A Flexible Software-Based Framework for Online Detection of Hardware Defects

Kypros Constantinides, *Student Member*, *IEEE*, Onur Mutlu, *Member*, *IEEE*, Todd Austin, *Member*, *IEEE*, and Valeria Bertacco, *Member*, *IEEE*

Abstract—This work proposes a new, software-based, defect detection and diagnosis technique. We introduce a novel set of instructions, called Access-Control Extensions (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 describe and evaluate different execution models for using the ACE framework. We also describe how the proposed ACE framework can be extended and utilized to improve the quality of post-silicon debugging and manufacturing testing of modern processors. 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 percent of all silicon defects detected. Moreover, our results show that the average performance overhead of software-based testing is only 5.5 percent. Based on a detailed register transfer level (RTL) implementation of our technique, we find its area and power consumption overheads to be modest, with a 5.8 percent increase in total chip area and a 4 percent increase in the chip's overall power consumption.

Index Terms—Reliability, hardware defects, online defect detection, testing, online self-test, post-silicon debugging, manufacturing test.

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 [4], [53]. 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 [52] and ReVive [42], [39]. 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 [48]. With a sufficient number of processing resources, the performance of a future parallel system will gracefully degrade as manifested defects increase.

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) [51], lockstep systems [27], 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 percent extra hardware in the case of DMR and lockstep systems). Furthermore, continuous checking consumes significant energy and requires part of the power envelope to be dedicated to it. In contrast, *periodic* techniques check periodically the integrity of the hardware without requiring redundant execution [50]. 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

- *K. Constantinides, T. Austin, and V. Bertacco, are with the University of Michigan, Ann Arbor, 2260 Hayward, 2773 CSE, MI 48109. E-mail: {kypros, austin, valeria}@umich.edu.*
- *O. Mutlu is with the Carnegie Mellon University, 5000 Forbes Avenue, ECE-HH-A305, Pittsburgh, PA 15213. E-mail: onur@cmu.edu.*

Manuscript received 18 Feb 2008; revised 30 Aug. 2008; accepted 20 Nov. 2008; published online 20 Mar. 2009.
Recommended for acceptance by C. Bolchini.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2008-02-0078.
Digital Object Identifier no. 10.1109/TC.2009.52.

0018-9340/09/$25.00 © 2009 IEEE Published by the IEEE Computer Society

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

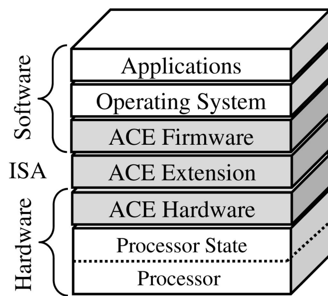


Fig. 1. The ACE framework fits in the hardware/software stack below the operating system.

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 test [7]. BIST techniques use specialized circuitry to generate test patterns and validate the responses generated by the hardware. There are two main ways to generate test patterns on chip: 1) by using pseudorandom test pattern generators and 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 [7]. The second approach not only requires significant hardware overhead [10] 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 [17], 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 upgraded in the field are very desirable.

High-level overview of our approach. Our goal in this work 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.

Fig. 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.

2 Why Does Silicon Fail? A Brief Overview of Silicon Failure Mechanisms

We first provide a brief overview of the silicon failure mechanisms that motivate the solution we propose in this work. The interested reader can refer to [14], [44], [49], [54], [23] for a detailed treatment of these mechanisms.

Time-dependent wearout:

- *Electromigration*: Due to the momentum transfer between the current-carrying electrons and the host metal lattice, ions in a conductor can move in the direction of the electron current. This ion movement is called electromigration [14]. Gradually, this ion movement can cause clustered vacancies that can grow into voids. These voids can eventually grow until they block the current flow in the conductor. This leads to increased resistance and propagation delay, which, in turn, leads to possible device failure. Other effects of electromigration are fractures and shorts in the interconnect. The trend of increasing current densities in future technologies increases the severity of electromigration, leading to a higher probability of observing open and short-circuit nodes over time [18].
- *Gate Oxide Wearout*: Thin gate oxides lead to additional failure modes as devices become subject to gate oxide wearout (or Time-Dependent Dielectric Breakdown, TDDB) [14]. Over time, gate oxides can break down and become conductive. If enough material in the gate breaks down, a conduction path can form from the transistor gate to the substrate, essentially shorting the transistor and rendering it useless [18], [23]. Fast clocks, high temperatures, and voltage scaling limitations are well-established architectural trends that aggravate this failure mode [54].
- *Hot Carrier Degradation (HCD)*: As carriers move along the channel of an MOSFET and experience impact ionization near the drain end of the device, it is possible that they gain sufficient kinetic energy to be injected into the gate oxide [14]. This phenomenon is called Hot Carrier Injection. Hot carriers can degrade the gate dielectric, causing shifts in threshold voltage and eventually device failure. HCD is predicted to worsen for future thinner oxide and shorter channel lengths [23].

Transistor infant mortality. Extreme device scaling also exacerbates early transistor failures. Early transistor failures are caused by weak transistors that escape postmanufacturing validation tests. These weak transistors work initially, but they have dimensional and doping deficiencies that subject them to much higher stress than robust transistors. Quickly (within days to months), they will break down from stress and render the device unusable. Traditionally, early transistor failures have been reduced through aggressive burn-in testing, where, before being placed in the field, devices are subjected to high voltage and temperature testing to accelerate the failure of weak

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

transistors [7]. Those that survive the burn-in testing are likely to be robust devices, thereby ensuring a long product lifetime. However, in the deep-submicron regime, burn-in becomes less effective as devices are subject to thermal runaway effects, where increased temperature leads to increased leakage current, which, in turn, leads to even higher temperatures [37]. The end result is that aggressive burn-in of deep-submicron silicon can destroy even robust devices. Manufacturers are forced to either sacrifice yield by deploying aggressive burn-in testing or experience more frequent early failures in the field by using less aggressive burn-in testing.

Manufacturing defects that escape testing. Optical proximity effects, airborne impurities, and processing material defects can all lead to the manufacturing of faulty transistors and interconnect [44]. Moreover, deep-submicron gate oxides have become so thin that manufacturing variation can lead to currents penetrating the gate, rendering it unusable [49]. Even small amounts of manufacturing variation in the gate oxide could render the device unusable. The problem of manufacturing defects is compounded by the immense complexity of current designs. Design complexity makes it more difficult to test for defects during manufacturing. Vendors are forced to either spend more time with parts on the tester, which reduces profits by increasing time-to-market, or risk the possibility of untested defects escaping to the field. Moreover, in highly complex designs, many defects are not testable without additional hardware support. As a result, even in today's manufacturing environment, untestable defects can escape testing and manifest themselves later on in the field.

Our goal. To overcome the possible errors caused by the aforementioned silicon failure mechanisms, our goal in this work is to develop a flexible, low-cost silicon defect detection and diagnosis technique. We next describe our technique in detail.

3 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.

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 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.

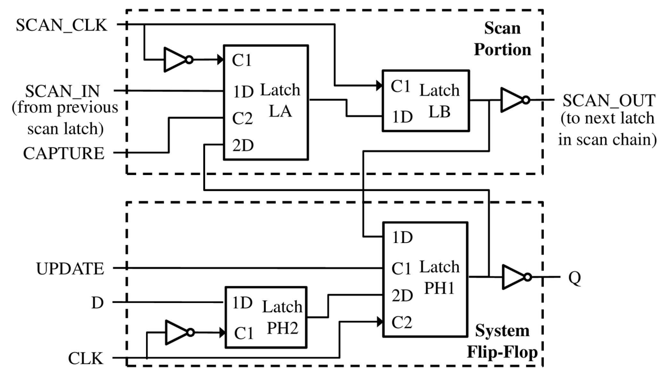


Fig. 2. A typical scan flip-flop (adapted from [38]).

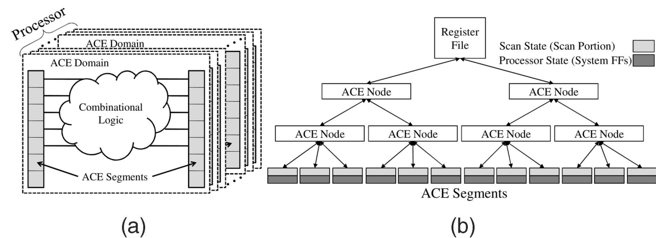


Fig. 3. *The ACE Architecture:* (a) the chip is logically partitioned into multiple ACE domains. Each ACE domain includes several ACE segments. The union of all ACE segments comprises the full chip's state (excluding SRAM structures). (b) Data are transferred from/to the register file to/from an ACE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

3.1 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 accessible (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 [30], [62]. Fig. 2 shows a typical scan flip-flop design [38], [30]. 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.

ACE domains and segments. In our ACE extension implementation, the microprocessor design is logically 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 Fig. 3a. 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, ACE instructions can read or write any part of the microprocessor's state. Table 1 shows a description of the ACE instruction set extensions.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

TABLE 1
The ACE Instruction Set Extensions

Instruction
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

ACE_set copies a value from an architectural register to the scan state (scan portion in Fig. 2) of the specified ACE segment at-speed (i.e., at the processor's clock frequency). Similarly, ACE_get loads a value from the scan state of the specified ACE segment to an architectural register at-speed. 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 flip-flops) of the ACE segment by asserting both the UPDATE and the CAPTURE signals (see Fig. 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 3.2 for example).

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.

ACE tree. During the execution of an ACE instruction, data need to be transferred from the register file 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 the register file and the ACE segments is pipelined through the *ACE tree* as shown in Fig. 3b. 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.

Design complexity. We believe that since the ACE Tree is a regular structure that routes data from the register file to the scan chains and vice versa, its implementation and insertion into the microprocessor implementation can be automated by CAD tools, similar to the way that scan chains are automatically implemented and inserted in current microprocessors today. The main intrusive portion of the ACE Tree that needs interaction with existing processor components are the additional read/write ports needed to connect the root of the ACE Tree to the processor register file. Similarly, the ACE instruction set extensions are likely not intrusive to the microarchitecture since their operations are relatively simple and their implementation does not affect the implementation of other instructions in the ISA.

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++
}
```

Fig. 4. *ACE firmware:* Pseudocode for 1) loading a test pattern, 2) testing, and 3) validating the test response.

3.2 ACE-Based Online Testing

ACE instruction set extensions make it possible to craft programs that can efficiently and accurately detect the 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 pseudocode of a firmware code segment that applies a test pattern and validates the test response is shown in Fig. 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. 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[1] and disables it through system reconfiguration [48], [13].

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. We use automatic test pattern generation (ATPG) tools [7] to generate compact test pattern sets adhering to specific fault models.

Basic core functional testing. When performing ACE testing, there is one initial challenge to overcome: ACE testing firmware relies on the correctness of a set of basic core functionalities that loads test patterns, executes ACE instructions, and validates 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 testing can begin.

1. By interpreting the correspondence between erroneous response bits and ACE domains.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

TABLE 2
Algorithmic Flow of ACE-Based Testing in a Checkpoint/Recovery Environment

Step	Action
1	Run regular application thread until the checkpointing interval is reached
2	Lightweight context switch [1], [28] to ACE testing mode
3	Run basic core functional test (described in Sections 3.2 and 5.1)
3-Fail	If the functional test fails twice, declare fault, disable core, and trap to system software for analysis and recovery
4	Run the ACE firmware (shown in Figure 4)
4-Fail	If ACE firmware results in ERROR, declare fault, disable core, and trap to system software for analysis and recovery
5	Discard old checkpoint, create new checkpoint, context switch back to regular application thread; go to Step 1

3.3 ACE Testing in a Checkpointing and Recovery Environment

We incorporate the ACE testing framework within a multiprocessor checkpointing and recovery mechanism (e.g., SafetyNet [52] or ReVive [42]) 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 5.4.

3.4 Algorithmic Flow of ACE-Based Online Testing

Table 2 shows the flow of ACE-Based Online testing in a checkpointing and recovery environment with single-threaded execution. Other execution models are examined in the next section. Two points are worth noting in the algorithm. First, a lightweight context switch is performed from the application thread to the ACE testing thread at the beginning of the test and vice versa at the end of the test. Lightweight context switching [1], [28] in a single cycle is supported by many simultaneously multithreaded processors today, including Sun's UltraSPARC T1. If lightweight context switch support is not available, then a pipeline flush is required. Our results show that context switch penalty, even if it is hundreds of cycles, only negligibly increases the overhead of ACE testing. Second, if the basic core functional test fails, the core is disabled and execution traps to the system software. If the ACE firmware test fails, the system software performs defect diagnosis to localize the defect. To do so, the system software maps the ACE segments that fail to match the expected test response to specific hardware components (i.e., the combinational logic driving the flip-flops of the ACE segments). If reconfigurability support is provided within those hardware components, the ACE firmware can pinpoint these components to be disabled. Since our focus is on flexible defect detection, we leave fault analysis and recovery to future work.

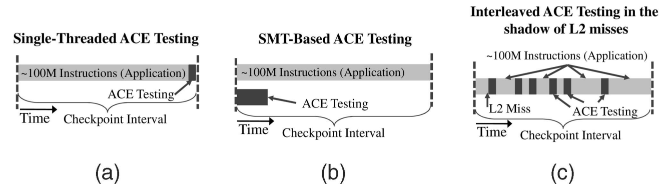


Fig. 5. *Different execution models of ACE testing:* (a) Illustrates ACE testing in a single-threaded sequential execution model, where the ACE testing thread is run exclusively after application execution. (b) The ACE testing thread runs simultaneously with the application in a 2-way SMT execution environment. (c) ACE testing is interleaved with application execution and run in the shadow of L2 cache misses.

3.5 ACE Testing Execution Models

Single-threaded sequential ACE testing. The simplest execution model for ACE testing is to implement the ACE testing process at the end of each checkpoint interval. In this execution model, the application runs normally on the processor until the buffering resources dedicated to the checkpoint are full and a new checkpoint needs to be taken. At this point, a context switch between the application process and the ACE testing process happens. If the ACE testing routine deems the underlying hardware defect free, a new checkpoint of the processor state is taken and the execution of the application process is resumed. Otherwise, system repair and recovery are triggered. Fig. 5a illustrates this single-threaded sequential execution model.

SMT-based ACE testing. In processors that support simultaneous multithreading (SMT) execution [47], [21], [59], it is possible for the ACE firmware to run simultaneously with the application threads running on separate execution contexts. This execution model is illustrated in Fig. 5b and could be higher performance since it overlaps the latency of ACE testing with actual application execution.

Fortunately, the majority of the instructions used by the ACE testing firmware do not entail any synchronization requirements between the ACE testing thread and the other threads running on the processor. For example, the ACE instructions used to load a test pattern into the scan state (`ACE_set`) or read and validate a test response (`ACE_get`) do not affect the execution of other threads running on the processor. The work performed by these instructions can be fully overlapped with application execution.

However, the `ACE_test` instruction momentarily changes the microarchitectural state of the entire processor, and thus, affects the normal execution of all running threads. To avoid the incorrect execution of other running threads when an `ACE_test` instruction is executed by the ACE testing thread, all other threads need to pause execution. This is implemented by using simple synchronization hardware that pauses execution of all other threads (i.e., stalls their pipelines) when an `ACE_test` instruction starts execution and resumes their execution once the test instruction is completed. Note that during testing, the processor's microarchitectural state is stored in the scan state. The microarchitectural state gets restored right after the test cycle (see Section 3.1) enabling the seamless resumption of normal processor execution.

The advantage of the SMT-based ACE testing model is its lower performance overhead compared to single-threaded sequential ACE testing. The disadvantage is that this model requires a separate SMT context to be present in

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

the underlying processor. Note that to guarantee correct recovery, with this execution model, the recovery mechanism needs to buffer the last two checkpoints.

Interleaved ACE testing in the shadow of L2 misses. When the ACE testing thread is sharing the processor resources with other critical applications, it is important to avoid penalizing the performance of these critical applications due to hardware testing. Performance penalties can be reduced by allowing the ACE testing thread to execute only when the processor resources are unutilized by the performance critical threads. An example scenario is to execute the ACE testing thread when the processor is stalled waiting for an L2 cache miss to complete, i.e., in the shadow of L2 cache misses. This execution scenario is illustrated in Fig. 5c.

In the execution model, the processor suspends the execution of the application and context switches into the ACE testing thread when the application incurs an L2 cache miss due to its oldest instruction. The context switch is similar to the lightweight context switches used in switch-on-event multithreading [1], [28]. When the L2 miss is fully serviced, the processor context switches back to the application and suspends the execution of the ACE thread. Under this execution policy, the ACE testing thread utilizes resources that would otherwise be unutilized. However, it is possible that the full ACE testing might not be completed in the shadow of L2 misses because the application might not incur enough L2 cache misses. If that is the case, the remaining portion of the ACE testing thread is executed at the end of the checkpoint interval.

The advantage of the ACE testing model is that it does not require a separate SMT context and can possibly provide lower performance overhead than sequential ACE testing. On the other hand, if L2 misses are not common in an application, the model can degenerate into single-threaded sequential ACE testing. As with the SMT-based model, to guarantee correct recovery with this execution model, the recovery mechanism needs to buffer the last two checkpoints.

4 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 [55], as our experimental testbed.

First, using the processor's RTL code, we divided the processor into ACE domains. We made the 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. Fig. 6 shows the processor modules covered by the ACE framework (note that the L1 caches within each core are also excluded).

Next, we used the Synopsys Design Compiler to synthesize each ACE domain using the Artisan IBM 0.13 μm standard cell library. We used the Synopsys TetraMAX ATPG tool to generate the test patterns.

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

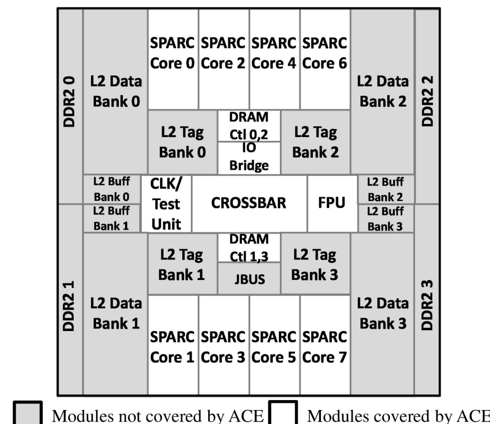


Fig. 6. *ACE coverage of the OpenSPARC T1 processor:* Modules that are dominated by SRAM structures, such as on-chip caches, are not covered by ACE testing since they are already protected by ECC.

circuit defects [36], [17]. In the N-detect test pattern sets, each single stuck-at fault is detected by at least N different test patterns. In addition to the stuck-at and N-detect fault models, we also generate test pattern sets using the path-delay fault model [7]. The path-delay fault model we use is the built-in path-delay fault model in the Synopsys TetraMAX commercial ATPG tool [56].

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 [45] to simulate a SPARC core enhanced with the ACE framework. The simulated SPARC core is a six-stage in-order core (with 16 KB IL1 and 8 KB DL1 caches) running at 1 GHz [55]. For each simulation run, we skipped the first billion instructions and then performed cycle-accurate simulation for different checkpoint interval lengths (10 M, 100 M, and 1 B dynamic instructions). To obtain the number of clock cycles needed for the ACE testing, we simulated a process that was emulating the ACE testing functionality. For the SMT experiments, we use a separate thread that runs the ACE testing software and we use a round-robin thread fetch policy. For these experiments, the simulation terminates when the ACE thread finishes testing and at least one of the other threads executes 100 M instructions. The thread combinations simulated for these experiments were determined randomly. Unless otherwise stated, we evaluate the single-threaded sequential execution model for ACE testing in our experiments.

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 [35]. We wrote a Pin tool that measures the amount of storage needed to buffer the cache lines written back from L2 cache to main memory during a checkpoint interval, based on the ReVive checkpointing scheme [42]. Benchmarks were run to completion for these experiments. Section 5.4 presents the memory logging overhead of our technique.

Performance overhead of I/O-intensive applications. An irreversible I/O operation (e.g., sending a packet to a network interface) requires the termination of a checkpoint before it is executed. If such operations occur frequently, they can lead to consistently short checkpoint intervals, and

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

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

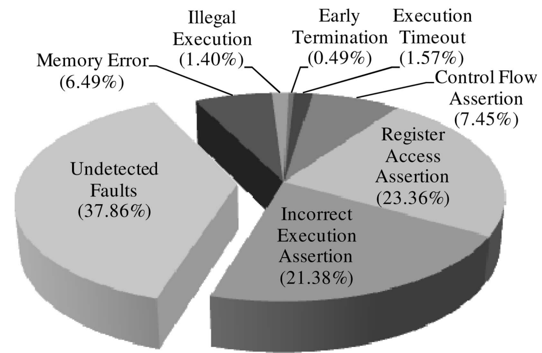


Fig. 7. Fault coverage of basic core functional testing: The pie chart on the right shows the distribution of the outcomes of a fault injection campaign on a five-stage in-order core running the purely software-based preliminary functional tests.

therefore, high performance overhead for our proposal. To investigate the performance overhead due to such frequent I/O operations, we simulated some I/O-intensive filesystem and network processing benchmarks. We evaluated microbenchmarks Bonnie and IOzone to exercise the filesystem by performing frequent disk read/write operations. We also used NetPerf benchmarks [20] to exercise the network interface by performing very frequent packet send/receive operations. In addition to the Netperf suite, we evaluated three other benchmarks, NetIO, NetPIPE, and ttcp, which are commonly used to measure the network performance. In these experiments, the execution of an irrecoverable I/O operation is preceded by a checkpoint termination and the new checkpoint interval begins right after the execution of the I/O operation. Section 5.5 presents our results.

RTL implementation. We implemented the ACE tree structure in RTL using Verilog in order to obtain a detailed and accurate estimate of the area and power consumption overheads 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. Section 5.6 evaluates and quantifies the area overhead of the ACE framework while Section 5.7 evaluates its power consumption.

5 Experimental Evaluation

5.1 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), 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 total size of the software functional test is approximately 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 five-stage in-order core (similar to the SPARC core with the exception of multithreading support). Fig. 7 shows the distribution of the outcomes of the fault injection campaign. Overall, the basic core test successfully detected 62.14 percent of the injected faults. The remaining 37.86 percent of the injected faults lie 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.

5.2 ACE Testing Latency, Coverage, and Storage Requirements

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 4. Table 3 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 Fig. 4).

For the stuck-at fault model, the most demanding module is the SPARC core, requiring about 150 K 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, which 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 percent of the clock period. The required number of test instructions to

TABLE 3
Number of Test Instructions Needed to Test Each of the Major Modules in the Design

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			

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

TABLE 4
Test Pattern/Response Storage Requirements per Fault Model and Design Module

Design Module	Storage Requirements of Test Patterns/Responses (MB)				
	Stuck-at	Path Delay	N-Detect (N=2)	N-Detect (N=4)	All Models
SPARC CPU Core	0.36	0.33	0.56	1.03	2.28
CPU-Cache Crossbar	0.17	0.03	0.30	0.51	1.01
Floating-Point Unit	0.22	0.10	0.30	0.50	1.13
e-Fuse Unit	0.03	0.01	0.08	0.16	0.27
Clock and Test Unit	0.17	0.03	0.32	0.61	1.14
I/O Bridge	0.28	0.10	0.43	0.79	1.60
DRAM Controller	0.59	0.72	0.93	1.44	3.69
Total	1.83	1.34	2.91	5.04	11.11

TABLE 5
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

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	

Each core is assigned to test its resources and some parts of the surrounding noncore modules as shown in this table.

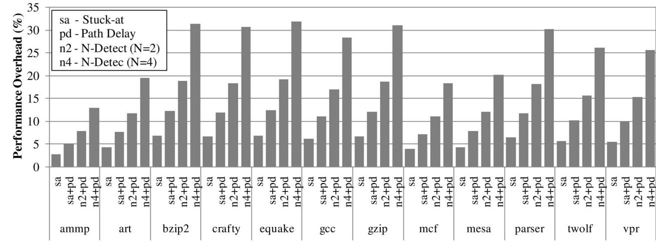


Fig. 8. Performance overhead of ACE testing for a 100 M instruction checkpoint interval.

complete the path-delay tests is usually less than or similar to that required by the stuck-at model.

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 generated increases almost linearly with N, an observation that is aligned with previous studies [36], [17].

Full test coverage. The overall chip test coverage for the stuck-at fault model is 99.22 percent (shown in Table 3). The test coverage for the two considered N-detect fault models is slightly less than that of the stuck-at model, at 98.88 percent and 98.65 percent, respectively, (not shown in Table 3 for simplicity).

Storage requirements for ATPG test patterns/responses. Table 4 shows the storage requirements for the ATPG test patterns and the associated test responses. The storage requirements are shown separately for each major module in the OpenSPARC T1 chip and for each fault model considered in this work. Note that since there is resource replication in the OpenSPARC T1 chip (e.g., there are eight SPARC cores and four DRAM controllers on the chip), only one set of the test patterns/responses is required to be stored per resource. The least amount of test pattern storage is required by the path-delay fault model (1.34 MB) while the most demanding fault model is N-detect, where $N = 4$, which requires about 5 MB. The overall test pattern/response storage requirement for all modules and all fault models is 11.11 MB, which is similar to what is reported in previous work [34]. In our scheme, the test patterns and responses are stored in physical memory and loaded into the register file during the testing phase. Therefore, for physical memories of several gigabytes in modern processors, the storage requirements of 11 MB is considered negligible.

5.3 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 noncore 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 noncore modules.

We distributed the testing responsibilities of the noncore modules to the eight SPARC cores based on the physical location of the modules on the chip (shown in Fig. 6). Table 5 shows the resulting distribution. 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 468 K 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 468 K dynamic test instructions, since all the other cores have shorter test sequences, and will therefore, complete their tests sooner.

5.4 Performance Overhead of ACE Testing

In this section, we evaluate the performance overhead of ACE testing for the execution models described in Section 3.5. For all experiments, we set the checkpoint interval to 100 M instructions.

Single-threaded sequential ACE testing. With this execution model, 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 in the graph of Fig. 8 show the performance overhead when the fault model used in ACE testing is

1. stuck-at,
2. stuck-at and path-delay,
3. N-detect ($\mathrm{N} = 2$) and path-delay, and
4. N-detect ($\mathrm{N} = 4$) and path-delay.

The minimum average performance overhead of ACE testing is 5.5 percent 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 percent. As expected, when test pattern sets are generated using the higher quality N-detect fault model, the average performance overhead increases to 15.2 and 25.4 percent for $\mathrm{N} = 2$ and $\mathrm{N} = 4$, respectively.

Table 6 shows the trade-off between memory logging storage requirements and performance overhead for checkpoint intervals of 10 M, 100 M, and 1 B 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

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

TABLE 6
Memory Log Size and ACE Testing Performance Overhead for Different Checkpoint Intervals

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

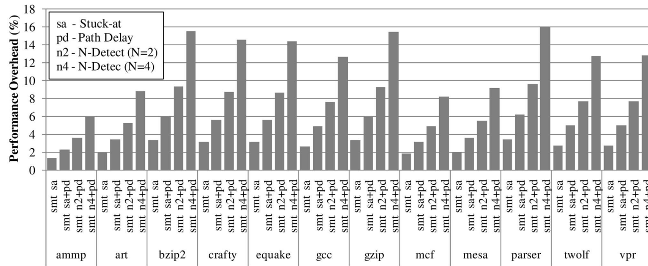


Fig. 9. Performance overhead of SMT-based ACE testing.

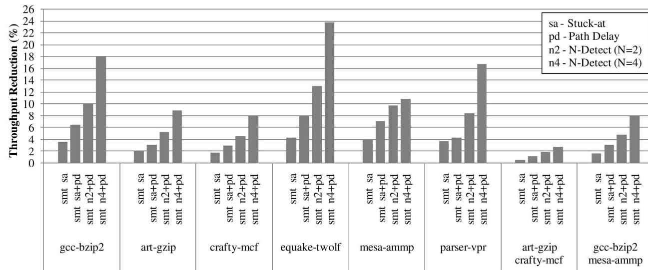


Fig. 10. System throughput reduction due to SMT-based ACE testing.

overhead of ACE testing decreases. From this experiment, 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.

SMT-based ACE testing. Fig. 9 shows the performance overhead when ACE testing is used in a 2-way SMT processor with several SPEC CPU2000 benchmarks. The ACE testing thread runs concurrently, on a separate SMT context, with the benchmark that is evaluated. In this execution model, when ACE testing checks for stuck-at failures, the average performance overhead is 2.6 percent, which is 53 percent lower than the 5.5 percent overhead observed when testing is performed in a single-threaded sequential execution environment. For other fault models, the observed results follow a similar trend: the performance overhead of SMT-based ACE testing is lower than the performance overhead of single-threaded sequential ACE testing. The performance overhead reduction observed under the SMT-based execution model stems from better processor resource utilization between the ACE testing thread and the running application. This is a consequence of the ACE testing thread simultaneously sharing the processor resources instead of sequentially executing exclusively on the processor. The latency of major portions of ACE testing (loading and checking of test patterns) is hidden by application execution.

In SMT-based ACE testing, the testing thread occupies an SMT context. Although performing ACE-based testing in an SMT environment can reduce the potential performance overhead of testing, it is important also to evaluate the system throughput loss due to the testing thread since the extra SMT context utilized by the testing thread could otherwise be utilized by another application thread. Fig. 10 shows the reduction in system throughput when the testing thread competes for processor resources with other threads in a 2-way and a 4-way SMT configuration. We define system throughput as the number of instructions per cycle executed by application threads (excluding the testing thread).

We observe that for stuck-at testing, the system throughput reduction in a 2-way SMT configuration is limited to 3 percent. The highest throughput reduction, 24 percent, is observed in a 2-way SMT configuration when high-quality testing is performed (N-Detect, $N = 4$, in combination with the path-delay fault model). We also observe that when the number of SMT contexts increases to 4, the throughput reduction due to software-based testing significantly reduces. This is because ACE testing occupies only a single thread context in the SMT processor and other thread contexts can still contribute to system throughput by executing application threads.

Interleaved ACE testing in the shadow of L2 misses. Fig. 11 shows the performance overhead when ACE testing is run in the shadow of L2 cache misses. With this execution model, whenever there is an L2 cache miss on the

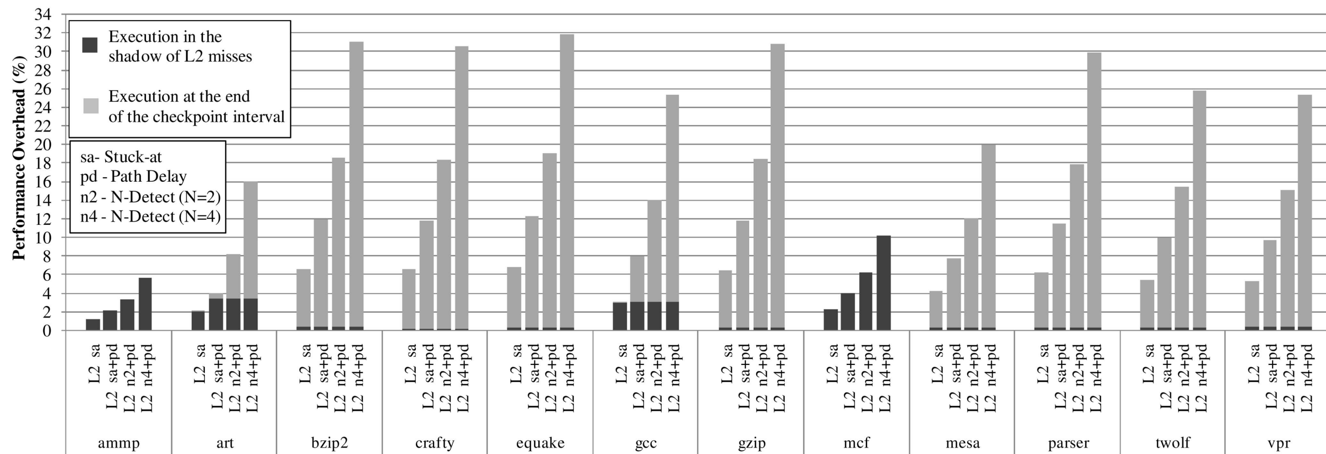


Fig. 11. Performance overhead of interleaved ACE testing in the shadow of L2 cache misses.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

application thread, there is a lightweight context switch with the ACE testing thread. The application thread resumes execution after the L2 cache miss is served. In the case that the checkpoint buffering resources are full (signaling the end of the checkpoint interval) and the ACE testing is not completed, the ACE testing thread starts running exclusively on the processor resources and executes the remaining of the ACE testing routine to completion. The dark part of each bar in Fig. 11 shows the fraction of ACE testing overhead that is due to testing performed in the shadow of L2 cache misses, while the gray part shows the fraction of ACE testing overhead that is due to testing performed at the end of the checkpoint interval. The overhead of testing that is performed in the shadow of L2 cache misses is caused by the additional time taken to switch between the application thread and the ACE testing thread, and vice versa.

We observe that for some memory intensive benchmarks that exhibit a high L2 cache miss rate, such as `ammp` and `mcf`, the ACE testing routine was able to run in its entirety in the shadow of L2 cache misses. For these benchmarks, we observe an average performance overhead reduction of 57 and 43 percent, respectively, compared to single-threaded sequential ACE testing. However, for the rest of the benchmarks, we noticed that due to the low L2 cache miss rate, there were very few opportunities to execute the ACE testing thread in the shadow of L2 cache misses. These benchmarks, depending on the amount of ACE testing performed in the shadow of L2 cache misses, exhibit the same or slightly less performance overhead when compared to single-threaded sequential ACE testing.

Based on these experimental results, we conclude that the interleaved ACE testing execution model benefits only benchmarks that exhibit a high enough L2 cache miss rate and provide enough opportunities for interleaved ACE testing to utilize the processor resources more efficiently. Different thread interleaving criteria other than L2 cache misses could lead to higher benefits and affect more uniformly all benchmarks. However, the overhead of switching between the application thread and the ACE testing thread should be kept low. We leave the design and investigation of such criteria and low overhead context switching to future work.

5.5 Overhead of ACE Testing in I/O-Intensive Applications

In I/O-intensive applications, frequent I/O operations significantly affect the performance overhead of checkpoint-based system rollback and recovery. Several system I/O operations are not reversible (e.g., sending a packet to a network interface, writing to the display, or writing to the disk), and thus, cause early checkpoint termination. Consequently, frequent I/O operations lead to shorter checkpoint intervals and more frequent hardware testing that can have a negative impact on system performance. This section evaluates the performance overhead of ACE testing under a heavy I/O usage environment using I/O-intensive filesystem and network processing benchmarks.

Fig. 12 shows the execution time overhead of ACE testing for the stuck-at fault model and the stuck-at combined with the path-delay fault model. Except for three of the *Netperf* benchmarks, all benchmarks exhibit an execution time overhead that ranges from 4 to 10 percent for the stuck-at fault model and from 6 to 17 percent when combined with the path-delay fault model. Note that the overheads are very high (greater than 25 percent)

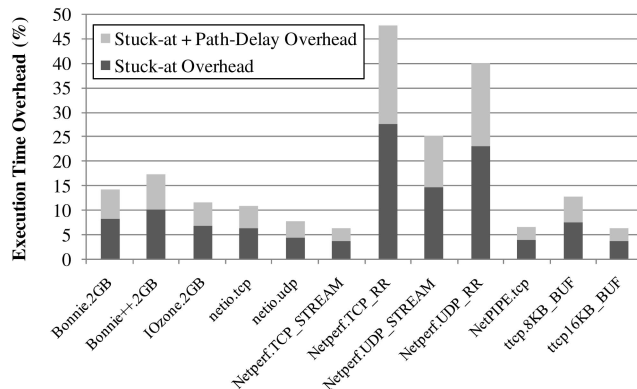


Fig. 12. Execution time overhead of ACE testing on I/O-intensive filesystem and networking applications.

in some *Netperf* benchmarks because these benchmarks are intentionally designed to stress-test the network interface by executing a very tight loop that continuously sends and receives packets to/from the network interface. Even with these adversarial benchmarks, the performance overhead of ACE testing is at most 27 percent with the stuck-at fault model and 48 percent with the combined stuck-at and path-delay fault models.

In this experiment, a checkpoint terminates whenever there is a write operation to the filesystem or a send/receive operation to the network interface (i.e., an irrecoverable I/O operation). This assumption is pessimistic. The execution time overhead observed in this experiment can significantly be reduced with more aggressive and intelligent I/O handling techniques like I/O buffering [39] or I/O speculation [40], which we do not consider in this work. Furthermore, we note that heavily I/O-intensive applications, such as the *Netperf* benchmarks, constitute an unfavorable running environment for the ACE testing technique due to two reasons. First, if high performance is desired when running such I/O intensive applications, the system can alternatively reduce the test quality requirements of ACE testing (or even completely switch it off) and trade-off testing quality with performance. Second, we note that such I/O intensive applications have very low CPU utilization; therefore, there might be little need for high-quality, high-coverage ACE testing of the CPU during their execution.

5.6 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 32 kilobits (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. Fig. 13a shows the topology of the ACE tree configuration, which has the ability to directly access any of the 32 kilobits. 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 the ACE framework configuration (for all eight trees) is 18.7 percent of the chip area.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

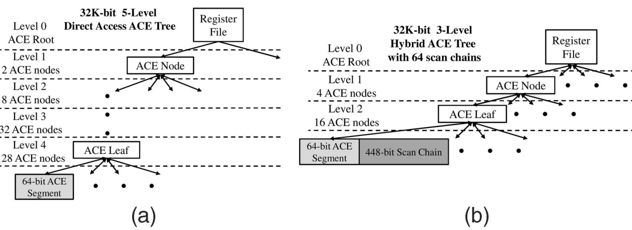
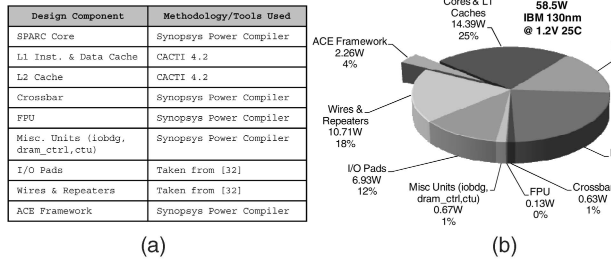


Fig. 13. *ACE tree implementation:* (a) Topology of a direct access ACE tree. (b) Topology of a hybrid (partial direct access, partial scan chain) ACE tree.



Design Component	Methodology/Tools Used
SPARC Core	Synopsys Power Compiler
L1 Inst. & Data Cache	CACTI 4.2
L2 Cache	CACTI 4.2
Crossbar	Synopsys Power Compiler
FPU	Synopsys Power Compiler
Misc. Units (iobdg, dram_ctrl,ctu)	Synopsys Power Compiler
I/O Pads	Taken from [32]
Wires & Repeaters	Taken from [32]
ACE Framework	Synopsys Power Compiler

Fig. 14. *Power consumption overhead of the ACE framework:* (a) All major design components and the methodology/tools used to estimate the associated power consumption. (b) The power envelope of the OpenSPARC T1 design enhanced with the ACE framework.

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 32 K ACE-accessible bits into sixty-four 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. Fig. 13b 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 percent of the chip area.[2]

5.7 Power Consumption Overhead of the ACE Framework

An important consideration in evaluating the ACE framework is the degree to which the extra hardware increases the baseline design's power consumption envelope. To evaluate this power consumption overhead for our design on Sun's OpenSPARC T1 chip multiprocessor, we first estimated the power consumption of the baseline design that lacks the ACE framework capabilities. We calibrated the estimated power consumption with actual power consumption numbers provided by Sun for each module of the chip [32]. After we validated our power estimates for the baseline OpenSPARC T1 design, we estimated the additional power required by the ACE framework.

Power estimation methodology. Fig. 14a shows the major design components of the OpenSPARC T1 and the methodology/tools we used to estimate their power consumption. We estimated the power consumption of the majority of OpenSPARC T1 modules using the Synopsys Power Compiler (part of the Synopsys Design Compiler package) and the available RTL code for the design. Each module's RTL code is synthesized using the Design Compiler. The resulting gate-level netlist is subsequently analyzed by the Power Compiler to estimate the module's power consumption. To perform the synthesis and power consumption analysis, we used the Artisan IBM 130 nm standard cell library, characterized at typical conditions of 1.2 V (Vdd) and 25°C average temperature. The average transistor switching activity factor was set to 0.5.

For modules dominated by SRAM structures, such as the on-chip caches, where logic synthesis and power analysis using the RTL code is inefficient,[3] we used existing tools designed specifically to characterize SRAM modules. To estimate the power consumption of the L1 and L2 caches, we used the CACTI 4.2 tool [57], a tool with integrated cache performance, area, and power models.

This methodology is sufficient enough to estimate the power consumption of most of the chip's logic modules. However, there are parts of the design whose power consumption cannot be accurately estimated with these tools. These include 1) numerous buses, wires, and repeaters distributed all over the design, which are very hard to model accurately using the Design and Power Compilers, unless the design is fully placed and routed, 2) I/O pads of the chip. In order to estimate the power consumption of these two parts, we used values from the reported power envelope of the commercial Sun UltraSPARC T1 design [32].

Results. The estimated power envelope for the whole OpenSPARC T1 chip without the addition of the ACE framework is 56.3 W.[4] Fig. 14b shows the power consumption for our enhanced OpenSPARC T1 design including the ACE framework. The power envelope of the ACE-enhanced design is 58.5 W, where the power consumption of the ACE framework is estimated to be 2.2 W. Thus, the ACE framework consumes 4 percent of the design's total power. Our estimation assumes that the ACE framework is enabled all the time while the chip is in operation. However, as illustrated in the previous sections, the ACE framework is actually used during very short testing periods at the end of each checkpoint interval. Therefore, we expect the actual power consumption and power envelope overhead of the ACE framework to be significantly lower than 4 percent, depending on the frequency and length of testing (i.e., checkpoint interval size and time spent in testing).

6 Other Applications of the ACE Framework

We believe that the ACE framework is a general framework that can be used in several other applications to amortize its hardware cost. We have recently shown that the ACE framework can be utilized for the flexible detection of hardware design bugs during online operation [11]. In this section, we describe how the ACE framework can be used in two other possible applications: post-silicon debugging and manufacturing testing.

6.1 ACE Framework for Post-silicon Debugging

Post-silicon debugging is an essential and highly resource-demanding phase that is on the critical path of the microprocessor development cycle. Following product tape-out (i.e., fabrication of the microprocessor design into a silicon die), the post-silicon debugging phase checks if the

2. We found that the ACE tree's impact on the processor's clock cycle time is negligible in both direct access and hybrid implementations.

3. In logic synthesis, memory elements are synthesized into either latches or flip-flops. Therefore, SRAM macrocells are implemented using memory compilers instead of using the conventional logic synthesis flow.

4. Our estimate of the OpenSPARC T1 power is within 12 percent of the reported power consumption of the commercial Sun Niagara design [32].

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

actual physical design of the product meets all the performance and functionality specifications as they were defined in the design phase. The goal of post-silicon debugging is to find all design errors, also known as design bugs, and to eliminate them through design changes or other means before selling the product to the customer [24], [22], [25].

The first phase of post-silicon debugging is to run extended tests to validate the functional and electrical operation of the design. The validation content commonly consists of focused software test programs written to exercise specific functionalities of the design or randomly generated tests that exercise different parts of the design. We refer to these test programs as the *validation test suite*. These tests are applied under different operating conditions (i.e., voltage, clock frequency, and temperature) in order to electrically characterize the product. When the observed behavior diverges from the expected prespecified correct behavior (i.e., when a failure is found), further investigation is required by the post-silicon debugging team. During a failure investigation, the post-silicon debug engineer tries to 1) isolate the failure, 2) find the root cause of the failure, and 3) fix the failure, using features hardwired into the design to support debugging as well as tools external to the design [24].

Motivation. The trends of higher device integration into a single chip and the high complexity of modern processor designs make the post-silicon debugging phase a significantly costly process, both in terms of resources and time. For modern processors, the post-silicon debugging phase can easily cost $15-20 million and take six months to complete [16]. The post-silicon debugging phase is estimated to take up to 35 percent of the chip design cycle [8], resulting in a lengthy time-to-market. As the level of device integration continues to rise and the complexity of modern processor designs increases [15], this problem will be exacerbated leading to either 1) very expensive and long post-silicon debugging phases, which would adversely affect the processor cost and/or time-to-market or 2) more buggy designs being released to the customers due to poor post-silicon debugging [61], [46], which would likely increase the fraction of chips that fail in the field.

There are two major challenges in the post-silicon debugging of modern highly integrated processors. First, because the internal signals of the microarchitecture have limited observability to the testing software, it is difficult to isolate a failure and find its root cause. Second, because the hardware design is not easily or flexibly alterable by the post-silicon debug engineer, it is difficult to evaluate whether or not a potential fix to the design eliminates the cause of the failure [25]. Existing techniques that are used to address these two challenges are not adequate, as briefly explained below.

Traditional techniques used to address the limited signal observability problem are built-in scan chains [62], [25] and optical probing tools [63]. Unfortunately, both have significant shortcomings. The use of built-in scan chains to monitor internal signals is very slow due to the serial nature of external scan testing [19]. The effectiveness of optical probing tools reduces with each technology generation as direct probing becomes very difficult, if not impossible, with more metal layers and smaller devices [60]. Furthermore, it is very hard to integrate these two techniques into an automated post-silicon debugging environment [60].

The traditional technique used to evaluate design fixes is the Focused Ion Beam (FIB) [24] technique, which temporarily alters the design by physically changing the metal layers of the chip. Unfortunately, FIB is limited in two ways. First, FIB typically can only change metal layers of the chip and cannot create any new transistors. Therefore, some potential design fixes are not possible to make or evaluate using this technology. Second, FIB's effectiveness is projected to diminish with further technology scaling as the access to lower metal layers is becoming increasingly difficult due to the introduction of more metal layers in modern designs [8], [24].

Recently proposed mechanisms try to address the limitations of these traditional techniques. Specifically, recently proposed solutions suggest the use of reconfigurable programmable logic cores and flexible on-chip networks that will improve both signal observability and the ability to temporally alter the design [43]. However, these solutions have considerable area overheads [43] and still do not provide complete accessibility to all of the processor's internal state [43].

Solution—ACE framework for post-silicon debugging. The ACE framework can be an effective low-overhead framework that provides the post-silicon debug engineers with full accessibility and controllability of the processor's internal microarchitectural state at runtime. This capability can be helpful to post-silicon debug engineers in isolating design bugs and finding their root causes. Furthermore, once a design bug is isolated and its causes have been identified, the ACE framework can be used to dynamically overwrite the microarchitectural state, and thus, emulate a potential hardware fix. This allows the debug engineer to quickly observe the effects of a potential design fix and verify its correctness without any physical hardware modification.

Specifically, the event that triggers a failure investigation by a post-silicon debug engineer is an incorrect design output during the execution of the validation test suite. However, by just observing the incorrect output, it is very hard to pinpoint the root cause of the failure. Therefore, further debugging of the failure is required. The first step in this process is the reproduction of the conditions under which the failure occurred. Once the failure is reproduced, debugging tools can be used to analyze the design's internal state and pinpoint the design bug. This is where the ACE firmware could be very useful to a post-silicon debug engineer. The debug engineer can run the ACE firmware as an independent thread (called the ACE debugging thread) that runs in conjunction with the validation test thread to identify the root cause of the failure and evaluate a potential design fix. We first describe the required extensions to the ACE framework to support post-silicon debugging using the ACE firmware, then provide a detailed example of how the debug engineer uses the ACE framework.

ACE instructions for post-silicon debugging. Table 7 shows the ACE instruction set extensions that enable the synchronization between the validation test thread and the ACE debugging thread.

The `ACE_pause` instruction pauses the execution of the running validation test thread after it is executed for a given number of clock cycles and switches execution to the ACE debugging thread. The execution switch between the validation test thread and the ACE debugging thread is scheduled by setting an interrupt counter to the parameter value of the `ACE_pause` instruction. This interrupt counter decrements every clock cycle during the execution of the validation test thread. Once the counter becomes zero, the processor state and scan state get swapped, thus, taking a snapshot of the running microarchitectural state of the

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

TABLE 7
Additional ACE Instruction Set Extensions for Post-silicon Debugging

Post-silicon Debugging ACE Instructions
ACE_pause <# of clock cycles> Pauses the execution of the running validation test thread after it is executed for a given number of clock cycles, and switches execution into the ACE debugging thread.
ACE_return Returns execution from the ACE debugging thread to the validation test thread and swaps the scan state with the processor state in order to restore the microarchitectural state of the validation test thread.

1.Pause Execution Read Processor State

```
ACE_pause 10000
ACE_return
// Continue running the validation
// test for 10000 cycles
...
// 10000 cycles later the processor
// state and scan state are swapped
// and the ACE thread is resumed

// The bug is suspected to be in
// domain#3. Read and print
// domain's state for cycle 10000
for(j=0;j<#_of_ACE_Segments;j++){
  ACE_get $r1, 3, j
  print $r1
}
```

2. Step for one cycle Read Processor State

```
// Set the interrupt counter
// to step for one cycle
ACE_pause 1
// Swap processor state with
// scan state and resume the
// validation test execution
ACE_return

// After one cycle of validation
// test execution the ACE
// debugging thread is resumed

// Read and print domain's
// state for cycle 10001
for(j=0;j<#_of_ACE_Segments;j++)
{
  ACE_get $r1, 3, j
  print $r1
}
```

3. Fix Buggy State Continue Execution

```
// Bug found by debug engineer
// at the state of cycle 10001.
// A control signal should be 0
// instead of 1 in segment#6 bit 3.
// Modify processor state to check
// if bug is fixed.
ACE_get  $r1,3,6
and      $r1,$r1,FFFFFFF7
ACE_set  $r1,3,6

// Run the rest of the
// validation test
ACE_pause 90000
// Swap proc. state with scan state
// and resume execution
ACE_return
...
// At the end of validation test
// check if bug is fixed
```

Fig. 15. Example of ACE firmware pseudocode used for post-silicon debugging.

validation testing thread into the scan state. In the same clock cycle, execution is switched to the ACE debugging thread.

The `ACE_return` instruction returns execution from the ACE debugging thread to the validation testing thread and swaps the scan state with the processor state in order to restore the microarchitectural state of the validation test thread.

Post-silicon debugging example using the ACE framework. Fig. 15 shows example of a possible ACE firmware written to perform post-silicon debugging. Suppose that the debug engineer runs a validation test program that fails after 10,000 cycles of execution and the validation engineer suspects that the bug is in the third ACE domain of the core. Fig. 15 shows the pseudocode of the ACE firmware written to analyze such a failure. The first portion of the code (Fig. 15-left) pauses the execution of the validation test program at the desired clock cycle; the second portion (Fig. 15-middle) allows the debug engineer to single-step the execution by one cycle to observe state changes. Based on the information obtained by running these portions of the code, the engineer devises a possible fix. The third portion of the code (Fig. 15-right) is used by the engineer to evaluate whether or not the design fix would result in correct execution. We describe each code portion of the ACE firmware in detail below.

The debugging process starts with the execution of the ACE debugging firmware thread (Fig. 15-left). In this thread, the first instruction is an `ACE_pause` instruction that sets the interrupt counter to the clock cycle in which detailed debugging is desired by the post-silicon debug engineer. In the example shown in Fig. 15, the validation test is set to be interrupted at clock cycle 10,000 (assuming that this is the phase of the validation test, where the post-silicon debug engineer suspects that the first error occurs). The `ACE_pause` instruction is followed by an `ACE_return` instruction. `ACE_return` switches execution from the ACE debugging thread to the validation test thread, and thus, the validation test program's execution begins.

After 10,000 cycles into the execution of the validation test thread, the validation test thread is interrupted. At this point, 1) processor state is swapped with the scan state and 2) execution is switched from the validation test thread to the ACE debugging thread. Once execution is transferred to the ACE debugging thread, the post-silicon engineer uses the ACE framework to investigate the microarchitectural state of the validation test thread during clock cycle 10,000 (which is stored in the scan state). The example scenario in Fig. 15 assumes that the suspected bug is in the third ACE domain of the core. `ACE_get` instruction reads the third ACE domain's microarchitectural state and prints it to the debugging console. We assume that the domain's microarchitectural state is checked by the debug engineer and is found to be error free. Therefore, the debug engineer decides to check the domain's state in the next clock cycle. In order to step the execution of the validation test thread for one clock cycle, the interrupt counter is set to one using the `ACE_pause` instruction, and the validation test thread's execution is resumed with the execution of the `ACE_return` instruction (Fig. 15-middle).

After one clock cycle of validation test execution, control is transferred again to the ACE debugging thread and the domain's new microarchitectural state is checked by the debug engineer. After inspecting the domain's microarchitectural state, the debug engineer finds that the third bit of the domain's sixth segment is a control signal that should be a zero, but instead, it has the value of one. Thus, the engineer pinpoints the root cause of the failure. In order to verify that this is the only design bug that affects the execution of the validation test thread, and that fixing the specific control signal does not cause any other erroneous side effects, the debug engineer modifies the domain's microarchitectural state and sets the control signal to its correct value using the `ACE_set` instruction (Fig. 15-right). Assuming that the whole validation test takes 100,000 clock cycles to execute, the debug engineer sets the next debugging interrupt to occur after 90,000 clock cycles, which is right after the completion of the validation test. At this point, the execution is transferred to the validation test thread, which runs uninterrupted to completion. After completion, the debug engineer checks the final output to verify that the potential design bug fix led to the correct output and there were not any erroneous side effects due to the introduction of the bug fix. In the case that the final output is incorrect, a new failure investigation starts from the beginning and the debug engineer writes another piece of firmware to investigate the failure.

We would like to note the analogy between ACE framework-based post-silicon debugging and conventional software debugging. `ACE_pause` instruction is analogous to setting a breakpoint in software debugging. `ACE_return` is analogous to the low-level mechanism that allows switching from the debugger to the main program code. Examining the state of the processor and stepping hardware execution for one cycle are analogous to examining the state of program variables and single stepping in software debugging. Finally, ACE framework's ability to modify the state of the processor while the test program is running is analogous to a software debugger's ability to modify memory state during the execution of a software program that is debugged. We note that, similar to a software debugging program, a graphical interface can be designed to encapsulate the post-silicon debugging commands to ease the use of ACE firmware for post-silicon debugging.

Advantages. The results of the detailed debugging process, demonstrated by the above example, are sometimes

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

achievable using traditional post-silicon debugging techniques that were described previously. However, the use of the ACE framework provides a promising post-silicon debugging tool that can ease, shorten, and reduce the cost of the post-silicon design process. The main advantages of ACE framework-based post-silicon debugging are the following:

1. It eases the debugging process: ACE framework-based debugging is closer to software, very similar to the software debugging process, and therefore, is trivial to understand and use by the debug engineer. This ease in debugging is achieved by providing complete accessibility and controllability of the hardware state to the debug engineer.
2. It can test potential design bug fixes without physically and permanently modifying the underlying hardware. This reduces both the cost and difficulty of post-silicon debugging by reducing the manual labor involved in fixing the design bugs.
3. It can accelerate the post-silicon debugging process because it does not require very slow procedures such as scan-out of the whole microarchitectural state or manual modification of the underlying hardware using the aforementioned FIB technique to evaluate potential design fixes.

6.2 ACE Framework for Manufacturing Testing

Manufacturing testing is the phase that follows chip fabrication and screens out parts with defective or weak devices. Today, most complex microprocessor designs use scan chains as the fundamental design for test (DFT) methodology. During the manufacturing testing phase, the design's scan chains are driven by external automatic test equipment (ATE) that applies pregenerated test patterns to check the chip under test [7]. During the manufacturing testing phase, every single chip has to go through this testing process multiple times at different voltage, temperature, and frequency levels. Therefore, the manufacturing testing cost for each chip can be as high as 25-30 percent of the total manufacturing cost [19].

Motivation. Although this testing methodology served the semiconductor industry well for the last few decades, it has started to face an increasing number of challenges due to the exponential increase in the complexity of modern microprocessors [15], a product of the continuous silicon process technology scaling.

Specifically, the external ATE testers have a limited number of channels to drive the design's scan chains due to package pin limitations [19]. Furthermore, the speed of test pattern loading is limited by the maximum scan frequency that is usually much lower than the chip's operating frequency [19], [7]. The limited throughput of the scan interface between the external tester and the design under test constitutes the main bottleneck. These limitations in combination with the larger set of test patterns required for testing modern multimillion gate designs lead to longer time spent on the tester per chip. Even today, the amount of time a chip spends on a tester can be several seconds [19]. Considering that the amortized testing cost of high-end test equipment is estimated to be at thousands of dollars per hour [5], [19], the conventional manufacturing testing process can be very cost-ineffective for microprocessor vendors.

Alternative solutions. Logic BIST is a testing methodology based on pseudorandom test pattern generation and test response compaction. To speed up manufacturing testing, logic BIST techniques use the scan infrastructure to apply the on-chip pseudorandomly generated test patterns and employ specialized hardware to compact the test responses [7]. Furthermore, the control signals used for testing are driven by an on-chip test controller. Therefore, a clear advantage of logic BIST over the traditional manufacturing testing methodology is that it significantly reduces the amount of data that is communicated between the tester and the chip. This leads to shorter testing times and, as a result, lower testing cost. Logic BIST also allows the manufacturing test to be performed at-speed (i.e., at the chip's normal operating frequency rather than the frequency of the automatic test equipment), which improves both the speed and quality of testing.

Although logic BIST addresses major challenges of the traditional manufacturing testing methodology, it also imposes some new challenges. First, logic BIST requires the on-chip storage of a very large amount of pseudorandomly generated test patterns. Second, because logic BIST uses pseudorandomly generated test patterns, it often provides significantly lower fault coverage than that provided by a much smaller number of high-quality, ATPG-pregenerated test patterns [7]. Third, the use of the logic BIST methodology requires significantly more stringent design rules than conventional manufacturing testing [19]. For example, bus conflicts must be eliminated and the circuit must be made random-pattern testable [19]. Therefore, logic BIST techniques significantly increase both the hardware cost and the design complexity, while resulting in lower test coverage.

Proposed solution—use of the ACE framework for manufacturing testing. The ACE infrastructure incorporates the advantages of both the scan-based and logic BIST testing methodologies, while it also can effectively address their limitations. Specifically, the ACE infrastructure provides two capabilities that are not together present in previous manufacturing testing techniques. First, the ACE framework is a built-in solution for fast loading of high-quality pregenerated ATPG test patterns into the scan-chain structures through software. This capability can eliminate the need for expensive and slow external equipment, currently needed for test pattern loading. Second, the ACE framework allows the test patterns to be loaded and applied at-speed at the chip's normal operating frequency rather than the much slower operating frequency of the automatic test equipment, which results in higher quality testing.

With these two capabilities, the ACE framework provides the best of both existing manufacturing testing techniques: 1) fast loading of test patterns to reduce testing time, 2) at-speed testing of the chip to improve testing quality as well as to reduce testing time, and 3) testing with ATPG-pregenerated test patterns rather than the use of pseudorandomly generated test patterns, to improve testing quality. Thus, if employed by the future integrated circuit manufacturing testing methodologies, it can greatly improve the speed, cost, and test coverage of the costly manufacturing testing phase of the microprocessor development cycle.

7 Related Work

Hardware-based reliability techniques. The previous work most closely related to this work is [50]. In [50], we 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.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

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 [7]. Logic BIST incorporates pseudorandom pattern generation and response validation circuitry on the chip. Although on-chip pseudorandom 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 [7].

This 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;
3. It has higher test coverage and shorter testing time because it uses ATPG instead of pseudorandomly generated patterns;
4. In contrast to [50], it can uniformly be 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 noncore modules.

Software-based reliability techniques. A very recent approach proposes the detection of silicon defects by employing low overhead detection strategies that monitor for simple software symptoms at the operating system level [33]. These software-based detection techniques rely on the premise that silicon defects manifested in some microarchitectural structures have a high probability (~ 95 percent) to propagate detectable symptoms through the software stack to the operating system [33].

The main differences between [33] and our work are: 1) unlike the probabilistic software symptom-based defect detection, our technique checks the underlying hardware in a deterministic process through a structured high-quality test methodology with very high fault coverage (99 percent) and can be executed on demand, 2) software symptom-based defect detection techniques can flag the possible existence of a hardware failure, but they do not have the capability to diagnose which part of the underlying hardware is defective. In our technique, by employing ATPG test patterns, it is trivial to diagnose the defective device at a very fine granularity.

Instruction-based functional testing. A large amount of work has been performed in functional testing [6], [26], [31] of microprocessors. The most relevant of these to our approach are the instruction-based functional self-test techniques. In general, these techniques apply randomly generated or automatically selected instruction sequences and/or combinations of instruction sequences and randomly or automatically generated operands to test for hardware defects. If the result of the test sequence does not match the expected output of the instruction sequence, then a hardware fault is declared.

We briefly describe the state-of-the-art approaches that work in this manner: In [58], a self-test program written in processor assembly language and the expected results of the program are stored in on-chip ROM memory. When invoked, the self-test program performs at-speed functional testing of the processor. The proposed scheme requires very little additional hardware cost. It requires an LFSR for generating randomized operands for test instructions and an MISR for generating the result signature. Also, a minor modification of the ISA is required for the test instructions to read/write from the LFSR/MISR. Similarly, Kranitis et al. [29] use the knowledge of the ISA and the RTL-level model of a processor to select high fault coverage instructions and their operands to include in self-test software routines. Batcher and Papachristiou [3] employ instruction randomization hardware to generate randomized instructions to be used in self-test software routines for functional testing. Brahme and Abraham [6] describe how to generate randomized instruction sequences to be used in self-test software routines. Building upon these works, Chen and Dey [9] propose a mechanism that generates instruction sequences to exercise structural test patterns designed to test processor components and applies such instruction sequences in the software-based self-test routines to achieve higher coverage than other approaches that randomly generate instruction sequences.

Our technique is fundamentally different from these instruction-based functional testing techniques in that it is a structural testing approach that uses software routines to apply test patterns. We introduce new instructions that are capable of applying high-quality ATPG-generated structural test patterns to every processor segment by exposing the scan chain to the instruction set architecture. Software self-test routines that use these instructions can therefore directly apply test patterns to processor structures and read test responses, which results in the fast and high-coverage structural testing of each processor component. In contrast, none of the previously proposed instruction-based functional testing techniques are capable of directly applying test patterns to processor components. Instead, they execute existing ISA instruction sequences to indirectly (functionally) test the hardware for faults. As such, previous instruction-based functional test approaches, in general, lead to higher testing times or lower fault coverage since they rely on (randomized) functional testing.

One recent previous work [41] employed purely software-based functional testing techniques during the manufacturing testing of the Intel Pentium 4 processor. 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. In fact, any of the previously proposed instruction-based functional testing approaches can be used as the basic core functional test within the ACE framework.

8 Summary and 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. We described several execution models for the interaction of the special testing firmware with the applications running on the processor for both single-threaded and multithreaded processing cores.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

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 percent of the chip area. Based on our detailed RTL implementation, implementing the ACE framework requires a 5.8 percent increase in Sun Niagara's chip area and a 4 percent increase in its power consumption envelope.

We demonstrated how ACE testing can seamlessly be coupled 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 percent. 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.

We also described how the ACE framework can be used to improve the quality and reduce the cost of two critical phases of microprocessor development: post-silicon debugging and manufacturing testing. Our descriptions showed that the flexibility provided by the ACE framework can significantly ease and accelerate the post-silicon debugging process by making the microarchitecture state easily accessible and controllable by the post-silicon debug engineers. Similarly, the flexibility of the ACE framework can eliminate the need for expensive automatic test equipment or costly yet lower coverage hardware changes (e.g., logic BIST) needed for manufacturing testing. We conclude that the ACE framework is a general framework that can be used for multiple purposes to enhance the reliability and to reduce the design/testing cost of modern microprocessors.

ACKNOWLEDGMENTS

The authors thank the anonymous reviewers for their feedback. This work was supported by grants from the National Science Foundation (NSF), SRC, and GSRC, and is an extended and revised version of [12].

REFERENCES

[1] A. Agarwal, B.-H. Lim, D.A. Kranz, and J. Kubiatowicz, "April: A Processor Architecture for Multiprocessing," *Proc. 17th Ann. Int'l Symp. Computer Architecture (ISCA-17),* 1990.

[2] T.M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *Proc. 32nd Ann. Int'l Symp. Microarchitecture (MICRO-32),* 1999.

[3] K. Batcher and C. Papachristiou, "Instruction Randomization Self Test for Processor Cores," *Proc. Very Large Scale Integration (VLSI) Test Symp. (VTS),* 1999.

[4] S. Borkar, T. Karnik, and V. De, "Design and Reliability Challenges in Nanometer Technologies," *Proc. 41st Ann. Conf. Design Automation (DAC-41),* 2004.

[5] B. Bottoms, "The Third Millennium's Test Dilemma," *IEEE Design and Test of Computers,* vol. 15, no. 4, pp. 7-11, Oct.-Dec. 1998.

[6] D. Brahme and J.A. Abraham, "Functional Testing of Microprocessors," *IEEE Trans. Computers,* vol. 33, no. 6, pp. 475-485, June 1984.

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

[8] K.-H. Chang, I.L. Markov, and V. Bertacco, "Automating Post-Silicon Debugging and Repair," *Proc. Int'l Conf. Computer-Aided Design (ICCAD),* Nov. 2007.

[9] L. Chen and S. Dey, "Software-Based Self-Testing Methodology for Processor Cores," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 3, pp. 369-380, Mar. 2001.

[10] K. Constantinides, J. Blome, S. Plaza, B. Zhang, V. Bertacco, S. Mahlke, T. Austin, and M. Orshansky, "BulletProof: A Defect-Tolerant CMP Switch Architecture," *Proc. 12th Int'l Symp. High Performance Computer Architecture (HPCA-12),* 2006.

[11] K. Constantinides, O. Mutlu, and T. Austin, "Online Design Bug Detection: RTL Analysis, Flexible Mechanisms, and Evaluation," *Proc. 41st Ann. Int'l Symp. Microarchitecture (MICRO-41),* 2008.

[12] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco, "Software-Based Online Detection of Hardware Defects: Mechanisms, Architectural Support, and Evaluation," *Proc. 40th Ann. Int'l Symp. Microarchitecture (MICRO-40),* 2007.

[13] W.J. Dally, L.R. Dennison, D. Harris, K. Kan, and T. Xanthopoulos, "The Reliable Router: A Reliable and High-Performance Communication Substrate for Parallel Computers," *Proc. Parallel Computer Routing and Comm. Workshop (PCRCW),* 1994.

[14] N. Durrant and R. Blish, "Semiconductor Device Reliability Failure Models," http://www.sematech.org/, 2000.

[15] M.J. Flynn and P. Hung, "Microprocessor Design Issues: Thoughts on the Road Ahead," *IEEE Micro,* vol. 25, no. 3, pp. 16-31, May/June 2005.

[16] R. Goering, "Post-Silicon Debugging Worth a Second Look," *Electronic Eng. Times,* Feb. 2007.

[17] 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," *Proc. Very Large Scale Integration (VLSI) Test Symp. (VTS),* 2006.

[18] P. Guptan and A.B. Kahng, "Manufacturing-Aware Physical Design," *Proc. Int'l Conf. Computer-Aided Design (ICCAD),* 2003.

[19] G. Hetherington, T. Fryars, N. Tamarapalli, M. Kassab, A. Hassan, and J. Rajski, "Logic BIST for Large Industrial Designs: Real Issues and Case Studies," *Proc. Int'l Test Conf. (ITC),* Sept. 1999.

[20] *NetPerf: A Network Performance Benchmark.* Hewlett-Packard Company, 1995.

[21] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa, "An Elementary Processor Architecture with Simultaneous Instruction Issuing from Multiple Threads," *Proc. 19th Int'l Symp. Computer Architecture (ISCA-19),* 1992.

[22] H. Holzapfel and P. Levin, "Advanced Post-Silicon Verification and Debug," *EDA Tech Forum,* vol. 3, no. 3, Sept. 2006.

[23] A.M. Ionescu, M.J. Declercq, S. Mahapatra, K. Banerjee, and J. Gautier, "Few Electron Devices: Towards Hybrid CMOS-SET Integrated Circuits," *Proc. Design Automation Conf. (DAC),* 2002.

[24] D. Josephson, "The Good, the Bad, and the Ugly of Silicon Debug," *Proc. 43rd Design Automation Conf. (DAC-43),* pp. 3-6, 2006.

[25] D. Josephson and B. Gottlieb, "The Crazy Mixed up World of Silicon Debug," *Proc. IEEE Custom Integrated Circuits Conf. (IEEE-CICC),* 2004.

[26] H. Klug, "Microprocessor Testing by Instruction Sequences Derived from Random Patterns," *Proc. Int'l Test Conf. (ITC),* 1988.

[27] C. Kong, "A Hardware Overview of the NonStop Himalaya (K10000)," *Tandem Systems Overview,* vol. 10, no. 1, pp. 4-11, 1994.

[28] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded SPARC Processor," *IEEE Micro,* vol. 25, no. 2, pp. 21-29, Mar./Apr. 2005.

[29] N. Kranitis, A. Paschalis, D. Gizopoulos, and Y. Zorian, "Instruction-Based Self-Test of Processor Cores," *Proc. Very Large Scale Integration (VLSI) Test Symp. (VTS),* 2002.

[30] R. Kuppuswamy, P. DesRosier, D. Feltham, R. Sheikh, and P. Thadikaran, "Full Hold-Scan Systems in Microprocessors: Cost/Benefit Analysis," *Intel Technology J.,* vol. 8, no. 1, pp. 63-72, Feb. 2004.

[31] J. Lee and J.H. Patel, "An Instruction Sequence Assembling Methodology for Testing Microprocessors," *Proc. Int'l (r) Test Conf. (ITC),* Sept. 1992.

[32] A.S. Leon, K.W. Tam, J.L. Shin, D. Weisner, and F. Schumacher, "A Power-Efficient High-Throughput 32-Thread SPARC Processor," *IEEE J. Solid-State Circuits,* vol. 42, no. 1, pp. 7-16, Jan. 2007.

[33] M.-L. Li, P. Ramachandran, S.K. Sahoo, S.V. Adve, V.S. Adve, and Y. Zhou, "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design," *Proc. 13th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-XIII),* 2008.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.

[34] Y. Li, S. Makar, and S. Mitra, "CASP: Concurrent Autonomous Chip Self-Test Using Stored Test Patterns," *Proc. Conf. Design, Automation and Test in Europe (DATE),* 2008.
[35] C.-K. Luk, R.S. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V.J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," *Proc. Conf. Programming Language Design and Implementation (PLDI),* 2005.
[36] E.J. McCluskey and C.-W. Tseng, "Stuck-Fault Tests vs. Actual Defects," *Proc. Int'l Test Conf. (ITC),* pp. 336-343, Oct. 2000.
[37] M. Meterelliyoz, H. Mahmoodi, and K. Roy, "A Leakage Control System for Thermal Stability during Burn-In Test," *Proc. Int'l Test Conf. (ITC),* 2005.
[38] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K.S. Kim, "Robust System Design with Built-In Soft-Error Resilience," *Computer,* vol. 38, no. 2, pp. 43-52, Feb. 2005.
[39] J. Nakano, P. Montesinos, K. Gharachorloo, and J. Torrellas, "ReViveI/O: Efficient Handling of I/O in Highly-Available Rollback-Recovery Servers," *Proc. Int'l Symp. High-Performance Computer Architecture (HPCA),* 2006.
[40] E.B. Nightingale, P.M. Chen, and J. Flinn, "Speculative Execution in a Distributed File System," *ACM Trans. Computer Systems,* vol. 24, no. 4, pp. 361-392, Nov. 2006.
[41] P. Parvathala, K. Maneparambil, and W. Lindsay, "FRITS—A Microprocessor Functional BIST Method," *Proc. Int'l Test Conf. (ITC),* 2002.
[42] M. Prvulovic, Z. Zhang, and J. Torrellas, "ReVive: Cost-Effective Architectural Support for Rollback Recovery in Shared-Memory Multiprocessors," *Proc. 29th Int'l Symp. Computer Architecture (ISCA-29),* 2002.
[43] B.R. Quinton and S.J.E. Wilton, "Post-Silicon Debug Using Programmable Logic Cores," *Proc. Conf. Field-Programmable Technology (FPT),* pp. 241-248, 2005.
[44] J.M. Rabaey, *Digital Integrated Circuits: A Design Perspective.* Prentice-Hall, Inc., 1996.
[45] 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.
[46] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, and J. Torrellas, "Patching Processor Design Errors with Programmable Hardware," *IEEE Micro,* vol. 27, no. 1, pp. 12-25, Jan./Feb. 2007.
[47] M.J. Serrano, W. Yamamoto, R.C. Wood, and M. Nemirovsky, "A Model for Performance Estimation in a Multistreamed, Superscalar Processor," *Proc. Seventh Int'l Conf. Modeling Techniques and Tools for Computer Performance Evaluation,* 1994.
[48] P. Shivakumar, S.W. Keckler, C.R. Moore, and D. Burger, "Exploiting Microarchitectural Redundancy for Defect Tolerance," *Proc. Int'l Conf. Computer Design (ICCD),* 2003.
[49] M. Shulz, "The End of the Road for Silicon," *Nature Magazine,* June 1999.
[50] S. Shyam, K. Constantinides, S. Phadke, V. Bertacco, and T. Austin, "Ultra Low-Cost Defect Protection for Microprocessor Pipelines," *Proc. 12th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS-12),* pp. 73-82, 2006.
[51] D.P. Siewiorek and R.S. Swarz, *Reliable Computer Systems: Design and Evaluation,* third ed. AK Peters, Ltd., 1998.
[52] D.J. Sorin, M.M.K. Martin, M.D. Hill, and D.A. Wood, "SafetyNet: Improving the Availability of Shared Memory Multiprocessors with Global Checkpoint/Recovery," *Proc. 29th Int'l Symp. Computer Architecture (ISCA-29),* 2002.
[53] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers, "The Impact of Technology Scaling on Lifetime Reliability," *Proc. Int'l Conf. Dependable Systems and Networks (DSN-34),* 2004.
[54] J.H. Stathis, "Reliability Limits for the Gate Insulator in CMOS Technology," *IBM J. Research and Development,* vol. 46, nos. 2/3, pp. 265-286, 2002.
[55] *OpenSPARC T1 Microarchitecture Specification.* Sun Microsystems, Inc., Aug. 2006.
[56] *TetraMAX ATPG User Guide,* version 2002.05. Synopsys, http://www.synopsys.com, 2002.
[57] D. Tarjan, S. Thoziyoor, and N.P. Jouppi, "Cacti 4.0.," Technical Report hpl-2006-86, Hewlett-Packard, 2006.
[58] M.H. Tehranipour, S. Fakhraie, Z. Navabi, and M. Movahedin, "A Low-Cost At-Speed Bist Architecture for Embedded Processor and Sram Cores," *J. Electronic Testing: Theory and Applications,* vol. 20, no. 2, pp. 155-168, 2004.
[59] D. Tullsen, S. Eggers, and H. Levy, "Simultaneous Multithreading: Maximizing On-Chip Parallelism," *Proc. 22nd Int'l Symp. Computer Architecture (ISCA-22),* June 1995.
[60] D.P. Vallett, "Future Challenges in IC Testing and Fault Isolation," *Proc. IEEE Ann. Meeting of Lasers and Electro-Optics Society (LEOS),* vol. 2, pp. 539-540, Oct. 2003.
[61] I. Wagner, V. Bertacco, and T. Austin, "Shielding against Design Flaws with Field Repairable Control Logic," *Proc. 43rd Design Automation Conf. (DAC-43),* 2006.
[62] T.J. Wood, "The Test and Debug Features of the AMD-K7 Microprocessor," *Proc. Int'l Test Conf. (ITC),* pp. 130-136, 1999.
[63] W.M. Yee, M. Paniccia, T. Eiles, and V. Rao, "Laser Voltage Probe (LVP): A Novel Optical Probing Technology for Flip-Chip Packaged Microprocessors," *Proc. Int'l Symp. Physical and Failure Analysis of Integrated Circuits (IPFA-7),* 1999.

Kypros Constantinides received the BS degree in computer science from the University of Cyprus, in 2004, and the MS degree in electrical engineering and computer science from the University of Michigan, Ann Arbor, in 2006. He is currently working toward the PhD degree in electrical engineering and computer science at the University of Michigan, Ann Arbor. He is interested in computer architecture research with a focus in reliable system design. He previously worked at Microsoft Research and Intel Corporation. He received the Intel Foundation PhD Fellowship in 2008. He is a student member of the IEEE.

Onur Mutlu received the BS degree in computer engineering and psychology from the University of Michigan, Ann Arbor, and the MS and PhD degrees in ECE from the University of Texas at Austin. He is currently an assistant professor of ECE at Carnegie Mellon University. Prior to Carnegie Mellon, he worked at Microsoft Research, Intel Corporation, and Advanced Micro Devices. He is interested in computer architecture and systems research, especially in the interactions between languages, operating systems, compilers, and microarchitecture. He was a recipient of the Intel PhD Fellowship in 2004, the University of Texas George H. Mitchell Award for Excellence in Graduate Research in 2005, the Microsoft Gold Star Award in 2008, and five "Computer Architecture Top Pick" Paper Awards by the *IEEE Micro Magazine*. He is a member of the IEEE.

Todd Austin received the PhD degree in computer science from the University of Wisconsin, Madison, in 1996. He is an associate professor of electrical engineering and computer science at the University of Michigan, Ann Arbor. Prior to joining academia, he was a senior computer architect at Intel's Microprocessor Research Labs, a product-oriented research laboratory in Hillsboro, Oregon. His research interests include computer architecture, compilers, computer system verification, and performance analysis tools and techniques. He is a member of the IEEE.

Valeria Bertacco received the Laurea degree in computer engineering from the University of Padova, Italy, and the MS and PhD degrees in electrical engineering from Stanford University in 2003. She is an assistant professor of electrical engineering and computer science at the University of Michigan. She joined the faculty at Michigan after being at Synopsys for four years. Her research interests are in the areas of formal and semiformal design verification with emphasis on full design validation and digital system reliability. She is an associate editor of the *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* and has served on the program committees for DAC and ICCAD. She is a recipient of the US National Science Foundation (NSF) CAREER Award and the University of Michigan's Outstanding Achievement Award. She is a member of the IEEE.

Authorized licensed use limited to: Carnegie Mellon Libraries. Downloaded on September 11, 2009 at 22:02 from IEEE Xplore. Restrictions apply.