Hardware/Software Codesign Architecture for Online Testing in Chip Multiprocessors

Omer Khan, Member, IEEE, and Sandip Kundu, Fellow, IEEE

Abstract—As the semiconductor industry continues its relentless push for nano-CMOS technologies, long-term device reliability and occurrence of hard errors have emerged as a major concern. Long-term device reliability includes parametric degradation that results in loss of performance as well as hard failures that result in loss of functionality. It has been reported in the ITRS roadmap that effectiveness of traditional burn-in test in product life acceleration is eroding. Thus, to assure sufficient product reliability, fault detection and system reconfiguration must be performed in the field at runtime. Although regular memory structures are protected against hard errors using error-correcting codes, many structures within cores are left unprotected. Several proposed online testing techniques either rely on concurrent testing or periodically check for correctness. These techniques are attractive, but limited due to significant design effort and hardware cost. Furthermore, lack of observability and controllability of microarchitectural states result in long latency, long test sequences, and large storage of golden patterns. In this paper, we propose a low-cost scheme for detecting and debugging hard errors with a fine granularity within cores and keeping the faulty cores functional, with potentially reduced capability and performance. The solution includes both hardware and runtime software based on codesigned virtual machine concept. It has the ability to detect, debug, and isolate hard errors in small noncache array structures, execution units, and combinational logic within cores. Hardware signature registers are used to capture the footprint of execution at the output of functional modules within the cores. A runtime layer of software (microvisor) initiates functional tests concurrently on multiple cores to capture the signature footprints across cores to detect, debug, and isolate hard errors. Results show that using targeted set of functional test sequences, faults can be debugged to a fine-granular level within cores. The hardware cost of the scheme is less than three percent, while the software tasks are performed at a high-level, resulting in a relatively low design effort and cost.

Index Terms—Chip Multiprocessor (CMP), hard error detection, isolation and tolerance, hardware/software codesign.

1 INTRODUCTION

Transistor scaling has enabled integration of an exponentially increasing number of transistors. It is widely believed that Chip Multiprocessors (CMPs) will allow a clear path to ITRS technology scaling projections of 100 billion transistors on a single chip by 2020 [1]. In the area of computing, availability of an ever-increasing number of transistors has generally translated to additional resources. However, due to emerging device reliability and marginality problems, coupled with the lack of exhaustive testing and verification during various phases of design and operation of a chip, the susceptibility of these components to hard errors has also grown. Manufacturers are shipping more parts with incomplete testing than ever before [2], [3]. Silicon debug and diagnosis is now at the forefront of design constraints. A deeper perspective and analysis of silicon debug and diagnosis in context of this paper is presented in Section 2. A system that enables speedy debug and diagnosis in today’s vigorous time-to-market environment is highly desirable.

Fault tolerance techniques are generally categorized into detection/isolation followed by correction/recovery phases.

To enable hard error tolerance, researchers have explored microarchitectural redundancy techniques by disabling defective execution pipelines [4] or redirecting execution to spare or alternate resources, thus avoiding the defective components [5], [6]. Recently, techniques that exploit intercore architectural redundancy [7], [8] have been shown to provide an effective and efficient mechanism for hard error tolerance in multicore systems. The underlying assumption for such mechanisms is the capability to detect, isolate, and debug hard errors within a reasonable granularity for system reconfiguration. Hard errors can occur at manufacturing time or in the field at runtime, making the process of detecting and diagnosing errors a challenging task. Many failures occur after long sequences of operation that can last for minutes or hours. Given that a modern day processor can execute billions of instructions in a second, there is no test environment known to us that can simulate such long sequences of instructions. The key novelty of this paper is the iterative and dynamic nature of debugging repeatable errors in the field without requiring a tester and/or a golden output. Diagnosis and debug objectives have some overlap but the problems are fundamentally different. In case of diagnosis, the patterns that cause failures are known, narrowed to a few, that can be simulated, the reference outputs are known, and the task is to identify the failing portion of the logic. By contrast, in debug only known is that an error happens after a processor runs from seconds to hours. Number of patterns often exceeds billions; naturally they cannot be simulated. Without simulation, reference outputs are not known. The objective is to narrow the

O. Khan is with the Department of Electrical and Computer Engineering, University of Massachusetts, One University Avenue, Lowell, MA 01854. E-mail: Omer_Khan@uml.edu.

S. Kundu is with the Department of Electrical and Computer Engineering, University of Massachusetts, 309J Knowles Engineering Bldg, 151 Holdsworth Way, Amherst, MA 01003. E-mail: kundu@ecs.umass.edu.


For information on obtaining reprints of this article, please send e-mail to: tds@computer.org, and reference IEEECS Log Number TDSC-2009-10-0147. Digital Object Identifier no. 10.1109/TDSC.2011.19.
patterns to a small number where the error is triggered such that the patterns can be diagnosed. In the proposed scheme, after a long sequence of program execution, first we detect an error; then our hardware/software iteration-based scheme invokes and adapts dynamically to the observed responses, thus finding the initial fault appearance down to the correct cycle. This scheme is fundamentally different from the conventional diagnosis problem, where given a pattern and occurrence of fault, the objective is to diagnose down to the failing module.

Traditionally, there are two methods of testing digital circuits: structural (also known as scan) test and functional test. In structural test, logic elements such as latches, flip-flops are converted into scan elements that are reconfigured as scan chains in test mode [9]. Caches, CAMs, and other arrays are structurally isolated with scan collar that provides direct access for testing them in isolation. Recently, researchers have explored scan infrastructure to enable online error detection and diagnosis [10], [11]. Structural test has the advantage of high accessibility into internal state of the design, but the disadvantage is that tests may not be applied at full speed. Scan is well known to excite nonfunctional paths as well, which leads to failures of chips during test that might be good in functional mode, leading to yield loss problem. To circumvent this problem, testing is often conducted at lower frequency to strike a compromise between yield loss and test escapes [12]. Many of the hard errors are only visible when the chip operates at full speed. Thus, scan-based delay tests may not screen many chips with delay-fault problems. In real designs, many of the failing nodes are excited after a multicycle sequence of events. Due to the isolation of array structures during scan testing, multicycle paths through them are not exercised at functional speeds. Finally, augmenting scan patterns with long functional sequences is also problematic. Storing test patterns that exercise billions of instructions require large memory footprint, which may not fit into the tester memory. Converting functional patterns into scan without any loss of coverage is a nontrivial challenge of its own. Due to the persistence of well-known test escapes [13], there is a need to detect and debug hard errors using functional test rather than scan.

In functional test, instructions execute in normal mode. The expected results are compared with known good results to detect failures. Functional test suffers from a number of disadvantages as well. Many failures are intermittently masked in functional test and may propagate, remanifest, and eventually corrupt the program output. Some of these masked failures are logic failures and some of them are performance failures. A thorough classification of error propagation is discussed in [14]. In such cases, without intermediate observation, the fault is masked and debugging for the root cause of the error becomes a painful and sometimes impossible process. If intermediate observation points are established, the amount of observed data grows. As millions or billions of instructions are executed during functional test, the impracticality of observing intermediate data after each computation is self-evident [15], [16].

We propose an online functional test architecture that addresses the following key objectives: 1) Enable testing a multicore in the field at runtime, without a physical tester, 2) Enable observability and controllability for the hard-to-test structures for fine-grain detection and debug, and finally 3) Keep the test content storage overhead, design effort, and hardware costs low. The solution includes both hardware and a runtime software based on codesigned virtual machine concept. Transmeta Crusoe processor [17] demonstrates the practicality of using codesigned virtual machines (VMs). Crusoe uses VM technology for runtime binary translations from conventional x86 ISA to VLIW-based ISA. Unlike Crusoe, we use VM technology for managing concurrent online functional testing of cores in software; functionality orthogonal to binary translation. Our architecture has the ability to detect, debug, and isolate hard errors in hard to test-and-detect components such as large structures including integer multiply and divider, floating-point execution unit, instruction decoders and queues, and small array like structures such as Reorder Buffer and Load Store Queues. These structures are primarily chosen because they are traditionally nonredundant due to their area overhead and require fault tolerance mechanism in case they are faulty. Hardware signature registers are used to capture the footprints of execution at the output of functional modules within the cores. A runtime layer of software (microvisor) initiates functional tests concurrently on multiple cores to capture the signature footprints across cores. The proposed integrated detection and debug scheme is superior to static analysis [18] as it adapts dynamically to the observed response and achieves fine-grain isolation of errors. Results show that using a set of functional test sequences, faults can be debugged down to a fine-granular level without incurring test content storage overheads. The hardware cost of the scheme is less than 3 percent, while the software tasks are performed at a high-level, resulting in a relatively low design effort and cost.

The rest of the paper is organized as follows: In Section 2, we highlight the importance of silicon debug and diagnosis. The proposed architecture for hard error detection and debug is presented in Section 3 followed by a detailed analysis of the microarchitecture components and a case study in Sections 4 and 5. Section 6 presents our experimental methodology and Section 7 discusses the results and analysis of our experiments. We provide some discussion of related work in Section 8 before concluding in Section 9.

2 Motivation

Emergence and detection of hard errors is a continuous process starting from early design phases such as silicon debug through factory testing and lifetime mortality. There is an increasing recognition among the semiconductor industry that verification efforts are encountering difficulties with the growing processor design complexity. It is very hard to test the system with all possible inputs in simulation. During bring-up process, many corner cases are specific to long sequences to stimulate a logic path that leads to the detection of an error. Therefore, many errors may slip through the bring-up phase and show up during operation, causing catastrophic system failures or hangs in some cases. Similarly, during manufacturing test, exhaustive testing is
not possible, mainly due to test costs. Many manufacturers today are shipping parts with incomplete testing [3].

During operation, wear-out related device failure is a growing problem for advanced nanometer technologies [2]. For example, with the advent of 90 nm technology, Negative Bias Temperature Instability (NBTI) has become a major reliability concern [19], where a PMOS device degrades continuously with voltage and temperature stress. The potential for errors decreases the expected lifetime of the processor, resulting in a lifetime reliability problem. Device aging has had a significant impact on transistor performance. Increased current density and temperature leads to faster degradation of transistors over time due to oxide wear-out and hot-carrier degradation effects. Until 90 nm technology, the degradation was small enough to be concealed by an upfront design margin in the product specification. But as the technology approaches 22 nm and below, the worst-case degradation is expected to become too large to be taken as an upfront design margin [2]. Product life acceleration with burn-in test is becoming less meaningful as well. To quote ITRS [1], “Two trends are forcing a dramatic change in the approach and methods for assuring product reliability. First, the gap between normal operating and accelerated test conditions is continuing to narrow, reducing the acceleration factors. Second, increased device complexity is making it impossible or prohibitively expensive to exercise or stimulate the product to obtain sufficient fault coverage in accelerated life tests. As a result, the efficiency and even the ability to meaningfully test reliability at the product level are rapidly diminishing.” Many of these errors may translate to frequent failures during expected operational periods; therefore, making detection, debug, and correction to allow correct operation necessary.

3 Architecture for Online Test

The central component of our hard error detection and debug architecture is microvisor, a layer of implementation-dependent software, codesigned with the hardware. The primary function of microvisor is to manage functional test of structures within cores of a CMP. Of interest is to concurrently run directed test threads on multiple cores and evaluate distributed functional test outcome for hard error detection and debug. Microvisor isolates the faults at a fine granularity for a possible hardware reconfiguration. Researchers have shown that such a fine-grain reconfiguration by keeping the faulty cores functional can be efficiently used for hard error tolerance in performance critical systems [7], [8]. By their very nature, system reconfiguration mechanisms are implementation-dependent and cannot be easily managed by conventional software. The microvisor as envisioned in Fig. 1 resides in a region of physical memory that is concealed from all conventional software. This concealment is desired to tackle against malicious security attacks using the online test framework. Microvisor software builds on the traditional operating system protection modes to run in isolation from user software. This is conceivable as most of the state-of-art hardware platforms provide support for virtualization features like expanded isolation, and mechanisms for quick thread migration [20], [21].

The hardware components are signature registers strategically placed within the cores to capture the execution footprints on the output of structures under test. When a test thread is run on a core, signature registers formulate footprints of execution. These footprints are later used by microvisor to compare against concurrently captured footprints on another core. For debug, microvisor can control the execution footprints at the fine granularity of cycle-by-cycle. Further details on microarchitecture and the use of signature registers to deterministically capture footprints are discussed in Section 4.

The software component is the microvisor software that runs natively as a privileged process on a subset of CMP cores. Microvisor collaborates with the operating system in a time-multiplexed manner, while ensuring that complete isolation of online test is maintained from user software. More details of the execution model of online test architecture are discussed in subsequent sections. The invocation of microvisor is controlled by a dedicated mTimer, as well as special hardware interrupts. mTimer is setup by the microvisor to invoke testing at the granularity of desired debug time window. The debug algorithm starts off by setting the mTimer based on instructions committed per core. If a core is hung for predetermined cycles, it generates an interrupt to invoke microvisor. If a signature mismatches, the algorithm first attempts to identify a known good instruction and subsequently enters a cycle-by-cycle evaluation mode (also based on mTimer) to identify which signature was corrupted first. Examples of other hardware interrupts are built-in-self-test structures to predict occurrence of hard errors [22], and flags to indicate the completion of test thread’s execution. The main feature of microvisor software is to invoke functional test threads.

Fig. 1. Online test architecture system-level framework.
and in case of footprint mismatches, debug and isolate the faults to a targeted set of hardware structures. To accomplish the tasks of running hard error detection and debug procedures, microvisor maintains several data structures to assist with the management details.

### 3.1 Managing Functional Test

A key prerequisite to enable the proposed functional test architecture is to identify a functional core to run the microvisor software; and a subset of cores for testing. System firmware can run traditional online test [15], [16] using built-in self test structures or error detecting codes to perform basic functionality testing at the granularity of cores. At this point, the firmware loads the microvisor into a functional core and identifies the cores for testing. The functional test paradigm is shown in Fig. 2. Microvisor runs on a core identified by \( M \) and the two cores selected for testing are identified using \( C_1 \) and \( C_2 \). Although the proposed test architecture is based on no redundancy for microvisor and dual modular redundancy (DMR) for cores selected for test, the underlying scheme is scalable. For example, two or even three cores can be selected to concurrently run microvisor, and similarly triple modular redundancy (TMR) can be used to select cores for testing [23]. Overheads are low as the management complexity is handled via microvisor software.

Before invoking test threads, microvisor initiates an initialization sequence for each core to flush the machine into a known state [24]. This ensures that the signature registers capture a deterministic footprint of execution. In this work, we assume a deterministic system that synchronizes on a global clock. Nondeterminism in execution is tackled by capturing deterministic state in the signature registers. Details of this mechanism are discussed in further detail in Section 4. Microvisor select test threads within the core and invokes test execution. Details of appropriate test thread selection are presented in the next section. Test execution is terminated by either the completion of the thread or interrupted by the expiration of the mTimer. Thread termination results in a hardware interrupt from the corresponding core to inform the microvisor, whereas, mTimer ensures that each core runs the test thread for a predetermined number of cycles. When all cores report to the microvisor, hard error detection phase is initiated. The captured footprints of structures within cores are accessed by the microvisor, and compared among the equivalent signature registers for the two cores. In case footprint mismatches, hard error debug and isolation routines are initiated. Otherwise, the microvisor selects the next set of threads.

The hard error isolation and debug is divided into two categories. In the first category, when multiple footprints within the core mismatch, then microvisor invokes debug for the time window of the occurrence of the first mismatching footprint. This is accomplished by reinvoking the failing thread iteratively and controlling the termination of thread using the mTimer. Microvisor uses a divide-and-conquer approach to debug the timeline of first mismatching structure. A detailed case study is presented in Section 5 with an example of the proposed algorithm. The second category of debug is to isolate the fault to a specific physical location. This is accomplished by controlling the scope of observability in the faulty structure. In Section 4, we present the detailed microarchitecture for such a debug. The main objective of the proposed debug capability is to isolate a fault for possible hardware reconfiguration around the error-prone circuitry [8].

### 3.2 Tests Thread Selection

An important ingredient of any test framework is the selection of input stimulus to trigger the targeted fault sites for the unit under test. Test vector generation generally falls in two categories: random or pseudorandom test patterns, and targeted or synthetic test patterns. Although random patterns are easy to generate, they may require long test times to trigger a fault site. On the other hand, targeted tests require the knowledge of the design under test. In this paper, we propose a general-purpose scheme to generate functional test threads that can be used to target-specific structures within cores. As wear-out related hard errors emerge at runtime under specific workload and voltage/temperature conditions, a test thread that matches the behavior of actual environment can be helpful for testing related faults. When a test escapee fails in the field after a long sequence of functional patterns that cause failure, error detection, and debug must be done with the failing pattern, not the pattern that was generated before the chip is deployed in the field.

We profiled the expected workload conditions and picked interesting and disparate phases from various workloads to stress the target structures within cores. Such traces are considered a proxy for in-field behavior that results in a bug. This profiling may be performed offline for the expected fault models and workloads, or online using on-chip performance counters that may be readily available [25]. We used SPEC 2000 benchmarks as a proxy for the expected workload conditions [26].

We implemented an offline program phase classifier based on instruction type distributions for the benchmarks using a modified version of the SimPoint toolset [27]. Instruction Type Vectors (ITVs) have been shown to reveal computational demands of threads as they track the execution frequency of instruction types during a phase of the application [8]. We implemented the ITV scheme within the SimPoint toolset and selected 30 test threads of 10 million instructions each. The instruction type distributions and the associated performance (IPC) of each test thread is shown in Fig. 3. These test threads provide a balanced mix of compute bound as well as memory bound threads.

Microvisor uses the instruction distribution of a test thread to match with the targeted structure for testing within the core. For example, if the Load Store Queue (LSQ) is being tested for a possible hard error, test thread with maximum load, and store type instructions is chosen. Similarly to test the Reorder Buffer (ROB), integer ALU dominated thread
with high IPC is chosen. To test floating-point ALU and multiply unit, applu 2 is an excellent candidate thread. We will show in our results section that using such targeted test threads, the number of instructions required to detect a hard error can be reduced.

4 Microarchitecture Support

Fig. 4a shows the scope of structures within a superscalar core that are covered by the proposed hard error detection and debug architecture. Instructions are fetched, decoded, and sent to the instruction queue in the front end of the pipeline. The branch predictor state is also consulted in the front end. The back end of the processor is out-of-order execution, where reorder buffer is responsible for maintaining sequential completion and precise interrupts. The integer and floating-point window issues the instructions for execution, and also keeps track of each instruction’s data flow. The shaded SR blocks represent the signature registers used to capture the execution footprints of the associated structure. For example, the ROB is read out when the instruction at the head of the ROB is committed or an intermediate instruction result is forwarded to the instruction window after resolving data dependencies. In both cases, the output ports of the ROB structure are compressed using spatial and temporal compactor circuits to form a footprint [28]. We assume that large array structures such as multilevel caches and register files are protected by error correcting codes and spare rows and columns [29]; therefore, they are excluded from the proposed architecture.

4.1 Microarchitecture of Signature Register

The key idea behind our signature register is to increase the visibility of internal state of the structures within a core to the external observation port, i.e., the microvisor software. The basic operation is as follows: a set of nodes critical for detection and debug are selected within each structure. Examples of such nodes are outputs of the ALUs or read ports, and data path and control output signals for small array structures such as ROB or LSQ. These signals are then fed into spatial followed by temporal compactor circuits. Space compaction shrinks the number of observed signals and the temporal compaction formulates a footprint over time. Smolens have shown the effectiveness of such compaction design that minimizes aliasing at low hardware overhead cost [28]. Fig. 4b shows our signature register microarchitecture, which is computed concurrently without affecting the functionality of the core. First, the output

![Image of Fig. 4a](image-url)

![Image of Fig. 4b](image-url)

Fig. 4. Scope of microarchitectural observability for online test. (a) Structure-level breakdown of a superscalar core. (b) Signature Register (SR).
signals for observability are compressed using an X-compact error correcting code-based parity tree. Although many parity tree structures are possible, X-compact has been shown to reduce the number of signals dramatically, while preserving all single-bit, double-bit, odd-bit, and contiguous bit errors [30]. The number of X-compact trees for a particular structure within the core is a function of the granularity of debug capability. For example, in case of an ALU, the complete structure is either considered error-free or faulty. Therefore, a single X-compactor is used to compress the output signals. We discuss control mechanism for debug in the next section.

All spatially compressed signals are fed into a Multiple Input Signature Register (MISR) [12]. An MISR is a form of linear feedback shift register that is capable of compacting the observed state of its inputs over time. The key criterion for MISR selection is the width of the registers, which has been shown to be highly sensitive to aliasing. We use a wide 16- to 36-bit MISR, which reduces the probability of aliasing significantly compared to MISR with smaller widths. The output of the MISR at the end of a test sequence is termed as Signature Register and made available to the microvisor via memory-mapped registers. Microvisor software compares the signature registers across execution of a test thread on independent partitions such that each partition is a self-standing and independently usable unit. Ponomarev et al. [31] present details of such partitioning and our online test architecture provides hard error debug and isolation for such structures.

4.2 Controlling the Scope of Observability

The main contribution of the proposed scheme is to debug and isolate faults for possible hardware reconfiguration around the error-prone circuitry. To accomplish this task, the microvisor first isolates the occurrence of first signature register failure. At this point, the structure within core that is error-prone is identified. Shutting off an entire structure for reconfiguration can lead to large performance loss or make the entire core nonfunctional. For example, if the ROB has a hard error, disabling the entire ROB is not possible. It stores the intermediate state of the out-of-order execution engine and, therefore, a central component to keeping the core functional. In such structures, it is desirable to only shut off faulty areas. Structures, such as ROB, LSQ, Instruction Queue, and Instruction Windows can be designed with independent partitions such that each partition is a self-standing and independently usable unit. Ponomarev et al. [31] present details of such partitioning and our online test architecture provides hard error debug and isolation support for these structures.

We extend the signature register microarchitecture by enabling control circuitry to selectively capture the outputs from a structure into the signature register. Fig. 5 shows an example implementation of such selective observation. The microvisor selects a partition within a structure for debug and reruns the failing test thread. The control logic observes the output of the structure and based on the directives from microvisor enables the signature register to capture the appropriate outputs. For example, in the ROB, when the microvisor enables observation of a bank of entries, the control logic uses the read port addresses to select the data being captured in the signature register. By rerunning the test threads and capturing and comparing the signature registers, microvisor enables debug and isolation of hard errors to a fine granular level.

4.3 Avoiding Signature Register Corruption

Although signature register is a very powerful technique, casualness in implementation may lead to corruption of the signature. In this section, we discuss some of the potential issues and present solutions to avoid signature corruption. If all memory structures in the core are not initialized prior to signature computation, the signature is subject to corruption. Therefore, the first condition for the microvisor is to initialize all memory elements to a known state before running test threads.

The second condition is that all internal nodes are required to be valid when sampled by the signature register. There are several potential scenarios that can corrupt the MISR signature. Single-cycle memory access is an important feature in almost all architectures. However, one also desires to have the largest possible memory that can be accessed in a single cycle. An engineering design trick, which is often used in conjunction with memory structures, is to launch a read at the rising edge of a clock, but sample the output at the falling edge of the next clock cycle [32]. This allows a 1.5 cycle memory access. Unfortunately, if the signature register samples the data at the rising edge of the clock, the output of memory structure may not be stable at that point, corrupting the signature. On another front, certain operations simply take longer than others. For example, multiply invariably takes longer than add operation. In an out-of-order superscalar core, a queue is added at the inputs of a functional unit that schedule data every \( n \) cycle. The functional unit produces output every \( n \) clock cycles. The output of the functional unit is read every \( n \) clock cycles. Now, if the output signal is sampled by the signature register every cycle, only one of these cycles is required to produce valid data and other \( n-1 \) cycles are invalid for observation. However, an implementation that makes no such distinction can end up corrupting the signature. Lastly, internal X generation can corrupt the signature. A fully initialized circuit can end up producing Xs during actual operation. An example of such a scenario arises frequently in asynchronous reset operation. Suppose there is a counter that counts up. Counter overflow causes an asynchronous reset signal for some flip-flops. The asynchronous reset can create a temporary X by setting off internal signal contention as illustrated in Fig. 6. Consequently, if signature register is sampling data then it will end up with a corrupt signature.

We present a rule for avoiding corruption of signature register. On an appropriate clock edge, signature register samples data only if the valid signal from the core structure to the control logic of Fig. 5 is set. The valid signal is
asserted true only if the observed data are actually selected by the immediate downstream logic with the following exception: based on the current state information, if it is already known that the result of the successor path will be blocked at selection point downstream, valid signal is not asserted true. This proposed scheme differs from other X-tolerant schemes used in testing [33], in the fact that unlike scan test; vector space for microvisor-based functional testing is not predefined. X masks cannot be prestored as usual in test schemes because the input stimulus belongs to an unbounded set.

5 CASE STUDY: REORDER BUFFER

In this section, we present a detailed mockup case study of hard error detection and debug for an error-prone entry in the ROB module from Fig. 4a. We chose ROB because this noncache array type structure offers no natural redundancy for error tolerance. Additionally, implementing redundancy solely for error tolerance for such structures require significant area overhead with no power or performance benefits to an error-free core. On the other hand, using the proposed signature register, our hardware/software code-sign framework offers an effective mechanism to detect and diagnose any fault in the field at runtime.

For illustration purposes, assume a 128 entry ROB, partitioned into four self-standing and independently controlled banks for debug support. Assuming one entry e0 in the first bank is corrupted. When a test thread is run by the microvisor on this error-prone core, at the time of issue, the core assigns an instruction to this entry. When this instruction finishes execution, the data are written back to it, therefore, effectively getting corrupted at time stamp t1 shown in Fig. 7a. This instruction waits in the ROB for commit, but the data may be forwarded to the integer-window for supplying operands to a younger instruction that is dependent on this instruction. This will effectively corrupt the signature register in integer-window at t2. Subsequently, these faulty data propagate to execution units and will eventually corrupt younger instructions at t3 - t5. In addition, when the original instruction commits, it will corrupt the architecture state of the machine. When the thread finishes, the microvisor compares the signature registers of the error-prone core to the concurrent execution on an error-free core, thus detecting the hard errors in multiple signature registers.

At this point, the microvisor invokes a path-finding debug for the first corrupted signature register. This process is iterative such that the execution of the test thread is repeated from the start until an exit time controlled by microvisor’s mTimer. Figs. 7b and 7c show two such iterations that result in diagnosing the fault site to a single signature register corruption. When the microvisor invokes the debug phase, the error-prone bank in the faulty ROB is identified and subsequently shut off to enable hard error tolerance.

To debug the hard error location to a bank within ROB, four iterations of the test thread are invoked sequentially by the microvisor. The control logic from Fig. 5 is adjusted by the microvisor to capture the appropriate outputs from the bank being tested. This debug phase is visualized in Fig. 8, where a logical view of the selective observation process is shown. Microvisor selects each bank for observing its outputs into the signature register. When the error-prone bank mismatches, the debug phase is complete.

6 METHODS

We used a hybrid simulation framework to evaluate the effectiveness of the proposed hard error detection and debug architecture. For microarchitecture studies such as fault simulations and analysis of area overheads, we created synthesis-ready RTL for the main structures covered by the proposed scheme. To evaluate the architecture of the microvisor-based hard error detection and debug, we modified the SESC cycle-level MIPS simulator [34]. SESC supports multicores within the symmetric multiprocessing paradigm. We evaluated the effectiveness of the test threads and the online test overheads for the proposed error detection and debug scheme.

6.1 Microarchitecture Framework

We created a detailed Verilog-based RTL implementation of the structures covered by the proposed scheme. Table 1 shows the description and features of the core structures used in this paper. We used the Synopsis Design Compiler to synthesize each unit using the TSMC 65 nm standard cell library [35]. The synthesized netlists were subsequently used to run random fault simulations using Synopsis TetraMAX ATPG tool [36]. The goal of random fault simulations was to estimate the probability of fault...
propagation to the signature register for the structure under test. We will show later in this section how such probabilities were used to conduct core-level hard error detection and debug using our architecture-level simulations of the microvisor framework. To conduct random fault simulations for each structure in Table 1, the Verilog netlist was read into the TetraMAX fault simulation tool and a fault list was generated using stuck-at fault and N-detect fault models, which have been shown to be effective for hard failures [37]. In the N-detect test patterns, each stuck-at fault was detected by at least N different patterns. Then, a random set of 10,000 test patterns was generated for each structure and the fault simulation for the target structure was carried out. A set of 1,000 random faults was picked from the previously generated fault list. Fault simulation was a two-step process iterated over 1,000 times for each of the 1,000 random fault, thus resulting in one million samples for each core structure. First, a pattern was randomly picked from the previously generated test patterns. Second, the pattern was fault simulated using Tetra-MAX assuming all primary outputs of the structure were capable of being captured into the signature register. If the fault was detected, it corrupted that unit’s signature register. When a consumer unit read the corrupted unit, the fault was propagated to the associated signature register. This process was repeated for 1,000 iterations for each structure. The goal was to observe the number of instructions needed to identify a fault site in each major block using the proposed architecture.

To perform architecture fault simulations, we assumed a superscalar out-of-order core with the features described in Table 2. For each simulation run, the SPEC 2000 benchmarks were fast-forwarded five to seven billion instructions (depending on the starting instruction for the classified test thread) and cycle-level performance simulations of the test threads from Fig. 3 were performed for 10 million instructions. All signature registers were logged with corruption information and the associated clock cycles. This information was postprocessed to estimate the number of instructions and execution time required by each test thread to detect and diagnose a fault for each structure within the core.

7 Evaluation
In this section, we present the evaluation of our online hard error detection and debug architecture. The key questions we intend to answer are:

1. What is the probability of fault detection for each of the major structures that use the signature register to capture the execution footprints?
2. How long does it take (number of instructions or clock cycles) to detect and fully diagnose a hard error in the covered structures?
3. This is related to question 2. What is a good test thread to detect a fault in a specific core structure?
4. As the main contribution of this hardware/software codesign hard error detection and debug architecture is to deliver fine-granular online test with low design overhead, what is the storage overhead of test threads, how is the system performance impacted as online test is conducted, and the area overhead of the signature registers and associated logic?
7.1 Random Fault Simulations for Synthesized Core Structures

We conducted random fault simulations for each synthesized structure from Table 1. Table 3 shows the NAND2 equivalent gate count and the stuck-at fault list for each structure. We used a 1,000-detect fault model to generate the 10,000 test patterns for the fault simulations. These patterns achieved 80 percent test coverage with 1,000-detect fault model and 99.9 percent coverage with the single stuck-at fault model using the fault list shown in Table 3. The signals observed to flag the success of a fault detection (using the signature register) are also shown as outputs in Table 3.

Fig. 9 shows the probability distribution of successful fault detections using the proposed random fault simulations. This graph identifies easy to detect samples that can be filtered out of the final probabilities to be used. Each bin of probability range shows the percentage of successful detections that fall within a particular probability bound. For example, FPU shows that 25 percent of the successfully detected fault simulations for the 1,000 fault sites are detected within a probability of greater than 0.4. This information is used to filter out the high probability ranges. We eliminated the top 20 percent fault simulations that result in high probability of success error detections. Fig. 9 shows that array type structures generally fall in very low probability range of less than 0.1, which implies that a test thread that stresses such structures is highly desirable. On the other hand, execution units, specifically multipliers and dividers show higher probability of fault detection.

Fig. 10 shows the probability of fault detection for each core structure as an average across 1,000 randomly selected faults, as well as, filtered probabilities that threw away 20 percent of highly probable detections. The results show that ROB has the lowest probability of detection. Using functional threads to test ROB is useful because, architecturally ROB is a highly accessed unit in the core (as it is the bookkeeper-in-chief of the sequential execution model for a modern out-of-order core). The execution units are generally highly structured with most of the logic associated with data path elements. Therefore, the symmetric structure of such units offer high probability of success when a random stimulus is applied for test. We used the probabilities with

<table>
<thead>
<tr>
<th>Core parameters</th>
<th>Architecture Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fetch/Issue/Retire Width</td>
<td>4/4/4 (out-of-order execution)</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>64 KB, 8-way set associativity, 1 cycle, LRU</td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td>64 KB, 8-way set associativity, 2 cycle, Write-back, LRU</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>Shared, 2 MB, 8-way set associativity, 10 cycle, LRU</td>
</tr>
<tr>
<td>Branch Predictor</td>
<td>8K, Hybrid 2-level</td>
</tr>
<tr>
<td>Branch Target Buffer</td>
<td>4K entries, 4-way associativity</td>
</tr>
<tr>
<td>Floating point Units</td>
<td>fpALU: 2 units, 2 cycle; fpMult: 1 unit, 4 cycle; fpDiv: 1unit, 10 cycle</td>
</tr>
<tr>
<td>Integer Units</td>
<td>iALU: 5 units, 1 cycle; iMult: 1 unit, 3 cycle; iDiv: 1 unit, 12 cycle</td>
</tr>
<tr>
<td>Decode delay</td>
<td>2 cycles</td>
</tr>
<tr>
<td>Reorder Buffer</td>
<td>128 entries</td>
</tr>
<tr>
<td>Virtual Registers</td>
<td>Int: 80; FP: 40</td>
</tr>
<tr>
<td>Load Store Queue</td>
<td>32/16 entries</td>
</tr>
<tr>
<td>Off-chip DRAM</td>
<td>200 cycles (100 ns @ 2 GHz)</td>
</tr>
</tbody>
</table>

**TABLE 3**

**RTL Fault Simulations**

<table>
<thead>
<tr>
<th>NAND2 gates</th>
<th>Fault list count</th>
<th>Outputs to Signature Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPU</td>
<td>78,900</td>
<td>218,464</td>
</tr>
<tr>
<td>I-ALU</td>
<td>6,790</td>
<td>22,904</td>
</tr>
<tr>
<td>I-COMPLEX</td>
<td>59,138</td>
<td>155,818</td>
</tr>
<tr>
<td>ROB</td>
<td>233,992</td>
<td>1,057,464</td>
</tr>
<tr>
<td>LSQ</td>
<td>68,929</td>
<td>298,998</td>
</tr>
<tr>
<td>IQ</td>
<td>16,773</td>
<td>79,222</td>
</tr>
<tr>
<td>Decode</td>
<td>10,679</td>
<td>37,634</td>
</tr>
</tbody>
</table>

**TABLE 3**

**RTL Fault Simulations**

<table>
<thead>
<tr>
<th>NAND2 gates</th>
<th>Fault list count</th>
<th>Outputs to Signature Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPU</td>
<td>78,900</td>
<td>218,464</td>
</tr>
<tr>
<td>I-ALU</td>
<td>6,790</td>
<td>22,904</td>
</tr>
<tr>
<td>I-COMPLEX</td>
<td>59,138</td>
<td>155,818</td>
</tr>
<tr>
<td>ROB</td>
<td>233,992</td>
<td>1,057,464</td>
</tr>
<tr>
<td>LSQ</td>
<td>68,929</td>
<td>298,998</td>
</tr>
<tr>
<td>IQ</td>
<td>16,773</td>
<td>79,222</td>
</tr>
<tr>
<td>Decode</td>
<td>10,679</td>
<td>37,634</td>
</tr>
</tbody>
</table>
filtering for high probability samples as our input to the architectural fault simulations of the proposed hard error detection and debug scheme.

7.2 Test Thread Selection and Test Latency

We used architectural fault simulations discussed in Section 6 to determine the efficiency of test threads (from Fig. 3). Not every thread is capable of generating a stimulus that triggers a fault in a structure under test. For example, integer SPEC 2000 threads often do not have any floating-point instructions; therefore, it is useless to run these for detecting a hard error in the FPU. Similarly, to efficiently detect faults in the LSQ, threads with high load/store instruction mixes and low IPC make good candidates. Lower IPC generally implies a memory-bound thread, therefore, stressing the LSQ. For the IQ and ROB, all instruction types are ordered using these structures, specifically threads with high IPC and dominated by ALU-type low-latency operations. All execution units can be stressed by selecting a thread that incurs a high percentage of the associated instruction type mix.

To evaluate the effectiveness of thread selection based on the instruction type mixes, we ran each test thread to detect a randomly selected fault in each of the core structure presented in Table 1. The metric of effectiveness was the number of committed instructions required to detect a fault. We repeated this process for 1,000 iterations by selecting a different fault site for each structure and average the number of committed instructions across all iterations. Fig. 11 shows the average number of instructions required to detect a hard error for each core structure. We plot threads with low, medium, and high criteria (left to right in Fig. 11 for each structure), reflecting a thread’s capability to detect the associated hard error. This selection is based on instruction type mix. A hard error in divider unit is hardest to test, primarily because this instruction is rarely used in a realistic workload. For the rest of the units, our criteria works well, as the number of instructions needed to detect the occurrence of hard error categorically reduce as we pick the thread most appropriate for testing that type of faulty unit. Although LSQ, ROB, and IQ units have very low probability of hard error detection; these units are also the most frequently exercised components in the core. Therefore, using a functional thread that matches the actual program behavior that the machine is originally designed for is a very effective test technique. The exact test contents and the policy of test invocation is left open to implementation. As an example, we propose to selectively choose test threads for testing each covered structure. It is feasible (as seen in Fig. 11) to apply a small subset of tests initially and increase the complexity of test threads as a time-space trade-off. Additionally, considering that the primary debug usage model is to use the long sequence of a functional application that fails in the field, the test thread is the actual application itself and the proposed scheme is capable of handling debug and isolation of errors causing the failure.

We also measured the latency to detect and diagnose hard errors based on the threads selection from Fig. 11. The debug is an iterative process described earlier in Section 4. Fig. 12 shows that using the best candidate thread for each core structure, the latency overhead of online test ranges from 7 to 8,000 microseconds. This overhead is negligible.
In an online test setup, the selection and sequencing of test threads is an important parameter. As shown in Fig. 11, a selective set of test threads can be applied to achieve high probability of fault detection. But, in specific situations, a more comprehensive and targeted set of test threads may be necessary to detect a hard fault. A key question is how to determine storage of the test threads in the memory hierarchy of the system, which subsequently impacts the test time of fault detection at runtime. We measured the instruction and data storage requirements of our test threads at runtime. Fig. 13 presents the runtime storage of threads, where left bar shows that instruction and data storage of 40 megabytes is required to run a selected set of test threads with highest probability of fault detection in all covered structures. On the other end, the right bar shows that all 32 test threads result in 5X increase in storage. The storage requirement is dominated by data, while instructions range from 250 kilobytes to 1 megabyte. The working set of the proposed test threads is small enough to fit in the last-level caches and main memory, thereby, potentially decreasing test time.

7.4 Performance of Online Test Architecture

Online test within a runtime system environment has many parameters that impact the overall system performance relative to test coverage and test time. To evaluate such trade-offs, we developed an analytical model to reason about key runtime parameters for online test. The goal was to be able to conduct testing concurrent to functional mode with a minimum performance impact observed by the end user.

An analytical equation that computes the time to completion of a test thread within the proposed online test framework is presented in (1). Interruption frequency ($\alpha$) represents the interval period at which online test is initiated. Interruption period ($\beta$) accounts for the time overhead associated with a test. Equation (2) shows the breakdown of interruption period with three distinct components: SwitchIn accounts for the time to load the test thread, SwitchOut accounts for unloading the test thread, and TestPeriod is the time dedicated for conducting online test. As the architecture uses time-multiplexing to switch between test and functional mode, the second part of (1) accounts for the number of invocations to complete a test thread. TestPeriodPerThread ($\lambda$) is the time dedicated to running a test thread to completion. TestPeriod is a parameter that is chosen to balance the time to completion and performance overheads of online test.

\[
TimeToCompletion = (\alpha + \beta) \times (\lambda \div TestPeriod),
\]

\[
\beta = SwitchIn + SwitchOut + TestPeriod.
\]

We assumed a 20 GB/second bandwidth for the off-chip memory interface. On-chip memory transfer assumed a 128-bit bus and the multicore running at 3 GHz. SwitchIn of a test thread requires loading the architecture state, memory state, and finally an initialization sequence. Results from Fig. 13 show that on average a test thread does not exceed 4.5 MB of runtime data and instruction space. Assuming this state is stored in off-chip memory at runtime, reading this state on-chip requires 75 $\mu$s using a 20 GB/second transfer rate. Initialiation of the caches assuming a 2.5 MB total cache accounts for 5 $\mu$s. SwitchOut, on the other hand, requires flushing the pipelines, and write-back the architectured state as well as memory state to off-chip memory. This process takes 35 $\mu$s. In summary, interruption period is 115 $\mu$s excluding the TestPeriod, which is considered a variable for tuning online test trade-offs. Each test thread is 10 million instructions in length as discussed in Section 3, which translates to 3,500 $\mu$s TestPeriodPerThread, assuming a single-issue-commit per cycle. Finally, interruption frequency is also considered a configurable parameter for tuning online test trade-offs.

Fig. 14 shows a sweep study of varying TestPeriod from 100 $\mu$s to 2 ms and InterruptionFrequency from two seconds to 30 minutes. The y-axis shows the number of test threads completed per day and the associated performance overhead of online test relative to zero overhead testing (ideal). Note that the y-axis is presented in log scale. It is clear from this Fig. 14 that online test in a concurrent model incurs very small performance overhead (less than 0.1 percent in all cases). If all observed performance losses across the sweep study are acceptable, a 2 ms InterruptionFrequency at five minutes TestPeriod intervals allows the architecture to complete 1,000 test threads every day! This test thread count is sufficient to sustain a high degree of fault detection coverage.

7.5 Area/Power Overheads

One of the main parameters to judge the effectiveness of a test scheme is the hardware area overhead, mainly because the added hardware offers no performance benefit and consumes power. As the proposed scheme in most part utilizes microvisor software to perform detection and debug related tasks, the hardware overhead is the bare-bone
mechanism needed to control and capture the execution footprints. Table 4 shows the area overheads of the proposed scheme. Area of each structure is estimated using synthesized RTL to a NAND2 gate equivalent. We implemented two versions of the RTL, one without the signature registers and the associated control logic and another one with it. It is important to note that the area overhead of the proposed scheme relative to core or CMP area is expected to be much lower as the memory and caching structures are assumed to be protected by error correcting codes. Synthesis results show that the overall area overhead of online test scheme is 2.91 percent. The overhead of decode and instruction queue are larger than the rest, primarily because these units relatively export large number of outputs to the consumer units. One solution to reduce the area overhead of such units is to select smaller set of signals for signature register and use an iterative mechanism to test these units using the microvisor.

For power reduction, the signature registers can be power gated (or at least clock gated) during functional mode using the control logic in Fig. 5. As the test thread is loaded, the signature register power-up can be performed concurrently. As discussed in the previous section, the test threads are only active for a small fraction of time each day. Therefore, the power overheads of online test are minimal.

### TABLE 4
Area Overhead of Signature Register Scheme

<table>
<thead>
<tr>
<th>Structure</th>
<th>NAND2 Comb area</th>
<th>Storage area</th>
<th>Area overhead</th>
<th>Diagnosis success rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>FPU</td>
<td>78,900</td>
<td>48%</td>
<td>52%</td>
<td>2.33%</td>
</tr>
<tr>
<td>I-ALU</td>
<td>6,790</td>
<td>62%</td>
<td>38%</td>
<td>7.54%</td>
</tr>
<tr>
<td>I-COMPLEX</td>
<td>59,138</td>
<td>62%</td>
<td>38%</td>
<td>1.43%</td>
</tr>
<tr>
<td>ROB</td>
<td>233,992</td>
<td>74%</td>
<td>26%</td>
<td>2.51%</td>
</tr>
<tr>
<td>LSQ</td>
<td>68,929</td>
<td>70%</td>
<td>30%</td>
<td>3.42%</td>
</tr>
<tr>
<td>IQ</td>
<td>16,773</td>
<td>81%</td>
<td>19%</td>
<td>6.52%</td>
</tr>
<tr>
<td>Decode</td>
<td>10,679</td>
<td>80%</td>
<td>20%</td>
<td>12.38%</td>
</tr>
<tr>
<td>TOTAL</td>
<td></td>
<td></td>
<td></td>
<td>2.91%</td>
</tr>
</tbody>
</table>

8 RELATED WORK

Hard error detection techniques are generally categorized into postsilicon debug and manufacturing time, or at runtime during field operation. Manufacturing test has the advantage over runtime detection because the tester has access to scan and debug mechanisms, allowing fine-grain visibility into microarchitectural state. Manufacturing test to detect hard errors is a well-understood topic [12], although fault debug and isolation is an active research area. Schuchmann and Vijaykumar [4] and Holdbrook et al. [39] propose altering the microarchitecture to increase visibility of scan. Schuchmann suggests increasing intercycle independence of core structures to isolate faults within two successive pipeline stages. As these techniques are dependent on a physical tester and due to the lack of burn-in [1], they may miss a significant number of faults that manifest only under certain environmental and workload conditions at runtime.

Runtime hard error detection and debug is more challenging as test is performed in the field when no physical tester is attached to the chip. Therefore, a chip must be self-tested. Both structural and functional self-test schemes using hardware and/or software mechanisms have been proposed for microprocessors. Smolens et al. [11] use existing scan chains to detect wear-out faults and rely on operating system to switch between functional and test mode to test the chip in near-marginal conditions. Similarly, Constantinidis et al. [10] propose a virtualization layer between the operating system and the hardware to introduce special instructions for fault detection and isolation. They propose modifying the scan infrastructure to enable structural test using software routines to apply test. Our proposed scheme is fundamentally different from these proposed works in the sense that we apply directed functional test threads to detect, diagnose, and isolate faults in specific structures within the cores. If distributed scan is available, our microvisor runtime software can be extended to provide additional structural testing at the cost of higher test times.

Several software-based and instruction-based functional testing paradigms have also been explored by researchers. Most recently, Apostolakis et al. propose a software-based...
online testing methodology for shared memory multiprocessors [40]. They exploit SMP parallelism to reduce the test execution time via intelligent test program allocation and scheduling. Ramachandran et al. propose the detection of silicon defects by deploying low overhead monitors for simple software symptoms at the operating system level [41]. They rely on the premise that defects manifested in some microarchitectural structures eventually propagate to detectable symptoms in the software stack to the operating system. Our technique is nonprobabilistic and can be executed on demand, essentially providing a detection and debug framework. This software symptoms-based technique does not provide any mechanisms to diagnose and isolate faults. A previous work by Parvathala et al. [15] and Bayraktaroglu et al. [16] employed software-based functional testing techniques during manufacturing test for Intel and Sun processors. Since our test threads are applied in the system environment, unlike FRITS for example, microvisor can make explicit memory references.

Microarchitectural redundancy has also been exploited for error detection in microprocessors. Reis et al. [42] propose using hardware/software-based scheme that achieve fault tolerance by replicating instructions at the compiler level and using hardware fault detectors that use this redundancy. IBM G5 [43] replicates the front end and execution units, and all instructions are executed twice in parallel. By comparing the output of instructions, it detects errors. DIVA [44] uses a simple in-order core as a checker for an out-of-order core. These techniques are mostly costly in hardware and offer very little if any hard error diagnosis capabilities. Bower et al. [45] use DIVA checker and saturating hardware counters for every field deconfigurable unit used by the error-prone instruction to diagnose the fault. This scheme suffers from significant area overhead primarily due to a DIVA checkers and the diagnosis is probabilistic, i.e., the scheme estimates occurrence of hard errors through heuristics based on the usage of different structures. The probabilistic nature of such diagnosis makes it nonexact, nondeterministic, and nonreproducible. Our scheme, on the other hand, offers a low cost (three percent) debug capability without requiring simulations (which becomes impractical for long program sequence to obtain golden reference). As a consequence, our approach is not encumbered by large storage requirements. Li et al.’s proposal [18] suffers from hardware overhead of storing large trace records that blow up when diagnosing hours of program execution. Also, their trace-based diagnosis scheme is static, as the trace is collected and subsequently diagnosed. This results in masking of errors that can hide details when the trace collects information at commit granularity. Our scheme being dynamic does not suffer from such effects as the program execution can rollback and restart at runtime with support to stop at any cycle with precision.

9 Conclusion
We have presented a novel architecture for hard error detection, debug, and isolation in chip multiprocessors based on a hardware/software codesign paradigm. We studied microvisor as stand-alone runtime software that manages functional test and debug of hard-to-test structures within cores. The benefit of such integrated scheme is the adaptation of the test and observation methods. This approach is superior to offline and/or static analysis as it adapts dynamically to observed response to achieve fine-grain debug. Hardware signature registers are shown to provide increased observability and are used to capture the footprint of execution in a controllable manner. We show that by applying a targeted set of functional test threads, faults can be detected and debugged to a fine-granular level within cores. The hardware cost of the proposed scheme is less than three percent, while the test management tasks are performed at a high-level in software. This results in a low-cost online functional test framework that is flexible and scalable for multicore era.

References


