Software-Based Online Detection of Hardware Defects: Mechanisms, Architectural Support, and Evaluation

Kypros Constantinides[‡] Onur Mutlu[†] Todd Austin[‡] Valeria Bertacco[‡]

[‡] *Advanced Computer Architecture Lab*
University of Michigan
Ann Arbor, MI
{*kypros, austin, valeria*}*@umich.edu*

[†] *Computer Architecture Group*
Microsoft Research
Redmond, WA
onur@microsoft.com

Abstract

As silicon process technology scales deeper into the nanometer regime, hardware defects are becoming more common. Such defects are bound to hinder the correct operation of future processor systems, unless new online techniques become available to detect and to tolerate them while preserving the integrity of software applications running on the system.

This paper proposes a new, software-based, defect detection and diagnosis technique. We introduce a novel set of instructions, called Access-Control Extension (ACE), that can access and control the microprocessor's internal state. Special firmware periodically suspends microprocessor execution and uses the ACE instructions to run directed tests on the hardware. When a hardware defect is present, these tests can diagnose and locate it, and then activate system repair through resource reconfiguration. The software nature of our framework makes it flexible: testing techniques can be modified/upgraded in the field to trade off performance with reliability without requiring any change to the hardware.

We evaluated our technique on a commercial chip-multiprocessor based on Sun's Niagara and found that it can provide very high coverage, with 99.22% of all silicon defects detected. Moreover, our results show that the average performance overhead of software-based testing is only 5.5%. Based on a detailed RTL-level implementation of our technique, we find its area overhead to be quite modest, with only a 5.8% increase in total chip area.

1. Introduction

The impressive growth of the semiconductor industry over the last few decades is fueled by continuous silicon scaling, which offers smaller, faster, and cheaper transistors with each new technology generation. However, challenges in producing reliable components in these extremely dense technologies are growing, with many device experts warning that continued scaling will inevitably lead to future generations of silicon technology being much less reliable than present ones [3, 32]. Processors manufactured in future technologies will likely experience failures in the field due to silicon defects occurring during system operation. In the absence of any viable alternative technology, the success of the semiconductor industry in the future will depend on the creation of cost-effective mechanisms to tolerate silicon defects in the field (*i.e.*, during operation).

The Challenge - Tolerating Hardware Defects: To tolerate permanent hardware faults (*i.e.*, silicon defects) encountered during operation, a reliable system requires the inclusion of three critical capabilities: 1) mechanisms for detection and diagnosis of defects, 2) recovery techniques to restore correct system state after a fault is detected, and 3) repair mechanisms to restore correct system functionality for future computation. Fortunately, research in chip-multiprocessor (CMP) architectures already provides for the latter two requirements. Researchers have pursued the development of global checkpoint and recovery mechanisms, examples of these include SafetyNet [31] and ReVive [22, 19]. These low-cost checkpointing mechanisms provide the capabilities necessary to implement system recovery.

Additionally, the highly redundant nature of future CMPs will allow low-cost repair through the disabling of defective processing elements [27]. With a sufficient number of processing resources, the performance of a future parallel system will gracefully degrade as manifested defects increase. Moreover, the performance impact of each degradation step is expected to decrease substantially as future CMP systems scale to larger numbers of processing elements.

Given the existence of low-cost mechanisms for system recovery and repair, the remaining major challenge in the design of a defect-tolerant CMP is the development of low-cost defect detection techniques. Existing online hardware-based defect detection and diagnosis techniques can be classified into two broad categories 1) *continuous:* those that continuously check for execution errors and 2) *periodic:* those that periodically check the processor's logic.

Existing Defect Tolerance Techniques and their Shortcomings: Examples of *continuous* techniques are Dual Modular Redundancy (DMR) [29] and DIVA [2]. These techniques detect silicon defects by validating the execution through independent redundant computation. However, independent redundant computation requires significant hardware cost in terms of silicon area (100% extra hardware in the case of DMR). Furthermore, continuous checking consumes significant energy and requires part of the maximum power envelope to be dedicated to it. In contrast, *periodic* techniques check periodically the integrity of the hardware without requiring redundant execution [28]. These techniques rely on checkpointing and recovery mechanisms that provide computational epochs and a substrate for speculative unchecked execution. At the end of each computational epoch, the hardware is checked by on-chip testers. If the hardware tests succeed, the results produced during the epoch are committed and execution proceeds to the next computational epoch. Otherwise, the system is deemed defective and system repair and recovery are required.

The on-chip testers employed by periodic defect tolerance techniques rely on the same Built-In-Self-Test (BIST) techniques that are used predominantly during manufacturing testing [6]. BIST techniques use specialized circuitry to generate test patterns and to validate the responses generated by the hardware. There are two main ways to generate test patterns on chip: (1) by using pseudo-random test pattern generators, (2) by storing on-chip previously generated test vectors that are based on a specific fault model. Unfortunately, both of these approaches have significant drawbacks. The first approach does not follow any specific testing strategy (targeted fault model) and therefore requires extended testing times to achieve good fault coverage [6]. The second approach, not only requires significant hardware overhead [7] to store the test patterns on chip, but also binds a specific testing approach (*i.e.*, fault model) into silicon. On the other hand, as the nature of wearout-related silicon defects and the techniques to detect them are under continuous exploration [10], binding specific testing approaches into silicon might be premature and therefore undesirable.

As of today, hardware-based defect tolerance techniques have one or both of the following two major disadvantages:

1. *Cost*: They require significant additional hardware to implement a specific testing strategy,
2. *Inflexibility*: They bind specific test patterns and a specific testing approach (*e.g.*, based on a specific fault model) into silicon. Thus, it is impossible to change the testing strategy and test patterns after the processor is deployed in the field. Flexible defect tolerance solutions that can be modified, upgraded, and tuned in the field are very desirable.

Motivation and Basic Idea: Our goal in this paper is to develop a low-cost, flexible defect tolerance technique that can be modified and upgraded in the field. To this end, *we propose to implement hardware defect detection and diagnosis in software*. In our approach, the hardware provides the necessary substrate to facilitate testing and the software makes use of this substrate to perform the testing. We introduce specialized Access-Control Extension (ACE) instructions that are capable of accessing and controlling virtually any portion of the microprocessor's internal state. Special firmware periodically suspends microprocessor execution and uses the ACE instructions to run directed tests on the hardware and detect if any component has become defective. To provide faster and more flexible software access to different microarchitectural components at low hardware overhead, our scheme leverages the pre-existing scan-chain infrastructure [12] that is conventionally integrated in microprocessor designs today and used during manufacturing testing.

Figure 1 shows how the ACE framework fits in the hardware/software stack below the operating system layer. Our approach provides particularly wide coverage, as it not only tests the internal processor control and instruction sequencing mechanisms through software functional testing, but it can also check all datapaths, routers, interconnect and microarchitectural components by issuing ACE instruction test sequences. We provide a complete defect tolerance solution by incorporating our defect detection and diagnosis technique in a coarse-grained checkpointing and recovery environment. If an ACE test sequence detects that the underlying hardware is defective, the system disables the defective component and restores correct program state by rolling back to the last known correct checkpoint.

1.1 Contributions

With this work we propose a novel software-based technique for online detection of hardware defects. We achieve this through the following contributions:

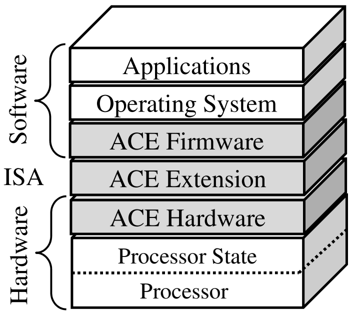


Figure 1: The ACE framework fits in the hardware/software stack below the operating system.

- We provide architectural support to the software layer that enables it to probe and control the underlying microarchitectural state with a high degree of accessibility and observability by introducing Access-Control Extension (ACE) instructions.
- We propose the use of testing firmware that periodically suspends normal system operation and uses the ACE instructions to check the integrity of the underlying hardware.
- We show that the flexibility of software-based defect detection allows the testing techniques to be modified/upgraded in the field without requiring any hardware changes. We also show that our technique can be used in a flexible way to trade-off performance with reliability (*i.e.*, defect coverage).
- We propose a complete defect-tolerance solution that has low hardware cost and performance overhead by incorporating our software-based defect detection and diagnosis technique within a checkpointing and recovery environment.
- We extensively evaluate the effectiveness, performance overhead, and area overhead of our technique on a commercial CMP based on Sun's Niagara [33]. Using commercial ATPG tools and a detailed RTL implementation of the hardware support required for our technique, we show that our technique can provide defect tolerance for 99.2% of the chip area, requiring only 5.8% area overhead. Through cycle-accurate simulation of SPEC CPU2000 benchmarks, we show that the average performance overhead of our technique is only 5.5%.

2. Software-Based Defect Detection and Diagnosis

A key challenge in implementing a software-based defect detection and diagnosis technique is the development of effective software routines to check the underlying hardware. Commonly, software routines for this task suffer from the inherent inability of the software layer to observe and control the underlying hardware, resulting in either excessively long test sequences or poor defect coverage. Current microprocessor designs allow only minimal access to their internal state by the software layer; often all that software can access consists of the register file and a few control registers (such as the program counter (PC), status registers, *etc.*). Although this separation provides protection from malicious software, it also largely limits the degree to which stock hardware can utilize software to test for silicon defects.

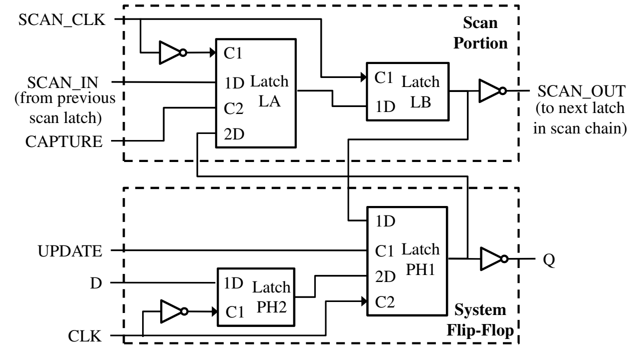


Figure 2: A typical scan flip-flop (adapted from [17])

ACE_set $src,<ACE Domain#>,<ACE Segment#>	Copy *src* register to the scan state (scan portion)
ACE_get $dst,<ACE Domain#>,<ACE Segment#>	Load scan state to register *dst*
ACE_swap <ACE Domain#>,<ACE Segment#>	Swap scan state with processor state (system FFs)
ACE_test	Three cycle atomic operation. Cycle 1: Load test pattern, Cycle 2: Execute for one cycle, Cycle 3: Capture test response
ACE_test <ACE Domain#>	Same as ACE_test but local to the specified ACE domain

Table 1: The ACE instruction set extensions

An example scenario where the lack of observability compromises the efficiency of software testing routines is a defective reorder buffer entry. In this scenario, a software-based solution can detect such a situation only when the defect has propagated to an accessible state, such as the register file, memory, or a primary output. Moreover, without specific knowledge as to how the architectural state was corrupted, the *diagnosis* of the source cause of the erroneous result is extremely challenging.[1]

To overcome this limited accessibility, we propose architectural support through an extension to the processor's ISA. Our extension adds a set of special instructions enabling full observability and control of the hardware's internal state. These Access-Control Extension (ACE) instructions are capable of reading/writing from/to any part of the microprocessor's internal state. ACE instructions make it possible to probe underlying hardware and systematically and efficiently assess if any hardware component is defective.

2.1 An ACE-Enhanced Architecture

A microprocessor's state can be partitioned into two parts: accessible from the software layer (*e.g.*, register file, PC, *etc.*), or not (*e.g.*, reorder buffer, load/store queues, *etc.*). An ACE-enhanced microarchitecture allows the software layer to access and control (almost) all of the microprocessor's state. This is done by using *ACE instructions* that copy a value from an architectural register to any other part of the microprocessor's state, and *vice versa*.

This approach inherently requires the architecture to access the underlying microarchitectural state. To provide this accessibility without a large hardware overhead, we leverage the existing scan chain infrastructure. Most modern processor designs employ full hold-scan techniques to aid and automate the manufacturing testing process [12, 34]. In a full scan design, each flip-flop of the processor state is substituted with a scan flip-flop and connected to form one or more shift registers (scan chains) [6]. Figure 2 shows a typical scan flip-flop design [17, 12]. The system flip-flop is used during the normal operating mode, while the scan portion is used during testing to load the system with test patterns and to read out the test responses. Our approach extends the existing scan-chain using a hierarchical, tree-structured organization to provide fast software access to different microarchitectural components.

[1]The sole fact that a hardware fault had propagated to an observable output does not provide information on where the defect originated.

ACE Domains and Segments: In our ACE extension implementation, the microprocessor design is partitioned into several *ACE domains*. An ACE domain consists of the state elements and combinational logic associated with a specific part of the microprocessor. Each ACE domain is further subdivided into *ACE segments*, as shown in Figure 3(a). Each ACE segment includes only a fixed number of storage bits, which is the same as the width of an architectural register (64 bits in our design).

ACE Instructions: Using this hierarchical structure, the ACE instructions can read or write any part of the microprocessor's state. Table 1 shows a description of the ACE instruction set extensions.

`ACE_set` copies a value from an architectural register to the scan state (scan portion in Figure 2) of the specified ACE segment. Similarly, `ACE_get` loads a value from the scan state of the specified ACE segment to an architectural register. These two instructions can be used for manipulating the scan state through software-accessible architectural state. The `ACE_swap` instruction is used for swapping the scan state with the processor state (system FFs) of the ACE segment by asserting both the UPDATE and the CAPTURE signals (see Figure 2).

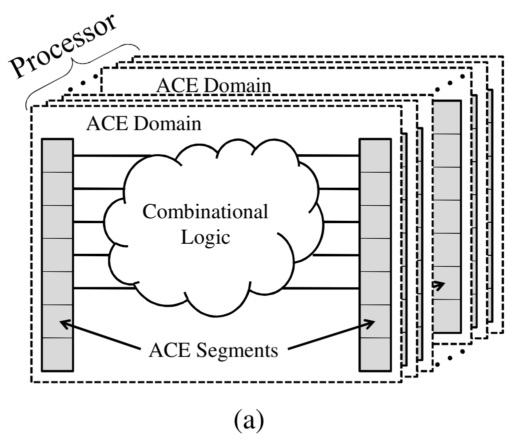
Finally, `ACE_test` is a test-specific instruction that performs a three-cycle atomic operation for orchestrating the actual testing of the underlying hardware (see Section 2.2 for an example). `ACE_test` is used after the scan state is loaded with a test vector using the `ACE_set` instruction. In the first cycle, the scan state is swapped with the processor state. The second cycle is the actual test cycle in which the processor executes for one clock cycle.[2] In the third cycle, the processor state is swapped again with the scan state. The last swap restores the processor state in order to continue normal execution and moves the test response back to the scan state where it can be validated using the `ACE_get` instruction. We also provide a version of `ACE_test` that takes as argument an ACE domain index, which allows testing to be performed locally only in the specified domain.[3]

ACE Tree: During the execution of an ACE instruction, data needs to be transferred from the register file[4] to any part of the chip that contains microarchitectural state. In order to avoid long interconnect, which would require extra repeaters and buffering circuitry, the data transfer between

[2]Note that this is analogous to single-stepping in software debugging.

[3]`ACE_test` is logically the same as an atomic combination of `ACE_swap`, followed by a single test cycle, followed by another `ACE_swap`.

[4]Either from general-purpose architectural registers or from special-purpose architectural registers.



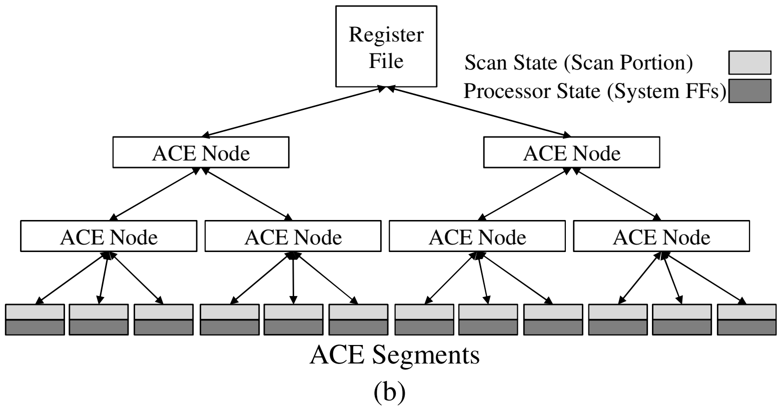


Figure 3: *The ACE Architecture:* (a) the chip is partitioned into multiple ACE domains. Each ACE domain includes several ACE segments. The union of all ACE segments comprises the full chip's state (excluding SRAM structures). (b) data is transferred from/to the register file to/from an ACE segment through the bidirectional ACE tree.

the register file and the ACE segments is pipelined through the *ACE tree*, as shown in Figure 3(b). At the root of the ACE tree is the register file while the ACE segments are its leaves. At each intermediate tree level there is an *ACE node* that is responsible for buffering and routing the data based on the executed operation. The ACE tree is a bidirectional tree allowing data transfers from the register file to the ACE segments and back. By designing the ACE tree as a balanced tree (all paths have the same length), each ACE instruction that reads/writes any segment of the microprocessor state takes the same number of clock cycles (*i.e.*, the tree's depth). Note that ACE instructions can be executed in a pipelined fashion over the ACE tree.

In a uniprocessor system the ACE topology is the simplest possible, since it consists of a single ACE tree rooted at the processor's register file. However, CMP systems consisting of several cores, on-chip caches, and supporting modules such as memory controllers and cross-bars (*i.e.*, *non-core* modules) that might require more complex ACE topologies. In CMP systems it is possible to design multiple ACE trees, each originating from a distinct register file of the multiple cores in the system. Since non-core modules usually do not have instruction execution capabilities, they cannot include an ACE tree of their own. Therefore, in our implementation, each core's ACE tree spans over the core's resources as well as over non-core modules.

In order to avoid any malicious use of the ACE infrastructure, ACE instructions are privileged instructions that can be used only by *ACE firmware*. ACE firmware routines are special applications running between the operating system layer and the hardware in a trusted mode, similarly to other firmware, such as device drivers. Each microprocessor vendor can keep the specific mapping between the microprocessor's state and the ACE domains/segments as classified information for security reasons. Therefore, we expect that ACE firmware will be developed by microprocessor vendors and distributed to the customers.

2.2 ACE-Based Online Testing

ACE instruction set extensions make it possible to craft programs that can efficiently and accurately detect underlying hardware defects. The approach taken in building test programs, however, must have high-coverage, even in the presence of defects that might affect the correctness of ACE instruction execution and test programs. This section describes how test programs are designed.

ACE Testing and Diagnosis: Special firmware periodically suspends normal processor execution and uses the ACE infrastructure to perform high-quality testing of the underlying hardware. A test program exercises the underlying hardware with previously generated test patterns and validates the test responses. Both the test patterns and the associated test responses are stored in physical memory. The pseudo-code of a firmware code segment that applies a test pattern and validates the test response is shown in Figure 4. First, the test program stops normal execution and uses the `ACE_set` instruction to load the scan state with a test pattern (Step 1). Once the test pattern is loaded into the scan state, a three-cycle atomic `ACE_test` instruction is executed (Step 2). In the first cycle, the processor state is loaded with the test pattern by swapping the processor state with the scan state (as described in the previous section). The next cycle is the actual test cycle where the combinational logic generates the test response. In the third cycle, by swapping again the processor state with the scan state, the processor state is restored while the test response is copied to the scan state for further validation. The final phase (Step 3) of the test routine uses the `ACE_get` instruction to read and validate the test response from the scan state. If a test pattern fails to produce the correct response at the end of Step 3, the test program indicates which part of the hardware is defective[5] and disables it through system reconfiguration [27, 8]. If necessary, the test program can run additional test patterns to narrow down the defective part to a finer granularity.

Given this software-based testing approach, the firmware designer can easily change the level of defect coverage by varying the number of test patterns. As a test program executes more patterns, coverage increases. To generate compact test pattern sets adhering to specific fault models we use automatic test pattern generation (ATPG) tools [6].

[5]By interpreting the correspondence between erroneous response bits and ACE domains.

Step 1: Test Pattern Loading

```
// load test pattern to scan state
for(i=0;i<#_of_ACE_Domains;i++){
  for(j=0;j<#_of_ACE_Segments;j++){
    load    $r1,pattern_mem_loc
    ACE_set $r1, i, j
    pattern_mem_loc++
}
```

Step 2: Testing

```
// Three cycle operation
// 1)load test pattern
// to processor state
// 2)execute for one cycle
// 3)capture test response &
// restore processor state
ACE_test
```

Step 3: Test Response Validation

```
// validate test response
for(i=0;i<#_of_ACE_Domains;i++){
  for(j=0;j<#_of_ACE_Segments;j++){
    load    $r1,test_resp_mem_loc
    ACE_get $r2, i, j
    if ($r1!=$r2) then ERROR else
    test_resp_mem_loc++
}
```

Figure 4: *ACE firmware:* Pseudo-code for 1) loading a test pattern, 2) testing, and 3) validating the response.

Basic Core Functional Testing: When performing ACE-based testing, there is one initial challenge to overcome: ACE-based testing firmware relies on the correctness of a set of basic core functionalities which load test patterns, execute ACE instructions, and validate the test response. If the core has a defect that prevents the correct execution of the ACE firmware, then ACE testing cannot be performed reliably. To bypass this problem, we craft specific programs to test the basic functionalities of a core before running any ACE testing firmware. If these programs do not report success in a timely manner to an independent auditor (*e.g.*, the operating system running on the other cores), then we assume that an irrecoverable defect has occurred on the core and we permanently disable it. If the basic core functionalities are found to be intact, finer-grained ACE-based testing can begin. Although these basic functionality tests do not provide high-quality testing coverage, they provide enough coverage to determine if the core can execute the targeted ACE testing firmware with a very high probability. A similar technique employing software-based functional testing was used for the manufacturing testing of Pentium 4 [21].

Testing Frequency: Device experts suggest that the majority of wearout-related defects manifest themselves as progressively slow devices before eventually leading to a permanent breakdown [3, 13]. Therefore, the initial observable symptoms of most wearout-related defects are timing violations. To detect such wearout-related defects early, we employ a test clock frequency that is slightly faster than the operating frequency. We extend the existing dynamic voltage/frequency scaling mechanisms employed in modern processors [15] to support a frequency that is slightly higher than the fastest used during normal operation.[6]

2.3 ACE Testing in a Checkpointing and Recovery Environment

We incorporate the ACE testing framework within a multiprocessor checkpointing and recovery mechanism (*e.g.*, SafetyNet [31] or ReVive [22]) to provide support for system-level recovery. When a defect is detected, the system state is recovered to the last checkpoint (*i.e.*, correct state) after the system is repaired.

In a checkpoint/recovery system, the release of a checkpoint is an irreversible action. Therefore, the system must execute the ACE testing firmware at the end of each checkpoint interval to test the integrity of the whole chip. A checkpoint is released only if ACE testing finds no defects. With this policy, the performance overhead induced by running the ACE testing firmware depends directly on the length of the checkpoint interval, that is, longer intervals lead to lower performance overhead. We explore the trade-off between checkpoint interval size and ACE testing performance overhead in Section 4.5.

To achieve long checkpoint intervals, I/O operations need to be handled carefully. I/O operations such as filesystem/monitor writes or network packet transmissions are irreversible actions and can force an early checkpoint termination. To avoid premature terminations, we buffer I/O operations as proposed in [19]. Alternatively, the operating system can be modified to allow speculative I/O operations as described in [20], an option we have not explored.

2.4 Flexibility of ACE Testing

The software nature of ACE-based testing inherently provides a more flexible solution than hardwired solutions. The major advantages offered by this flexibility are:

Dynamic tuning of the performance-reliability trade-off: The software nature of ACE testing provides the ability to dynamically trade-off performance with reliability (defect coverage). For example, when the system is running a critical application demanding high system reliability, ACE testing firmware can be run more frequently with higher quality and higher coverage targets (*i.e.*, use of different fault models and more test patterns). On the other hand, when running a performance critical application with relatively low reliability requirements (*e.g.*, mpeg decompression), the ACE testing frequency can be reduced.

Utilization-oriented testing: ACE testing allows the system to selectively test only those resources utilized by the running applications. For example, if the system is running integer-intensive applications, there might be no need to test unutilized FPU resources.

Upgradability: Both fault models and ATPG tools are active research areas. Researchers continuously improve the quality and coverage of the generated test patterns. Therefore, during the lifetime of a processor, numerous advances will improve the quality and test coverage of the ATPG patterns. The software nature of ACE testing allows processor vendors to periodically issue ACE firmware updates that can incorporate these advances, and thus improve the defect detection quality during the processor's lifetime.

Adaptability: ACE testing allows vendors to adapt the testing method based on in-the-field analysis of likely defect scenarios. For example, if a vendor observes that the failure of a specific processor is usually originating from a particular module, they can adapt the ACE testing firmware to prioritize efforts on that particular module.

[6]The safeguard margins used in modern microprocessors (to tolerate process variation) allow the use of a slightly faster testing frequency with a negligible number of false positives [9].

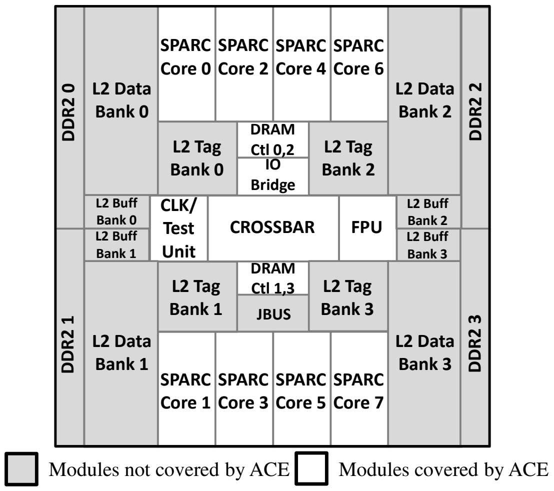


Figure 5: *ACE coverage of the OpenSPARC T1 processor:* Modules that are dominated by SRAM structures, such as on-chip caches, are not covered by ACE testing since they are already protected by ECC.

3. Experimental Methodology

To evaluate our software-based defect detection technique we used the OpenSPARC T1 architecture, the open source version of the commercial UltraSPARC T1 (Niagara) processor from Sun [33], as our experimental testbed.

The OpenSPARC T1 processor implements the 64-bit SPARC V9 architecture and targets commercial applications such as application servers and database servers. It contains eight SPARC processor cores, each with full hardware support for four threads. The eight cores are connected through a crossbar to a unified L2 cache (3MB). The chip also includes four memory controllers and a shared FPU unit [33].

First, using the processor's RTL code, we divided the processor into ACE domains. We made this partition based on functionality, where each domain comprises a basic functionality module in the RTL code. When dividing the processor into ACE domains we excluded modules that are dominated by SRAM structures (such as caches) because such modules are already protected with error-coding techniques such as ECC. Figure 5 shows the processor modules covered by the ACE framework (note that the L1 caches within each core are also excluded). Overall, our RTL implementation of the ACE framework consists of 79 ACE domains, each domain including on average 45 64-bit ACE segments. The whole chip comprises roughly 235K ACE-accessible bits.

Next, we used the Synopsys Design Compiler to synthesize each ACE domain using the Artisan IBM 0.13um standard cell library. We used the Synopsys TetraMAX ATPG tool to generate the test patterns. TetraMAX takes as input the gate-level synthesized design, a fault model, and a test coverage target and tries to generate the minimum set of test patterns that meet the test coverage target.

Fault Models: In our studies we explored several single-fault models: stuck-at, N-detect and path-delay. The stuck-at fault model is the industry standard model for test pattern generation. It assumes that a circuit defect behaves as a node stuck at 0 or 1. However, previous research has shown that the test pattern sets generated using the N-detect fault model are more effective for both timing and hard failures, and present higher correlation to actual circuit defects [16, 10]. In the N-detect test pattern sets, each single stuck-at fault is detected by at least N different test patterns. As expected, the benefit of more effective testing by using the N-detect model comes with the overhead of larger test pattern set sizes and longer testing times. To provide the flexibility of dynamically trading off between reliability and performance, we generate test pattern sets using both fault models.

In addition to the stuck-at and N-detect fault models, we also generate test pattern sets using the path-delay fault model [6]. This fault model tests the design for delay faults that can cause timing violations. The test patterns generated using the path-delay fault model exercise the circuit's paths at-speed to detect whether a path is too slow due to manufacturing defects, wearout-related defects, or process variation.

Benchmarks: We used a set of benchmarks from the SPEC CPU2000 suite to evaluate the performance overhead and memory logging requirements of ACE testing. All benchmarks were run with the reference input set.

Microarchitectural Simulation: To evaluate the performance overhead of ACE testing, we modified the SESC simulator [25] to simulate a SPARC core enhanced with the ACE framework. The simulated SPARC core is a 6-stage, in-order core (with 16KB IL1 and 8KB DL1 caches) running at 1GHz [33].[7] For each simulation run, we skipped the first billion instructions and then performed cycle-accurate simulation for different checkpoint interval lengths (10M, 100M and 1B dynamic instructions). To obtain the number of clock cycles needed for ACE testing, we simulated a process that was emulating the ACE testing functionality.

Experiments to Determine Memory Logging Requirements: To evaluate the memory logging storage requirements of coarse-grained checkpointing, we used the Pin x86 binary instrumentation tool [14]. We wrote a Pin tool that measures the storage needed to buffer the cache lines written back from the L2 cache to main memory during a checkpoint interval, based on the ReVive checkpointing scheme [22]. Note that only the first L2 writeback to a memory address during the checkpoint interval causes the old value of the cache line to be logged in the buffer. 64 bytes (same as our cache line size) are logged for each L2 writeback. Benchmarks were run to completion for these experiments.

RTL Implementation: We implemented the ACE tree structure in RTL using Verilog in order to obtain a detailed and accurate estimate of the area overhead of the ACE framework. We synthesized our design of the ACE tree using the same tools, cell library and methodology that we used for synthesizing the OpenSPARC T1 modules, as described earlier in this section.

4. Experimental Evaluation

4.1 Basic Core Functional Testing

Before running the ACE testing firmware, we first run a software functional test to check the core for defects that would prevent the correct execution of the testing firmware. If this test does not report success in a timely manner to an independent auditor (*i.e.*, the OS running on other cores),

[7]SESC provides a configuration file for the OpenSPARC T1 processor, which we used in our experiments.

Control Flow Assertion	Incorrect execution during the control flow test.
Register Access Assertion	Incorrect execution during the register access test.
Incorrect Execution Assertion	The final result of the test is incorrect.
Early Termination	The execution terminated without executing all the instructions (wrong control flow)
Execution Timeout	The test executed for more than the required clock cycles (wrong control flow, e.g., infinite loop)
Illegal Execution	The test executed an illegal instruction (e.g., an instruction with an invalid opcode)
Memory Error	Memory request for an invalid memory address
Undetected Fault	The test executed correctly

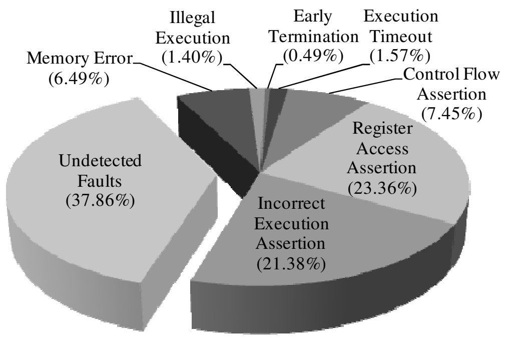


Figure 6: *Fault coverage of basic core functional testing:* The pie chart on the right shows the distribution of the outcomes of a fault injection campaign on a 5-stage in-order core running the purely software-based preliminary functional tests.

the test is repeated to verify that the failing cause was not transient. If the test fails again then an irrecoverable core defect is assumed, the core is disabled, and the targeted tests are canceled.

The software functional test we used to check the core consists of three self-validating phases. The first phase runs a basic control flow check where 64 basic blocks are executed in a non-sequential control flow and each of the 64 basic blocks sets the value of a bit in a 64-bit architectural register. At the end of the phase, a control flow assertion checks the value of the register to determine whether or not the execution was correct. The second phase checks the core's capability to access the register file. This phase consists of a sequence of data-dependent ALU instructions that eventually read and write all architectural registers. At the end of this phase, the final result of this chain of computation is checked by an assertion. The final phase of the basic core test consists of a sequence of dependent instructions that uses each of the instructions in the ISA at least once. The final result of the functional test is checked by an assertion that validates the last generated value. The total size of the software functional test is about 700 dynamic instructions.

To evaluate the effectiveness of the basic core test, we performed a stuck-at fault injection campaign on the gate-level netlist of a synthesized 5-stage in-order core (similar to the SPARC core with the exception of multithreading support). Figure 6 shows the distribution of the outcomes of the fault injection campaign. Overall, basic core test successfully detected 62.14% of the injected faults. The remaining 37.86% of the injected faults lied in parts of the core's logic that do not affect the core's capability of executing simple programs such as the basic core test and the ACE testing firmware. ACE testing firmware will subsequently test these untested areas of the design to provide full core coverage.

These results also demonstrate that software-based functional tests that, unlike the ACE testing firmware, do not have access/control on the core's internal state, are inadequate to provide a high-quality, high-coverage test of the underlying hardware. Similar software functional testing techniques were used for the manufacturing testing of the Intel Pentium 4 [21]. The coverage of these tests as reported in [21] is in the range of 70% which corroborates the results we observed from our fault-injection campaign on a simpler Niagara-based core.

4.2 ACE Testing Latency and Coverage

An important metric for measuring the efficiency of our technique is how long it takes to fully check the underlying hardware for defects. The latency of testing an ACE domain depends on (1) the number of ACE segments it consists of and (2) the number of test patterns that need to be applied. In this experiment, we generate test patterns for each individual ACE domain in the design using three different fault models (stuck-at, path-delay and N-detect) and the methodology described in Section 3. Table 2 lists the number of test instructions needed to test each of the major modules in the design (based on the ACE firmware code shown in Figure 4).

For the stuck-at fault model, the most demanding module is the SPARC core, requiring about 150K dynamic test instructions to complete the test. Modules dominated by combinational logic, such as the SPARC core, the DRAM controller, the FPU, and the I/O bridge are more demanding in terms of test instructions. On the other hand, the CPU-cache crossbar that consists mainly of buffer queues and interconnect requires much fewer instructions to complete the tests.

For the path-delay fault model, we generate test pattern sets for the critical paths that are within 5% of the clock period. The required number of test instructions to complete the path-delay tests is usually less than or similar to that required by the stuck-at model. Note that, with these path-delay test patterns, a defective device can cause undetected timing violations only if it is not in any of the selected critical paths and it causes extra delays greater than 5% of the clock period. We believe that this probability is extremely low, however, stricter path selection strategies can provide higher coverage if deemed necessary (with a higher testing latency). In our design we found that our path selection strategy does not lead to a large number of selected paths. However, in designs where delays of the majority of paths are within 5% of the clock period, more sophisticated path selection strategies can keep the number of selected paths low while maintaining high test coverage [18].

For the N-detect fault model, the number of test instructions is significantly more than that needed for the stuck-at model. This is because many more test patterns are needed to satisfy the N-detect requirement. For values of N higher than four, we observed that the number of test patterns gen-

Module	Area (mm^2)	ACE Accessible Bits	Stuck-at Test Insts	Test Coverage (%)	Path-Delay Test Insts	N-detect Test Insts	
						N = 2	N = 4
SPARC CPU Core (sparc)	8x17=136	8x19772=158176	152370	100.00	110985	234900	434382
CPU-Cache Crossbar (ccx)	14.0	27645	67788	100.00	10122	117648	200664
Floating Point Unit (fpu)	4.6	4620	88530	99.95	31374	126222	212160
e-Fuse Cluster (efc)	0.2	292	11460	94.70	4305	33000	68160
Clock and Test Unit (ctu)	2.3	4205	68904	92.88	10626	126720	240768
I/O Bridge (iobdg)	4.9	10775	110274	100.00	31479	171528	316194
DRAM controller (dram_ctl)	2x6.95=13.9	2x14201=28402	122760	91.44	126238	204312	365364
Total	175.9	234115		99.22			

Table 2: Number of test instructions needed to test each of the major modules in the design.

Module	Cores [0,1] Test Insts		Cores [2,4] Test Insts		Cores [3,5] Test Insts		Cores [6,7] Test Insts	
	Stuck-at	Path-delay	Stuck-at	Path-delay	Stuck-at	Path-delay	Stuck-at	Path-delay
1 x SPARC CPU Core	152370	110985	152370	110985	152370	110985	152370	110985
1/8 x CPU-Cache Crossbar	8474	1265	8474	1265	8474	1265	8474	1265
1/2 x Floating Point Unit							44265	15687
1/2 x e-Fuse Cluster					5730	2153		
1/2 x Clock and Test Unit	34452	5313						
1/2 x I/O Bridge			55137	15740				
1/2 x DRAM controller (pair)			61380	63119	61380	63119		
Total	195296	117563	277361	191109	227954	177522	205109	127937
Stuck-at + Path-delay total	312859		468470		405476		333046	

Table 3: *Number of test instructions needed by each core pair in full-chip distributed testing:* The testing process is distributed over the chip's eight SPARC cores. Each core is assigned to test its resources and some parts of the surrounding non-core modules as shown in this table.

erated increases almost linearly with N, an observation that is aligned with previous studies [16, 10].

Full Test Coverage: The overall chip test coverage for the stuck-at fault model is 99.22% (shown in Table 2). The only modules that exhibit test coverage lower than 99.9% are the e-Fuse cluster, the clock and test unit, and the DRAM controllers, which exhibit the lowest test coverage at 91.44%. The relatively low test coverage in these modules is due to ATPG untestability of some portions of the combinational logic. In other words, no test patterns exist that can set a combinational node to a specific value (lack of controllability), or propagate a combinational node's value to an observable node (lack of observability). If necessary, a designer can eliminate this shortcoming by adding dummy intermediate state elements in the circuit to enable controllability and observability of the ATPG untestable nodes. The test coverage for the two considered N-detect fault models is slightly less than that of the stuck-at model, at 98.88% and 98.65%, respectively (not shown in Table 2 for simplicity).

4.3 Full-Chip Distributed Testing

In the OpenSPARC T1 architecture, the hardware testing process can be distributed over the chip's eight SPARC cores. Each core has an ACE tree that spans over the core's resources and over parts of the surrounding non-core modules (*e.g.*, the CPU-cache crossbar, the DRAM controllers *etc.*). Therefore, each core is assigned to test its resources and some parts of the surrounding non-core modules.

We distributed the testing responsibilities of the non-core modules to the eight SPARC cores based on the physical location of the modules on the chip (shown in Figure 5). Table 3 shows the resulting distribution. For example, each of the cores *zero* and *one* are responsible for testing a full SPARC core, one eighth of the CPU-cache crossbar and one half of the clock and test unit. Therefore, cores *zero* and *one* need 195K dynamic test instructions to test for stuck-at faults and 117K instructions to test for path-delay faults in the parts of the chip they are responsible for. Note that the ACE tree of a core is designed such that it covers all the non-core areas that the core is responsible for testing.

The most heavily loaded pair of cores are cores *two* and *four*. Each of these two cores is responsible for testing its own resources, one eighth of the CPU-cache crossbar, one half of the DRAM controller and one half of the I/O bridge, for a total of 468K dynamic test instructions (for both stuck-at and path-delay testing). The overall latency required to complete the testing of the entire chip is driven by these 468K dynamic test instructions, since all the other cores have shorter test sequences and will therefore complete their tests sooner.

4.4 Memory Logging Requirements of Coarse-grained Checkpointing

The performance overhead induced by running the ACE testing firmware depends on the testing firmware's execution time and execution frequency. When ACE testing is coupled with a checkpointing and recovery mechanism, in order to reduce its execution frequency, and therefore its performance overhead, coarse-grained checkpointing intervals are required.

Figure 7 explores the memory logging storage requirements for such coarse-grained checkpointing intervals on the examined SPEC CPU2000 benchmarks. The memory log size requirements are shown for a system with a 2MB L2 data cache (recall that memory logging is performed only for the first L2 writeback of a cacheline to main memory in a checkpoint interval [22]). For each benchmark, we show the average and maximum required memory log size for intervals of 10 million, 100 million and 1 billion executed instructions. The maximum metric keeps track of the maximum memory

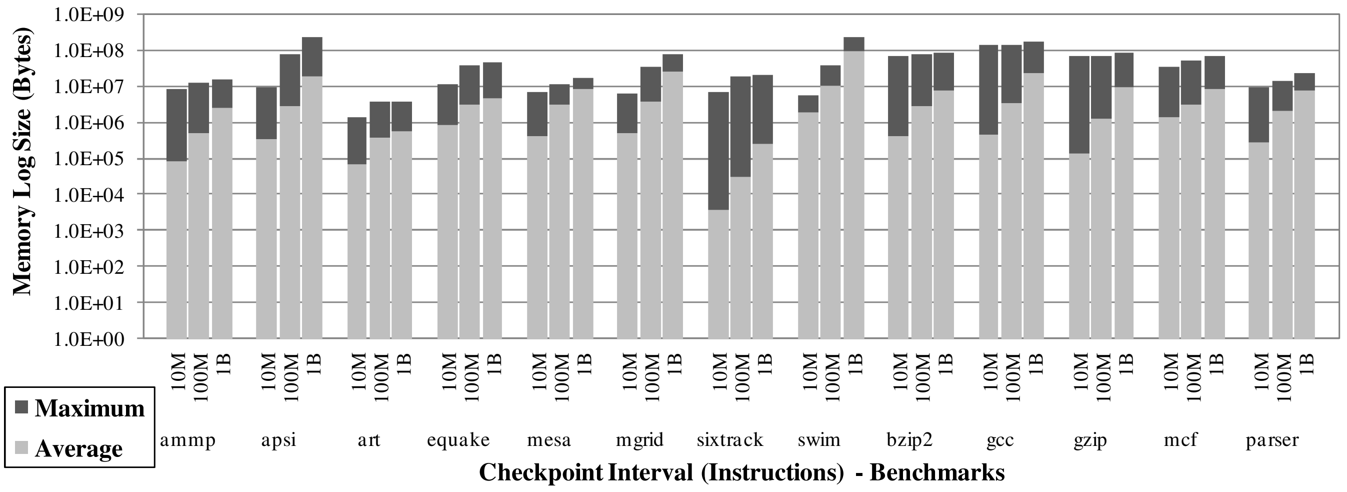


Figure 7: ***Memory logging storage requirements:*** **Average and maximum memory log size requirements for checkpoint intervals of 10 million, 100 million and 1 billion executed instructions.**

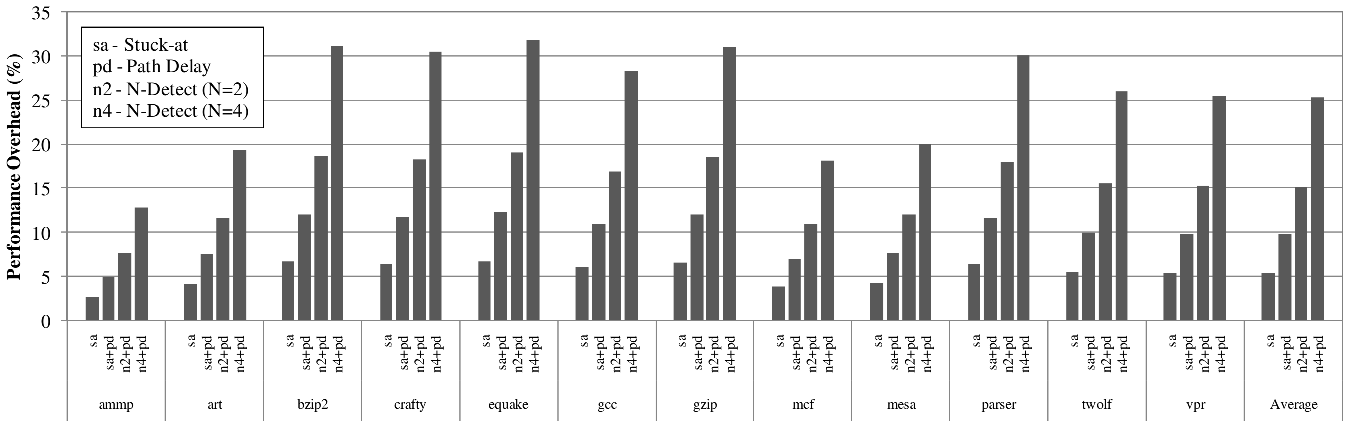


Figure 8: Performance overhead of ACE testing for a 100M instructions checkpoint interval.

log size required in any of the checkpoint intervals during the benchmark's execution, while the average metric averages the memory log size requirement over all the checkpoint intervals (note that the benchmarks were ran to completion with the reference inputs).

We observe that when considering checkpoint intervals that are in the order of 100 million executed instructions, the average memory log size requirements are in the range of a few kilobytes to 10MB. The most demanding benchmark is *swim*: on average it requires 1.8MB, 10MB and 91.4MB respectively for checkpoint intervals of 10M, 100M and 1B instructions. Since the memory log will be maintained at the system's physical memory, the results of this experiment suggest that checkpoint intervals of hundreds of millions of executed instructions are sustainable with insignificant memory storage overhead.[8]

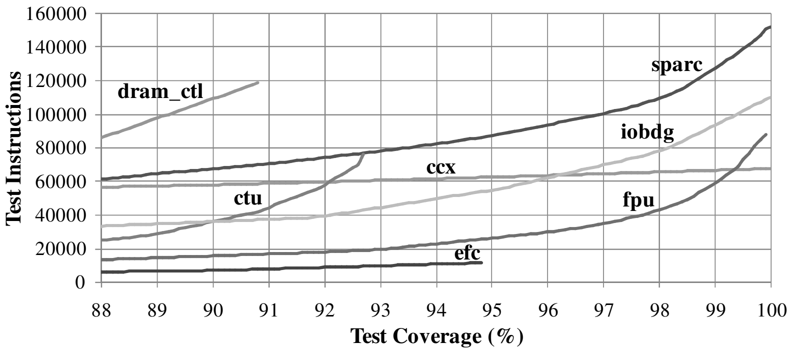
4.5 Performance Overhead of ACE Testing

Figure 8 shows the performance overhead of ACE testing when the checkpoint interval is set to 100M instructions. At the end of each checkpoint interval, normal execution is suspended and ACE testing is performed. In these experiments, the ACE testing firmware executes until it reaches the maximum test coverage. The four bars show the performance overhead when the fault model used in ACE testing is i) stuck-at, ii) stuck-at and path-delay, iii) N-detect (N=2) and path-delay, and iv) N-detect (N=4) and path-delay.

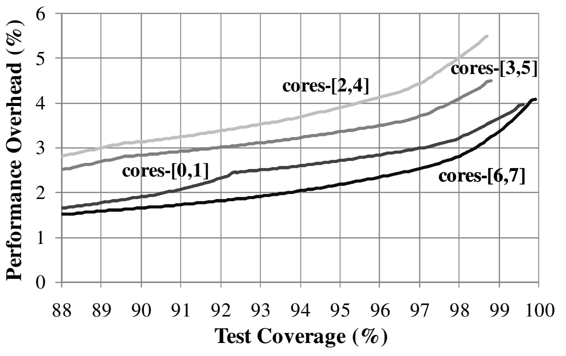
The minimum average performance overhead of ACE testing is 5.5% and is observed when only the industry-standard stuck-at fault model is used. When the stuck-at fault model is combined with the path-delay fault model to achieve higher testing quality, the average performance overhead increases to 9.8%. When test pattern sets are generated using the N-detect fault model, the average performance overhead is 15.2% and 25.4%, for N=2 and N=4 respectively.

Table 4 shows the trade-off between memory logging storage requirements and performance overhead for checkpoint intervals of 10M, 100M and 1B dynamic instructions. Both log size and performance overhead are averaged over all evaluated benchmarks. As the checkpoint interval size increases, the required log size increases, but the performance overhead of ACE testing decreases. *We conclude that checkpoint intervals in the order of hundreds of millions of instructions are sustainable with reasonable storage overhead, while providing an efficient substrate to perform ACE testing with low performance overhead.*

[8]Note that most current systems are equipped with at least 2GB of physical memory.



(a)



(b)

Figure 9: *Performance overhead of ACE testing vs. test coverage:* (a) number of executed test instructions versus the achieved test coverage for each of the major modules, (b) test coverage versus performance overhead for each core pair in full-chip distributed testing.

Checkpoint Interval	Average Memory Log Size (MB)	Perf. Overhead (%) (Stuck-at)	Perf. Overhead (%) (Stuck-at + Path Delay)
10M Instr.	0.48	53.74	96.91
100M Instr.	2.59	5.46	9.85
1B Instr.	14.94	0.55	0.99

Table 4: Memory log size and ACE testing performance overhead for different checkpoint intervals.

4.6 Performance-Reliability Trade-off

The test coverage achieved by the testing firmware increases as more test instructions are executed (and therefore more test patterns are applied). However, the relation between the number of executed test instructions and the test coverage level is not linear. Figure 9(a) shows the number of executed test instructions versus the test coverage obtained for each of the major modules (using the stuck-at fault model). We observe that for some of the modules there is an exponential increase in the number of instructions needed to earn the last few percentage points of coverage. For example, the number of dynamic instructions required to achieve 100% test coverage for the SPARC core is approximately 152K, almost twice the number of instructions required to achieve 93% coverage.

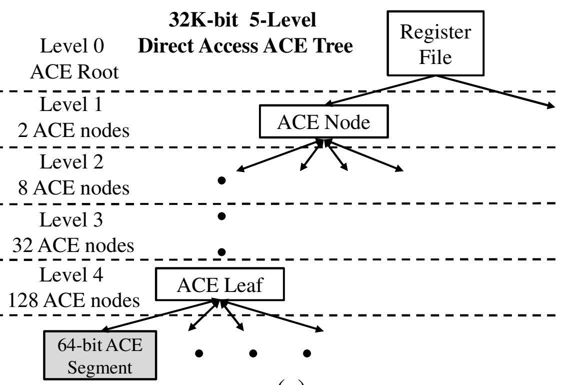
This observation suggests that there is plenty of opportunity to dynamically tune the performance-reliability trade-off in our ACE testing framework. Figure 9(b) shows the test coverage (for the stuck-at model) versus the performance overhead for each core pair (based on the testing partition described in Section 4.3). The results demonstrate that we can dynamically trade-off test coverage for reductions in the performance overhead of testing. For example, the performance overhead for cores *two* and *four* to reach 89% test coverage is only 3%. This is a 46% reduction from the performance overhead of 5.5% to reach 98.7% test coverage. We conclude that the software-based nature of the ACE testing provides a flexible framework to trade-off between test coverage, test quality, and performance overhead.

4.7 ACE Tree Implementation and Area Overhead

The area overhead of the ACE framework is dominated by the ACE tree. In order to evaluate this overhead, we implemented the ACE tree for the OpenSPARC T1 architecture in Verilog and synthesized it with the Synopsys Design Compiler. Our ACE tree implementation consists of data movement nodes that transfer data from the tree root (the register file) to the tree leaves (ACE segments) and *vice versa*. In our implementation, each node has four children and therefore in an ACE tree that accesses 32K bits (about 1/8 of the OpenSPARC T1 architecture), there are 42 internal tree nodes and 128 leaf nodes, where each leaf node has four 64-bit ACE segments as children. Figure 10(a) shows the topology of this ACE tree configuration, which has the ability to directly access any of the 32K bits. To cover the whole OpenSPARC T1 chip with the ACE framework we used eight such ACE trees, one for each SPARC core. The overall area overhead of this ACE framework configuration (for all eight trees) is 18.7% of the chip area.

In order to contain the area overhead of the ACE framework, we propose a hybrid ACE tree implementation that combines the direct processor state accessibility of the previous implementation with the existing scan-chain structure. In this hybrid approach, we divide the 32K ACE-accessible bits into 64 512-bit scan chains. Each scan chain has 64 bits that can be directly accessed through the ACE tree. The reading/writing to the rest of the bits in the scan chain is done by shifting the bits to/from the 64 directly accessible bits. Figure 10(b) shows the topology of the hybrid ACE tree configuration. The overall area overhead of the ACE framework when using the hybrid ACE tree configuration is 5.8% of the chip area.

Notice that although the hybrid ACE tree is a less flexible ACE tree configuration, it does not affect the latency of the ACE testing firmware. The ACE testing firmware accesses the 64 scan chains sequentially. Since there is an interval of at least 64 cycles between two consecutive accesses to the same scan chain, data can be shifted from/to the direct access portion of the chain to/from the rest of the scan chain without producing any stall cycles. For example, during test pattern loading, each 64-bit parallel load to a scan chain is followed by 64 cycles of scan chain shifting. While the parallel loaded data is shifted into the rest of the scan chain in an ACE segment, the testing firmware loads the rest of the scan chains in the other 63 ACE segments. By the time the testing firmware loads the next 64 bits to the scan chain, the previous 64 bits have already been shifted into the scan chain. Similarly, during test response reading, each parallel 64-bit data read is followed by shifting cycles that move the next 64 bits from the scan chain to the direct access portion.



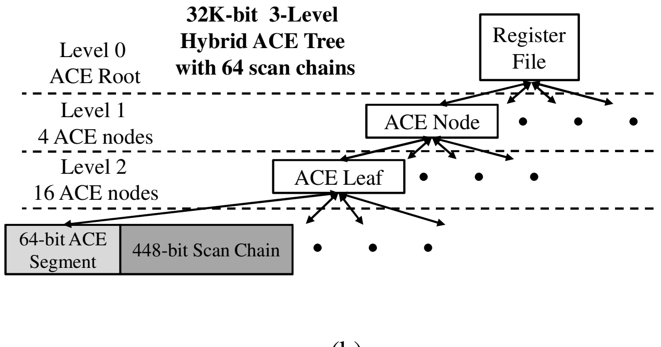


Figure 10: *ACE tree implementation:* (a) the topology of a direct-access ACE tree. (b) the topology of a hybrid (partial direct-access, partial scan-chain) ACE tree.

5. Related Work

Hardware-based Reliability Techniques: The previous work most closely related to ours is [28]. [28] proposed a hardware-based technique that utilizes microarchitectural checkpointing to create epochs of execution, during which on-chip distributed BIST-like testers validate the integrity of the underlying hardware. To lower silicon cost, the testers were customized to the tested modules. However, this leads to increased design complexity because a specialized tester needs to be designed for each module.

A traditional defect detection technique that is predominantly used for manufacturing testing is logic BIST [6]. Logic BIST incorporates pseudo-random pattern generation and response validation circuitry on the chip. Although on-chip pseudo-random pattern generation removes any need for pattern storage, such designs require a large number of random patterns and often provide lower fault coverage than ATPG patterns [6].

Our work improves on these previous works due to the following major reasons: 1) it effectively removes the need for on-chip test pattern generation and validation circuitry and moves this functionality to software, 2) it is not hardwired in the design and therefore has ample flexibility to be modified/upgraded in the field (as described in Section 2.4), 3) it has higher test coverage and shorter testing time because it uses ATPG instead of pseudo-randomly generated patterns; and compared to [28] 4) it can be uniformly applied to any microprocessor module with low design complexity because it does not require module-specific customizations, and 5) it provides wider coverage across the whole chip, including non-core modules.

Previous works proposed the creation of external circuitry fabricated from reliable devices to periodically test the hardware design and to reconfigure faulty parts [1, 11]. In contrast to these works, we propose that the periodic tests be performed by software instead (assisted by ISA extensions) in order to minimize the area cost of testing and to allow more flexibility and adaptability.

Numerous other previous works proposed hardware-based defect tolerance techniques, such as [2, 4, 5]. However, these works focused on providing hardware-based reliability solutions that are limited to processing cores in uniprocessor systems and do not address the testability of non-core modules that are abundant especially in chip-multiprocessor designs.

More recently, Smolens *et al.* [30] proposed a detection technique for emerging wearout defects that periodically runs functional tests that check the hardware under reduced frequency guardbands. Their technique leverages the existing scan chain hardware for generating hashed signatures of the processor's microarchitectural state summarizing the hardware's response to periodic functional tests. This technique allows the software to observe a signature of the microarchitectural state, but it does not allow the software to directly control (*i.e.*, modify) the microarchitectural state. In contrast, our approach provides the software with direct and fast *access and control* of the scan state using the ACE infrastructure. This direct access and control capability allows the software to run online directed hardware tests on any part of the microarchitectural state using high-quality test vectors (as opposed to functional tests that do not directly control the microarchitectural state and do not adhere to any fault model). Furthermore, the proposed direct fast access to the scan state enables the validation of each test response separately (instead of hashing and validating all the test responses together), thereby providing finer-grained defect diagnosis capabilities and higher flexibility for dynamic tuning between performance overhead (*i.e.*, test length) and test coverage.

Software-based Reliability Techniques: To our knowledge, this is the first work that proposes a software-based technique for online hardware defect detection and diagnosis. Only one previous work we are aware of [21] employed purely software-based functional testing techniques during the manufacturing testing of the Intel Pentium 4 processor (see Section 2.2 for a discussion of this work). In our approach, we use a similar functional testing technique (our "basic core functional test" program) to check the basic core functionality before running the ACE firmware to perform directed, high-quality testing.

There are numerous previous works, such as [24, 26], that proposed the use of software-based techniques for online detection of soft errors. However, none of them addresses the problem of online defect detection.

Checkpointing Mechanisms: There is also a large body of work proposing various versions of checkpointing and recovery techniques [31, 22, 19]. Both SafetyNet [31] and ReVive [22] provide general-purpose checkpointing and recovery mechanisms for shared memory multiprocessors. Our defect detection and diagnosis technique is closely coupled with such techniques in providing a substrate of coarse-grained checkpoint intervals that enable efficient ACE testing with low performance overhead.

6. Summary & Conclusions

We introduced a novel, flexible software-based technique, ISA extensions, and microarchitecture support to detect and diagnose hardware defects during online operation of a chip-multiprocessor. Our technique uses the Access-Control Extension (ACE) framework that allows special ISA instructions to access and control virtually any part of the processor's internal state. Based on this framework, we proposed the use of special firmware that periodically suspends the processor's execution and performs high-quality testing of the underlying hardware to detect defects.

Using a commercial ATPG tool and three different fault models, we experimentally evaluated our ACE testing technique on a commercial chip-multiprocessor design based on Sun's Niagara. Our experimental results showed that ACE testing is capable of performing high-quality hardware testing for 99.22% of the chip area, while itself requiring an area overhead of only 5.8% of the chip area based on our detailed RTL-level implementation.

We demonstrated how ACE testing can be coupled seamlessly with a coarse-grained checkpointing and recovery mechanism to provide a complete defect tolerance solution. Our evaluation shows that, with coarse-grained checkpoint intervals, the average performance overhead of ACE testing is only 5.5%. Our results also show that the software-based nature of ACE testing provides ample flexibility to dynamically tune the performance-reliability trade-off at runtime based on system requirements.

7. Future Directions

Looking forward, we believe that the ACE framework is a very general framework that can be helpful in several other applications to amortize its cost. Here, we briefly list two of the possible applications of the ACE framework we are currently investigating:

Post-silicon Debugging: With device scaling entering the nanometer regime, traditional techniques used for post-silicon debugging are becoming less viable. Recently proposed solutions have considerable area overheads and still do not provide complete accessibility to the processor's state [23]. We believe the ACE framework can be an attractive low-overhead framework that provides the post-silicon debug engineers with full accessibility and controllability of the processor's internal state at runtime.

Manufacturing Testing: Today, the processor manufacturing testing process uses automatic test equipment and scan chains for testing the manufactured parts. However, the testers are costly, and they have a limited number of channels designed to drive the scan chains and the pattern loading speed is limited by the maximum scan frequency. We believe that the use of the ACE infrastructure for applying test patterns to the design under test can improve both the speed and the cost of the manufacturing testing process.

Acknowledgments: This work was supported by grants from NSF, SRC and GSRC.

References

[1] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *ICS-4*, pages 1–6, June 1990.

[2] T. M. Austin. DIVA: A reliable substrate for deep submicron microarchitecture design. In *MICRO-32*, pages 196–207, 1999.

[3] S. Borkar, T. Karnik, and V. De. Design and reliability challenges in nanometer technologies. In *DAC-41*, 2004.

[4] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating hard faults in microprocessor array structures. In *DSN*, 2004.

[5] F. A. Bower, D. J. Sorin, and S. Ozev. A mechanism for online diagnosis of hard faults in microprocessors. In *MICRO*, 2005.

[6] M. L. Bushnell and V. D. Agrawal. *Essentials of Electronic Testing for Digital, Memory and Mixed-Signal VLSI Circuits.* Kluwer Academic Publishers, Boston, 2000.

[7] K. Constantinides, J. Blome, S. Plaza, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky. BulletProof: A defect-tolerant CMP switch architecture. In *HPCA-12*, 2006.

[8] W. J. Dally et al. The reliable router: A reliable and high-performance communication substrate for parallel computers. In *PCRCW*, pages 241–255, 1994.

[9] D. Ernst et al. Razor: A low-power pipeline based on circuit-level timing speculation. In *MICRO-36*, 2003.

[10] R. Guo, S. Mitra, E. Amyeen, J. Lee, S. Sivaraj, and S. Venkataraman. Evaluation of test metrics: Stuck-at, bridge coverage estimate and gate exhaustive. In *VTS*, 2006.

[11] J. R. Heath et al. A defect-tolerant computer architecture: Opportunities for nanotechnology. *Science*, 280(5370), 1998.

[12] R. Kuppuswamy et al. Full hold-scan systems in microprocessors: Cost/benefit analysis. *Intel Technology Journal (ITJ)*, 8(1):63–72, Feb. 2004.

[13] Y.-H. Lee et al. Prediction of logic product failure due to thin-gate oxide breakdown. In *IRPS-44*, 2006.

[14] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[15] G. Magklis et al. Profile-based dynamic voltage and frequency scaling for a multiple clock domain microprocessor. In *ISCA-30*, June 2003.

[16] E. J. McCluskey and C.-W. Tseng. Stuck-fault tests vs. actual defects. In *ITC*, pages 336–343, Oct. 2000.

[17] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *Computer*, 38(2):43–52, Feb. 2005.

[18] A. Murakami et al. Selection of potentially testable path delay faults for test generation. In *ITC*, pages 376–384, Oct. 2000.

[19] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas. ReViveI/O: Efficient handling of I/O in highly-available rollback-recovery servers. In *HPCA-12*, 2006.

[20] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. *ACM Transactions on Computer Systems*, 24(4):361–392, 2006.

[21] P. Parvathala, K. Maneparambil, and W. Lindsay. FRITS - A microprocessor functional BIST method. In *ITC*, 2002.

[22] M. Prvulovic, Z. Zhang, and J. Torrellas. ReVive: Cost-effective architectural support for rollback recovery in shared-memory multiprocessors. In *ISCA-29*, pages 111–122, May 25–29 2002.

[23] B. R. Quinton and S. J. E. Wilton. Post-silicon debug using programmable logic cores. In *FPT*, pages 241–248, 2005.

[24] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *CGO-3*, pages 243–254, 2005.

[25] J. Renau, B. Fraguela, J. Tuck, W. Liu, M. Privulovic, L. Ceze, S. Sarangi, P. Sack, K. Stauss, and P. Montesinos. SESC Simulator. http://sesc.sourceforge.net, 2002.

[26] P. P. Shirvani, N. Saxena, and E. J. Mccluskey. Software-implemented EDAC protection against SEUs. *IEEE Transactions on Reliability*, 49:273–284, 2000.

[27] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting microarchitectural redundancy for defect tolerance. In *ICCD*, 2003.

[28] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin. Ultra low-cost defect protection for microprocessor pipelines. In *ASPLOS-12*, pages 73–82, 2006.

[29] D. P. Siewiorek and R. S. Swarz. Reliable computer systems: Design and evaluation, 3rd edition. *AK Peters, Ltd*, 1998.

[30] J. C. Smolens, B. T. Gold, J. C. Hoe, B. Falsafi, and K. Mai. Detecting emerging wearout faults. In *SELSE-3*, 2007.

[31] D. J. Sorin et al. SafetyNet: Improving the availability of shared memory multiprocessors with global checkpoint/recovery. In *ISCA-29*, 2002.

[32] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The impact of technology scaling on lifetime reliability. In *DSN-34*, pages 177–186, 2004.

[33] Sun Microsystems Inc. OpenSPARC T1 Microarchitecture Specification. August 2006.

[34] T. J. Wood. The test and debug features of the AMD-K7 microprocessor. In *ITC*, pages 130–136, 1999.