circuits and System*, vol. 24, no. 1, pp. 88-99, Jan. 2005.

[2] P. Bernardi, L. M. Ciganda, E. Sanchez, and M. S. Reorda, "MIHST: A Hardware Technique for Embedded Microprocessor Functional On-Line Self-Test," *IEEE Trans. Computer*, vol. 63, no. 11, pp. 2760-2771, Nov. 2014.

[3] 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 Integration (VLSI) Systems*, vol. 19, no. 3, pp. 516-520, Mar. 2011.

[4] D.K. Pradhan and S.K. Gupta, "A New Framework for Designing and Analyzing BIST Techniques and Zero Aliasing Compression," *IEEE Trans. Computers*, vol. 40, no. 6, pp. 743-763, June 1991.

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