Application-Aware Diagnosis of Runtime Hardware Faults

Andrea Pellegrini and Valeria Bertacco

University of Michigan

{*apellegrini, valeria*}*@umich.edu*

ABSTRACT

Extreme technology scaling in silicon devices drastically affects reliability, particularly because of runtime failures induced by transistor wearout. Currently available online testing mechanisms focus on testing all components in the microprocessor, including hardware modules that have not been exercised, and thus have high performance penalties.

We propose a hybrid hardware/software online testing solution where components that are heavily utilized by the software application are tested more thoroughly and frequently. Thus, our online testing approach focuses on the processor units that most affect application correctness, and it achieves high coverage while incurring minimal performance overhead. We also introduce a new metric, Application-Aware Fault Coverage, representing test's capability to detect faults that might have corrupted the state or the output of an application. Test coverage is further improved through the insertion of observation points that augment the coverage of the testing system. By evaluating our technique on a Sun OpenSPARC T1, we show that our solution maintains high Application-Aware Fault Coverage while reducing the performance overhead required for online testing more than twice compared to previous solutions oblivious to application behavior. In detail, we found that our solution can achieve 95% fault coverage while maintaining a minimal (1.3%) performance overhead and (0.4%) area impact.

1. INTRODUCTION

Continued improvements in semiconductor fabrication technology have enabled the manufacturing of microchips comprised of billions of transistors. While such integration promises major advantages in terms of cost and performance, industry experts have raised concerns that it will come at the expense of transistor robustness [5, 6].

Reliability issues may be triggered by a wide range of causes: manufacturing problems such as optical proximity effects and processing material defects, component infant mortality, and transistor failures at runtime. Among these, runtime failures are the most concerning because they require expensive equipment replacements or, if undetected, may silently corrupt a computation. Several causes may lead to permanent hardware defects at runtime [27] including oxide breakdown [18], hot carrier injection [23], negative bias temperature instability [24], and electromigration [14]. The problem is further exacerbated by aggressive mainstream testing techniques such as burn-in [27], where devices are operated in a high temperature and voltage environment to accelerate the failure of weaker transistors. This reduces the number of system failures due to infant mortality, but impacts all transistors in a device, shortening their expected lifetime.

The impact of transistor failures on microprocessors have been detailed in the literature [10, 26]. Faults in current computer systems heavily affect the reliability of data centers as recently reported by Google [25]. Furthermore, a study from CERN revealed that thousands of files in their data centers where corrupted by faults that escaped hardware error detection and correction mechanisms [19]. In most situations hardware faults are known to cause disruptive software behavior [12] and often corrupt computation without providing any warning signs. Faults that cause silent alterations to the state or to the output of software applications are particularly concerning. For instance, silent data corruptions that lead system outputs to diverge from the expected results may cause financial losses or even have safety impacts. Since programs trust the underlying hardware to correctly execute instructions, software developers rarely handle unexpected events such as hardware faults, even for widely adopted, sensitive applications such as cryptographic routines. However, the effects on these applications can be dramatic when hardware infallibility is questioned, as shown in [21]. As transistors' size decreases with technology, expected fault rates are projected to increase drastically, causing worrisome concerns the correctness of computation by any computer system.

Traditionally, multiple modular redundancy has been adopted for mission critical systems, but the cost of this solution is prohibitive for most commercial applications. More recently, several researchers have proposed a different, more cost-efficient approach that does not rely on computation redundancy but, instead, assumes that the work performed on a processor cannot be trusted until the integrity of the underlying hardware is confirmed. Computations are then partitioned in epochs and normal execution is periodically suspended to execute tests on the microprocessor [8]. Periodic testing of microprocessor's can be accomplished through the addition of ad-hoc hardware testing components [8, 15] and/or through the execution of high-quality software test sequences [7, 13]. Even if effective at detecting faults, the execution of online tests is a time consuming task and results in a performance reduction up to 30% [7]. Independently from their implementation, all current online testing techniques focus on maximizing the portion of the silicon area where faults can be detected, striving to provide high fault coverage throughout the entire device.

This work proposes a novel approach on online testing of microprocessors, emphasizing application sensitivity to runtime hardware faults.

1.1 Contributions

This work provides an adaptive fault detection framework for periodic on-line testing with high coverage, low performance overhead, and near-zero area cost. Our solution can diagnose permanent faults in microprocessors at runtime. We make the following contributions:

- We propose a framework that **dynamically tunes a test to focus on the hardware units that a software application exercised**. As a result, we can limit testing overhead without compromising test quality. Since our technique tunes tests to the application, it provides close and careful monitoring of units that, if faulty, might have corrupted the state or the outputs of the software.

- We rely on a **hybrid hardware/software solution** to deliver this online approach. Software tests are run to check the integrity of the underlying hardware; the fault coverage provided by the tests is improved through the insertion of hardware observation points used to collect additional information on the outcome of the tests. The hardware integrity is checked through the execution of software routines, enabling high fault coverage without any hardware support. Extra

observation nodes are inserted in the microprocessor logic and yield a significant increase in test coverage for hard-to-observe components at the price of very little area overhead,

- We propose a **new metric called *Application-Aware Fault Coverage* (AFC)** to measure the quality of a test with regards to the dynamic usage of the underlying hardware. Most software applications only leverage a portion of the system's hardware components for their execution, so it makes sense for online test solutions to focus only on these relevant units. AFC takes into account the dynamic usage of hardware units, thus evaluating the effective fault protection that the user experiences.

We evaluated test quality, performance overhead, and area impact of our diagnosis mechanism on a microprocessor based on a OpenSPARC T1 core [28]. We found that the area overhead of our design is negligible (approximately 0.8%), and can provide very high Application-Aware Fault Coverage (95.5%) with extremely low runtime overhead (only 1.3% slowdown). Additionally, to the best of our knowledge, we are the first to analyze and detail the fault coverage achievable by software testing on a multi-threaded microprocessor.

In the reminder of the paper, we first introduce our proposed dynamic coverage metric, called *Application-Aware Fault Coverage* (AFC). We then overview the architecture of our testing framework and discuss in detail its dynamic software test selection and hardware mechanisms. We evaluate the framework for its fault protection capabilities, performance and area impact. Then we compare our solution against previous works and conclude discussing future research directions.

2. RELATED WORK

Classic runtime testing techniques focus on achieving a high overall area fault coverage on the microprocessor, regardless of its utilization. In these solutions a constant runtime overhead is incurred to apply the tests at regular intervals.

Hardware-based detection mechanisms insert extra microarchitectural features in order to detect faulty transistors on the chip. For example, Costantinides, *et al.* , [8] and Mehrara, *et al.* , [15] propose embedding Built-In-Self-Test units and checkers to test the integrity of VLIW microprocessors. These hardware additions account for a significant area overhead, 5.8% and 14%, respectively, and provide limited coverage against permanent faults (89% and 95%). **Structural testing** has recently been proposed as a viable way to perform runtime fault detection [7, 13]. Structural testing is able to achieve the highest fault coverage in digital systems. However, its area and performance overhead limits its viability for online reliability schemes. Structural testing requires significant area overhead, approximately 5%, to implement the logic necessary to dynamically load and unload scan chains in the microprocessor. Processor down time for these testing mechanisms can be extremely high, up to few seconds for the solution proposed in [13]. **Software-based** hardware integrity checks utilize software routines achieving high fault coverage with no hardware additions [4, 22]. Their focus is to achieve an elevated fault coverage across all structures of the CPU, which poses a significant test overhead, regardless of the dynamic usage and health of the processor. However, our hybrid solution takes advantage of the low cost of software testing, but improves its test quality through the addition of extra observation points. Gupta, *et al.* [9] suggested tuning the execution of software-based hardware routines based on the health of the silicon elements within the processor. Hardware health is estimated only through in-situ oxide breakdown and NBTI sensors spread throughout the silicon, occupying an overall area of about 2.6% [11]. Based on the information collected from the sensors, their solution strives to test weaker units for longer periods of time. Gupta, *et al.* focus only on a single type silicon defect: extending their approach to handle other sources of silicon degradation requires the development and the deployment of specialized sensors.

All these solutions are oblivious to application behavior; our hybrid solution, in contrast, proposes to monitor the utilization of processor's structures. By doing so, we can tune our tests to specifically target those structures that have been subjected to high activity, and thus have a higher risk of corrupting the application. As a result, we can deliver high confidence that faults that corrupt the application will be detected.

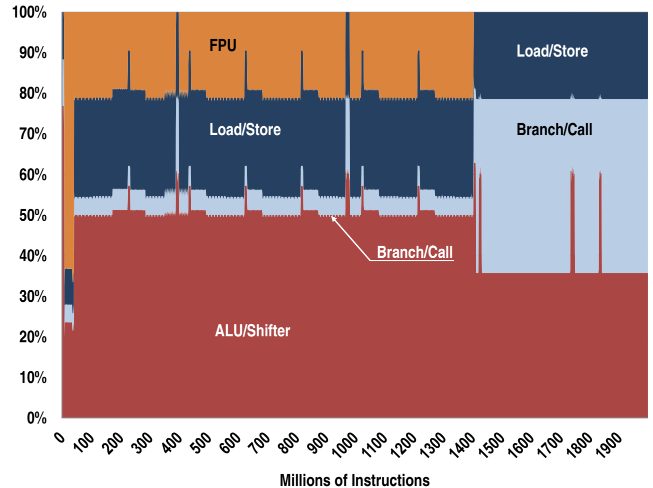

Figure 1: Dynamic instructions in the NAS FT benchmark. This application solves differential equations using Fourier transforms. The figure shows the type of dynamic instructions executed over a window of of 2 billion instructions and their distribution by type. The goal of our adaptive diagnosis mechanism is to test only those units that have been activated during the software execution interval preceding the test.

3. APPLICATION-AWARE FAULT COVERAGE

The quality of fault detection tests has traditionally been measured by the fraction of transistors in the systems for which a failure would be detected by a given test.

However, we note that the usage of different functional units by a software application varies greatly during execution. Consequently, the fault locations that might corrupt the state or the outputs of an application tend to change over time. For example, Figure 1 shows the type of dynamic instructions executed over a window of 2 billion instructions by a scientific benchmark application, Fast Fourier Transform from the NAS suite. Note that the execution is characterized by long phases where instructions requiring only a portion of the hardware units are being executed. These patterns of utilization are not unique to this benchmark, but are common to most applications.

Researchers have shown that applications are only susceptible to faults that occur in the hardware units contributing to the computation of the program's outcome [12, 16]. Thus, since software leverages different components at different times of the execution, the hardware units for which fault detection is relevant, for the sake of correct software computation, also change over time.

This metric is inspired by software testing techniques such as operational profile [17] and PathScore [2]. In these works, the code is profiled at design time and the software functions that are often executed by users are determined. Testing and debugging is then dictated by the results of these analyses such that more thorough tests are performed on the portions of the code that are highly stressed by a users' population. This approach is very powerful and widely adopted in the software industry due to its high benefit-to-cost ratio. Online hardware testing, on the other hand, is typically ignorant of customers's usage of the hardware components and thus targets high area coverage across the entire system.

To evaluate fault coverage in the context of an application's dynamic behavior, we introduce a new metric, called *Application-Aware Fault Coverage* - AFC. AFC measures the quality of a test with respect to its ability to detect a fault in hardware units that the application exercised. For instance, if an application only uses the integer pipeline of the processor, a test that detects faults occurring exclusively in the floating point unit (FPU) would provide an AFC of 0%. If an application were to use the FPU during half of its execution cycles, a test that exclusively provides 80% coverage over the FPU unit's transistors would have an AFC of 40%. In other words, instead of measuring the fraction of transistors that a test covers, AFC measures the likelihood of detecting a hardware fault that might have corrupted the software computation.

In the presentation below, we first analyze the relevance of a failure manifesting at the transistor level, the hardware unit level, and the chip level. We then consider the area fault coverage provided by a given test to define the AFC metric.

Let P_f represent the probability of a single transistor failing when it switches. Then the probability of that transistor not failing is $1 - P_f$. We call the number of switching events between two testing intervals s, so the probability of a transistor not failing after s switching events is $(1 - P_f)^s$, and the probability that the same transistor fails within s switching events is $1 - (1 - P_f)^s$.

Consider then a hardware unit comprising n transistors and, for sake of simplicity, assume that they are all subject to the same switching activity. Assuming that the probabilies of two transistors failing are independent from each other, then the probability that at least one error has occurred in the unit after s switching activities is $1 - (1 - P_f)^{sn}$. By applying the Taylor binomial expansion to this expression, we obtain:

$$1 - (1 - P_f)^{sn} = snP_f + \binom{sn}{2} P_f^2 - \cdots$$

If P_f is negligible when compared to s and n, the expression above can be approximated with snP_f. This is the case in all practical situations because the probability of a transistor failure due to a single switching event is extremely low. For example, if a 5GHz processor composed of 10 billion transistors has a Mean Time Between Failures of one day, and a hardware test is triggered every second, the value $1/P_f$ is 5 orders of magnitude greater than the product sn. For a conservative assumption, we consider that all transistors switch as often as the one that switches the most.

At the chip level, a processor is composed of i units. We assume a negligible probability of having two faults manifesting in the processor in the time frame delimited by two subsequent tests. Then the total probability of a chip incurring in a fault is given by the sum of the individual probabilities that any of its modules incurred in a fault:

$$P(chip_fails) \approx \sum_i s_i n_i P_f$$

Finally, we must take into account the ability of a test to detect these faults. Assuming that a test covers a fraction c_i of the transistors in each module i, the probability that it can detect a fault is: $\sum_i (s_i n_i c_i P_f)$. We therefore define *Application-Aware Fault Coverage* of the test to be the ratio between faults that can be detected by the test and all possible occurring faults. Because P_f is common to all terms, it can be removed from the expression, yielding:

$$AFC = \frac{\sum_i (s_i n_i c_i)}{\sum_i (s_i n_i)}$$

Note that our AFC metric takes into account usage of hardware modules due to a particular workload. For example, assume that a simple processor consists of only two modules that occupy the same area on the chip: an integer pipeline and a floating point unit; and consider two tests: one achieves 90% fault coverage over the entire processor area, while the other provides 95% coverage for the integer pipeline and 65% for the floating point unit. If we compare the two tests in terms of fault coverage of the silicon area, the latter provides significantly lower coverage (80%) than the former. However, if an application does not utilize the floating point unit, we attain much better protection from the second test. Taking the dynamic behavior into account, our AFC metric reports a 90% coverage for the first test, against a 95% for the second. We use the AFC metric in this work to evaluate the effectiveness of tests in protecting a system against failures that can affect a software application running on the system.

4. APPLICATION-AWARE DIAGNOSIS

Our diagnosis framework takes advantage of dynamic program behavior to reduce the overhead required for periodic testing without affecting AFC. Classic online testing technologies invest significant effort to thoroughly test all components of a processor. In contrast we propose to constantly monitor the activity of all functional units of the CPU and test only those contributing to the outcome of the user's application. With reference to Figure 1, note how the FPU is used steadily in the first part of the benchmark's execution, while the last portion only exercises the integer pipeline. Correspondingly, a test that optimizes performance without affecting AFC would invest time to check the FPU unit during the first part of the benchmark's execution, but would only focus on the integer pipeline in the last portion.

Application utilization of the hardware is assessed through hardware counters, called *activity monitors*. An activity monitor is associated to each functional unit in the processor. Every time an instruction exercises a particular functional unit the associated counter is incremented. To characterize the dynamic behavior of the application the activity monitors are reset at the beginning of every epoch. At the end of the epoch, our proposed framework evaluates the values of the activity monitors to decide which hardware needs to be tested. To optimize the overhead imposed by our detection mechanism, we develop a fully-adaptive framework, so that unit-focused tests are triggered on demand. Specifically, during each testing phase, we execute several test routines, each exercising a specific unit that was utilized during the last execution interval of the software application. Units which did not experience utilization, based on the information from the activity monitors, are not tested since they would not improve the test's AFC.

This approach is beneficial for two reasons. First, the units that have been exercised by the application, and might have corrupted it if faulty, are closely monitored. Second, test length is reduced by skipping tests of unused components, thus improving user's experience. Indeed, a fault occurring in one of those units would not affect the correctness of the computation of the software application running in the system.

To further boost fault coverage in hard-to-test units we increase design observability by adding dedicated *observation points*. Enhancing the system with observation points does not impact perfomance and requires very limited hardware additions. Since data from the observation points is collected only during the testing phase, they are transparent to software applications.

With our integrated approach, we can expose the vast majority of microprocessor's faults and, in particular, the ones an application is most sensitive to, without incurring the high cost of traditional testing mechanisms such as BIST and scan-chains.

4.1 Software Tests

In the hardware testing community, it is recognized that software-based fault testing can be a very effective way to expose the majority of faults in a processor design [22]. In our framework, we chose to use the software regression suite developed for the functional verification of the processor under study, since this software strives to check all, or most, corner cases of the system's behavior.

From this suite, we want to select several test subsets, one for each hardware unit. Each subset should comprise the most effective tests in detecting faults for a given unit. We accomplish this goal by formulating an integer linear programming (ILP) problem, such that the solution of this problem provides the set of tests we are seeking. To start this process, we partition the processor into several functional units and create an ILP problem for each of them. For instance, for the processor considered in our experimental evaluation, we partitioned the design in five separate units: integer pipeline, divider, multiplier, floating point frontend, and stream processing unit.

A fault coverage matrix is built from the outcome of the software tests. Coefficients in the matrix specify which fault locations are exposed by each test (Figure 2.a). From the fault coverage matrix, the constraints for the ILP problem are generated (Figure 2.b). A binary variable is associated with every test (t_i) and fault location (f_j) and one inequality is added for each possible fault location. The binary variable that represents a test i, t_i, is set to one if and only if the associated test is selected for execution. A variable associated with a fault location j, f_j, is greater than zero only if the fault is exposed by at least one of the test that will be executed.

Since the integer pipeline is active all the time while the system is operational, the corresponding test is selected for each testing session. As a result, this test has the most impact on test execution time. Therefore, we set a hard constraint on its time budget. In contrast, for all other units, test time is less critical since they are triggered only occasionally. For those, high coverage becomes the most relevant parameter. Below we present the specific aspects of the two problem setups.

Integer pipeline test. For this test, we add a hard constraint to the ILP problem instance so that the total execution time of the tests selected is below a pre-determined threshold. This choice is driven by the frequent use of this test, and its consequent high impact on overall performance. In addition, the objective function of the ILP instance is to maximize fault coverage, that is, to maximize the number of distinct faults covered by the execution of the tests (Figure 2.c).

Module directed tests. For the other functional units, the primary goal is to achieve high coverage. A specific modular test is developed for each complex module in the microprocessor not already covered by the integer pipeline test. The ILP problem setup is similar; however we do not set a hard constraint on the execution time of the test suite. Instead, we add a constraint requiring that coverage is above a specified threshold (Figure 2.c).

The ILP problem for the integer pipeline in a complex processor

Fault locations

Tests	F_0	F_1	F_2	F_3	F_4	F_5	Cost
T_0	0	1	1	0	1	0	c_0
T_1	1	1	0	0	0	1	c_1
T_2	1	0	1	1	0	0	c_2
T_3	0	0	0	0	1	0	c_3
T_4	0	0	0	1	1	1	c_4

a)

$$t_i = \begin{cases} 1, \text{ if } T_i\ (0 \le i \le 4) \text{ is selected} \\ 0, \text{ otherwise} \end{cases}$$

$$f_j = \begin{cases} 1, \text{ if one of more tests exposes } F_j\ (0 \le j \le 5) \\ 0, \text{ otherwise} \end{cases}$$

Constraint inequalities:

$t_1 + t_2 \ge f_0$
$t_0 + t_1 \ge f_1$
$t_0 + t_2 \ge f_2$
$t_2 + t_4 \ge f_3$
$t_0 + t_3 + t_4 \ge f_4$
$t_1 + t_4 \ge f_5$

b)

	Additional constraint	**Goal**
Integer Pipeline	$\sum_i t_i c_i \le$ test time budget	Max($\sum_j f_j$)
Module Directed	$\sum_j f_j \ge$ target fault coverage	Min($\sum_i t_i c_i$)

c)

Figure 2: Formulation of the ILP problems to select the subset of tests to execute at runtime. a. Example fault coverage matrix is build: a non-zero coefficient at location (i, j) indicates that the i-th test exposes the j-th fault. The last column indicates cost in execution cycles of a test. **b.** Constraints derived from the fault coverage matrix. **c.** Additional constraints and goals for the two ILP problems.

such as the OpenSPARC T1 consists of more than 300,000 fault locations, over 850 tests, and occupies more than 2GB of memory when stored in a file system. A commercial ILP solver spends between 2 and 40 hours to find a solution to the problem and peaks at 30GB of memory usage. The solution of the ILP problems must only be computed once when the microprocessor is designed.

4.2 Hardware Activity Monitors

To track switching activity in the various units we utilize activity monitors. These consist of counters associated with each complex unit in the microprocessor's architecture. Each activity monitor oversees a processor's functional module, and a counter is incremented every time the corresponding module is subject to switching activity. The counters are reset after each hardware integrity check (testing phase). In practice, module utilization can be approximated by analyzing the instruction flow: in our solution, we use a dedicated controller, which observes each instruction entering the processor's decoder stage and increments appropriate counters based on which units a given instruction exercises. The activity monitors are embedded in the processor's hardware, as shown in Figure 3. We envision that software routines evaluating the need of triggering a unit test can access their value.

Functional unit testing can be triggered when the functional unit's utilization rises above a preset threshold. In our framework, we envision that users, or the operating system could configure the desired trigger thresholds dynamically, in order to trade-off performance overhead with AFC.

4.3 Microprocessor Observability Extensions

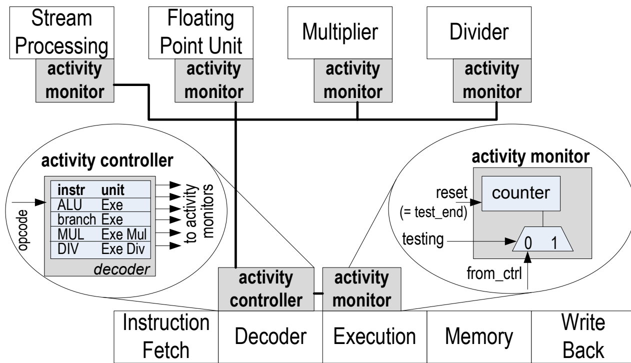

Figure 3: Activity monitors track the use of each processor's units so that tests can be adapted to target those activated during the last execution interval, and could thus affect the correctness of the computation if faulty. Monitors' counters are incremented by a controller based on the instruction flow at the decoder stage of the processor, and are reset at the beginning of each execution interval.

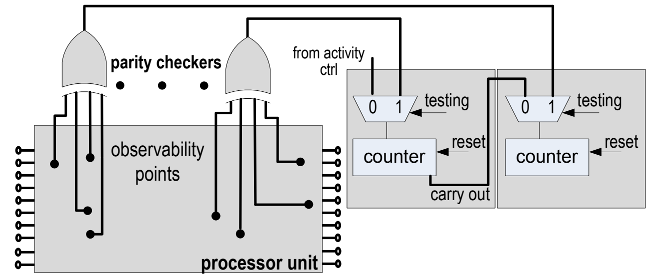

Figure 4: Observability Extensions. Each processor's unit is augmented by a set of observability monitors. Monitors are grouped based on physical proximity and compressed through a parity checker. The output of each parity checker is fed to a local counter - the same counters used for the activity monitors. Counters are evaluated at the end of the test to determine if the outcome is correct.

To boost the coverage provided by the test routines we augment the processor's logic with observability monitors. Indeed, faults not detected during a test can be classified as either non-controllable or non-observable. Non-controllable faults lay in logic paths that are not exercised by testing. Usually they correspond to nodes that are stimulated only by rare events not controllable through deterministic software programs, such as external interrupts and error conditions. Non-observable faults correspond, instead, to internal nodes that toggle during the test, but whose eventual failure does not manifest in test's outcome. Longer, more accurate software tests might be able to expose these faults at the price of higher runtime overhead.

We selected tests capable of controlling the vast majority of fault locations in the design, but the effects of some faults is masked by other logic elements. By analyzing these non-observable fault locations in the gate-level netlist of the processor, we found that they are often grouped in cones of logic. Thus, we added observability monitors at the output of these cones so that a single monitor could provide observability for several internal nodes.

To reduce the amount of signals to monitor, we developed a simple compression circuit consisting of a parity detector. Several observation points are fed to the parity detector and its output is connected to a counter, so that each time the parity signal is asserted, the value of the counter is incremented. The value stored in the counter is reset before the online testing routines are executed. After the test completes, the counter value is read and compared against a reference value: the test is considered successful only if the difference between these two values is within an acceptable range (%10). Counter value variations inferior to that threshold are considered acceptable because of the variation in events common during different executions of a same test in a complex processor system. In contrast, by experimental evaluation, we noted that the occurrence of a fault causes a significant difference in the values stored in these counters (greater than 10%). Figure 4 shows how the observation points are connectted to the parity detectors and to the counters.

5. EVALUATION

We evaluated the quality of our solution on a Sun's OpenSPARC T1 processor [28] and compared against traditional non-adaptive testing in terms of performance overhead, fault coverage, AFC, and area impact.

The processor implements the SPARC V9 ISA and supports 4-way fine grain multithreading. We synthesized the pipeline logic of the T1 with Synposys Design Compiler targeting the Artisan IBM 130nm library. Fault coverage was obtained through fault simulation of functional vectors with Synposys TetraMAX.

Benchmarks: The NAS parallel benchmark suite was used to estimate the overhead on cpu-intensive programs. In addition, we evaluated our solution on I/O intensive benchmarks such as *bonnie* and *stream*. To estimate performance on a benchmark that relies on both CPU and I/O, SPECWeb was also considered.

Performance: Statistics on the functional unit utilization were collected through Simics simulations. Performance was measured in number of committed instructions. Performance impact of our design was evaluated against three epoch lengths: 20, 50, and 100 million cycles.

Fault Coverage: Our experiments focus on stuck-at faults and do not account for faults either marked as undetectable by the ATPG tool are within the DFT structures. Because all memory structures are protected with either parity bits or error-correcting codes, single hard faults in memory are detected by mechanisms already present in the design [20].

Area Overhead: The hardware additions necessary for our diagnosis system were developed in Verilog RTL and synthesized with the IBM Artisan 130nm library with Synopsys Design Compiler.

5.1 Fault Coverage

We first found the maximum fault coverage achievable using Sun's functional verification software routines. Because the overhead introduced by running all these programs sequentially is very high, we paritioned the tests into subsets. We also split the processor into several functional units and selected our subsets such that each was composed of tests that provided high fault coverage for a particular functional unit. These functional units are: integer pipeline, error detection, divider, multiplier, floating point frontend, and stream processor.

We first focused on the processor's integer pipeline since its correctness is vital to nearly every instruction. The integer pipeline consists of four modules: instruction fetch, execution, load store, and trap logic. As detailed in Section 4.1, an ILP solver was used to select offline the subset of tests that yield the highest fault coverage within a preset time budget.

For our fault coverage reports, we partitioned the integer pipeline in into four sub-modules: instruction fetch, execution, load store,

OpenSPARC T1 Unit	Chip Area (%)	Test Coverage (%)					
		Maximum achievable	5.0M cycles	2.5M cycles	1.25M cycles	0.5M cycles	1.25M cycles w/ observation points
Instruction Fetch	7	94.4	93.8	93.2	88.9	82.8	
Execution	10	97.1	96.4	95.9	95.2	94.0	
Load Store	6	89.7	88.1	87.6	86.2	82.8	89.1
Trap Logic	10	88.7	86.0	85.5	84.3	78.7	87.1
Error Detection	1	33.6	33.5	29.6	27.7	26.5	
Multiplier	4	99.2	96.6	96.5	91.0	80.1	
Divider	4	98.7	98.7	95.5	95.5	91.3	
Stream Processing	3	93.7	89.1	84.8	79.9	60.5	
Floating Point Frontend	4	91.5	90.0	85.3	77.9	67.7	
Memory	51	100.0	100.0	100.0	100.0	100.0	
Total *(with Memory)*	100	96.3	95.5	94.9	93.6	91.0	94.1
AFC for Integer Pipeline		96.6	95.9	95.7	94.8	93.2	95.5

Table 1: Fault coverage by integer pipeline tests. We report the fault coverage achievable for the different logic modules of the OpenSPARC T1. The last two rows contain the fault protection achievable for the entire processor, for the area fault coverage and AFC, respectively. Note that, although coverage for units that are not in the integer pipeline plummets when test time is shortened, the AFC of the tests for applications that only use the integer pipeline decreases much less significantly.

and trap logic. These partitions roughly divide the processor along functional module boundaries; the approximate chip area fraction for each logic module is shown in the second column of Table 1. The other columns report the fault coverage achieved by the tests that focus on the integer pipeline. The column 'Maximum' shows the maximum percentage of faults detectable in the chip using Sun's functional verification tests. The remaining columns indicate the fault coverage achieved over a range of test time execution constraints, from .5 to 5 million cycles. The last two rows report, respectively, the total fault coverage when memory is taken into account, and the Application-Aware Fault Coverage for an application workload that relies 100% of the time on the integer pipeline.

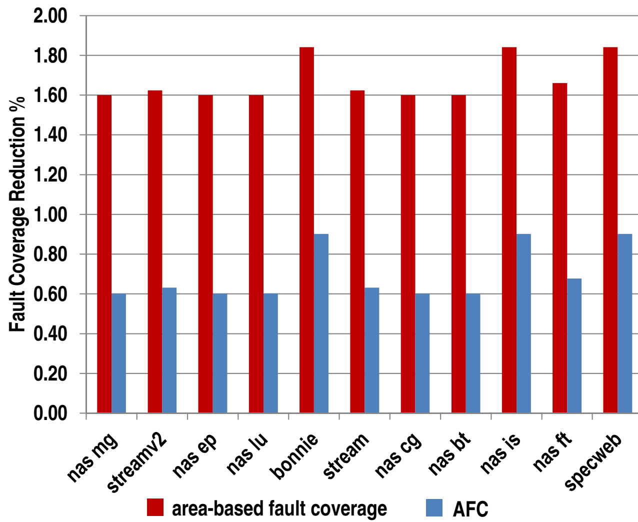

Figure 5: Degradation of Fault Coverage. This graph compares the area-based fault coverage reduction caused by our application-aware testing framework when the test trigger threshold is set to 1% of the committed instructions. Note that the impact on the AFC is much less significant than for area-based fault coverage.

The area-based fault coverage achievable when all tests are executed is extremely high, 96.3%, however, this comes at a very high cost: the execution takes nearly 26 million cycles. Thus we had to select a subset of tests that would still lead to high fault coverage with a limited time budget. As shown in Table 1, when the time budget is reduced, fault coverage for the microprocessor modules in the integer pipeline is not affected as significantly as for the other functional units. The load store unit and the trap logic Unit suffers of limited testability and, in an effort to achieve better fault coverage for these units, we enhanced them with 869 and 738 observation points respectively. For the 1.25 million cycles time budget solution, the area-based fault coverage for these units improves by 3%, as reported in the last column of Table 1.

OpenSPARC Unit	Target Coverage (%)	Cycles
Multiplier	98	27,383
Floating Point Frontend	96	230,033
Divider	98	290,715
Stream Processing Unit	96	1,572,807

Table 2: Area-based fault coverage tests for specific functional units. The fault coverage for each units acconts for both faults in the logic and in the memory structures.

Since this first test was focused on the integer pipeline, area-based fault coverage for the other functional units drops precipitously as the time budget decreases. To target these specialized units, distinct test subsets were selected by solving dedicated ILP module directed problems: target fault coverage and number of cycles necessary to execute the particular tests are reported are Table 2. These modular tests target a very high fault coverage but are very time consuming. For instance, performing a thorough integrity check on the SPU is extremely expensive, accounting for more than 1.5 million cycles. However, none of the benchmarks we evaluated utilized that particular functional unit. This results further support our idea that, in order to maintain low performance overhead, tests should only be triggered on the hardware that was exercised by the application and thus could have impacted computation results.

Avoiding these functional unit tests lowers area-based fault coverage. For example, Figure 5 plots the difference in fault coverage observed when running an application-oblivious test vs. an application-aware test. For each testbench we plot the difference in coverage measured both using the area-based fault coverage and also our AFC metric. In this experiment the epoch considered was 100M instructions long and we used a trigger threshold of 1% for the specialized tests in evaluating our AFC metric. Note from the

figure that the area-based fault coverage is reduced by 1.7% on average when going from an oblivious test to an adaptive one. At the same time, the AFC metric is only reduced by 0.7% on average (see Figure 5). We attribute this small degradation in AFC to a reduction of the fault coverage in the integer pipeline. In fact, instructions executed in the specialized functional units rely also on the integer pipeline. For instance, instructions used to load and store floating point values need to be supported by the load store unit. Thus, triggering hardware tests that need to move floating point values from and to memory will cause an increase in area-based fault coverage of the integer pipeline, which in turn impacts the AFC.

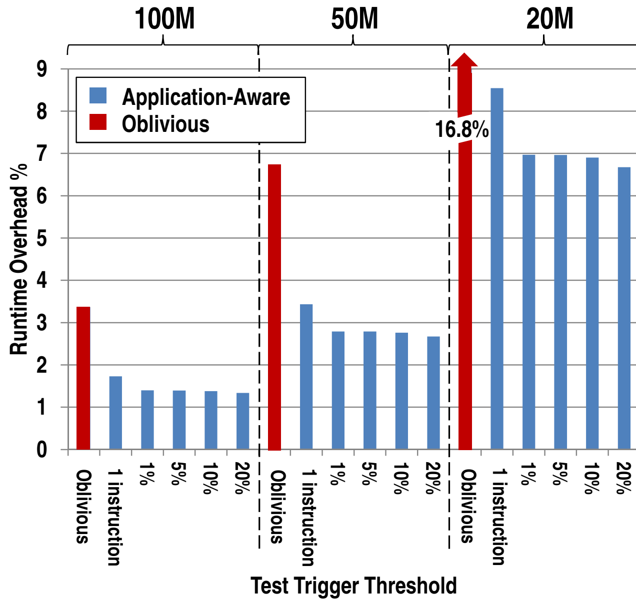

Figure 6: Performance overhead of our application-aware diagnosis for variable epoch sizes. For a given epoch size, our application-aware diagnosis mechanism reduces testing overhead by 50% compared to application-oblivious solutions. The data reported in this graph is the mean of the performance overhead over all the benchmarks.

5.2 Performance Overhead

To evaluate the perfomance overhead of our solution, we select the integer pipeline test bound to 1.25 million cycles, since it provides a good compromise between area-based fault coverage and test runtime.

We evaluate the effects of our application-aware solution against an online testing solution that is oblivious to application behavior. Figure 6 plots the geometric mean of the performance overhead experienced by the benchmarks considered. For both approaches, performance impact was measured using epoch lengths of 100M, 50M, and 20M cycles. Testing time reduction can be capitalized to reduce the epoch's length as shown in Figure 6. On one hand, a shorter epoch length is beneficial since it reduces a checkpoint memory footprint, as reported in [7]. On the other hand, longer epoch increases performance since it directly affects test frequency. However, due to the overhead reduction achieved using our scheme, is possible to run hardware tests more frequently without losing performance. We also evaluated the effect of varying the module utilization triggers, ranging from a minimum of one instruction (a single instruction exercising a specialized module is sufficient to trigger its test), to 20% of the executed instructions.

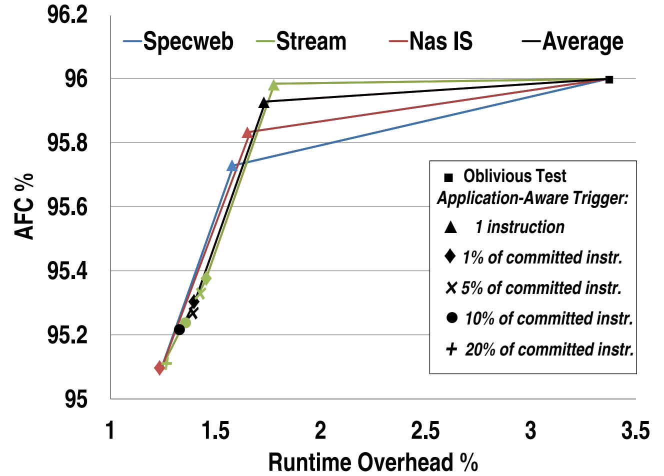

Figure 7: Trade-off between runtime overhead and AFC. This figure shows the impact on performance and AFC of our Application-Aware adaptive mechanism for some significant benchmarks and for the average among all the considered applications. The markers in the graph report different instructions thresholds triggering the functional test: 0 (oblivious solution, tests are always triggered), 1 instruction, 1%, 5%, 10%, and 20% of the committed instructions. Increasing test trigger thresholds impacts the performance of the different benchmarks, yielding lower runtime overhead with limited impact on AFC.

The runtime overhead of our adaptive test system was measured on several benchmarks against different test trigger thresholds. In Figure 7, we plot the runtime overhead of our proposed technique against an oblivious testing solution for the average of our benchmarks and for some significant applications. In particular, we compare the variation of AFC of our system against the AFC obtained by the oblivious solution. Runtime overhead in Figure 7 is reported for an epoch length of 100 million instructions. As for the graph in Figure 6, several test trigger thresholds to activate the modular tests are considered (from 1 instruction to 20% of the instructions executed in the epoch). Note that for the benchmark Nas IS the AFC achievable by our adapive system saturates when the adaptive test trigger reaches 1% of the committed instructions. This behavior is common among the several applications that only rely on very few processor functional units.

As expected, by using test adaptation the performance overhead caused by online testing decreases when the trigger threshold increases. As reported in the graph, application-aware hardware online tests allow a reduction in performance overhead of more than twice that of the application-oblivious solutions. For low values of the test trigger threshold, AFC changes are negligible.

5.3 Area Overhead

Our diagnosis mechanism requires additional hardware for counters used as activity monitors and as fault detectors for the inserted observation points. These logic counters can share hardware resources since they are never used at the same time: the activity monitors are used when the system is operative, while the observation point monitors are active when the system is under test. In the design considered, only five 64-bit harware counters were required, yielding a total area overhead of 0.4%.

Our framework can take advantage of counters already present in silicon for other purposes such as post-silicon verification or structural testing. Event counters for hardware monitoring are, in fact,

already used by the semiconductor industry [3]. If such event counters are not available, design for testability (DFT) structures can be used instead: modern digital systems already implement registers to store the outputs of test vectors applied during structural testing [1]. If any of these structures exist in the processor, the counters necessary for our solution will not have any hardware impact.

We assume that our framework adopts 64-bit hardware counters that can be split in eight 8-bit counters. If these 8-bit counters are time multiplexed 3 times during the test, the 5 activity monitors already present in our design are sufficient to monitor all the observation points. We estimated that the addition of the parity compressors for the extra observation points requires an additional area overhead of 0.4%. Therefore, depending on the presence of already available hardware counters, the total overhead of our system is comprised between 0.4% and 0.8%.

6. CONCLUSIONS AND FUTURE WORK

We proposed a novel solution to detect and diagnose permanent faults in microprocessors. Our system relies on a hybrid hardware/software technique to adapt hardware testing to the dynamic use of the processor's structures. Components that are exercised more often, are tested with higher frequency and accuracy. This allows a significant benefit in performance while improving software protection, since the test targets the faults that are most likely to corrupt computations.

We also intoduced a new Application-aware Fault Coverage (AFC), a metric that represents the ability of a test to detect faults that can corrupt the application. We implemented our framework on the Sun OpenSPARC T1, finding that it achieves an Application-Aware Fault Coverage of 95.5% while maintaining minimal performance overhead (1.3%) and area impact (0.4%). Compared against classic online testing solutions oblivious to application, we showed that considering dynamic application behavior is very beneficial, yielding a reduction of test overhead greater than 50% without severely affecting AFC.

Our results are very encouranging, and we would like to extend this work in several directions. In particular, we would like achieve a higher fault coverage through the development of software tests that target the currently unstimulated logical elements in the processor. We would like to further study the observation points to reduce their cost and improve their effectiveness. In this work we focus only on stuck-at faults. To achieve a better degree of online protection against permanent faults, we want to extend our system to other fault models.

7. REFERENCES

[1] M. Abramovici, M. Breuer, and A. Friedman. *"Digital Systems Testing and Testable Design"*. IEEE Press, 1995.

[2] S. Andrica and G. Candea. "PathScore-Relevance: A Metric for Improving Test Quality." In *Proceedings of the Workshop on Hot Topics in System Dependability (HotDep)*, Jun 2009.

[3] S. Baartmans and B. White. *U.S. Patent no. 6438664: "Customizable event creation logic for hardware monitoring"*. Intel Corporation, Oct 2007.

[4] A. Benso, A. Bosio, P. Prinetto, and A. Savino. "An on-line software-based self-test framework for microprocessor cores." In *Proc. of Design and Test of Integrated Systems in Nanoscale Technology*, Sep 2006.

[5] S. Borkar. "Designing reliable systems from unreliable components: the challenges of transistor variability and degradation." In *Proc. of MICRO*, Nov 2005.

[6] S. Borkar, N. Jouppi, and P. Stenstrom. "Microprocessors in the era of terascale integration." In *Proc. of DATE*, Apr 2007.

[7] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. "Software-based defect tolerance for chip-multiprocessors." In *Proc. of MICRO*, Dec. 2007.

[8] K. Constantinides, S. Shyam, S. Phadke, V. Bertacco, and T. Austin. "Ultra low-cost defect protection for microprocessor pipelines." In *Proc. of ASPLOS*, Oct. 2006.

[9] S. Gupta, A. Ansari, S. Feng, and S. Mahlke. "Adaptive online testing for efficient hard fault detection." In *Proc. of ICCD*, Oct 2009.

[10] E. Karl, D. Blaauw, D. Sylvester, and T. Mudge. "Reliability modeling and management in dynamic microprocessor-based systems." *Proc. of DAC*, Jul 2006.

[11] E. Karl, P. Singh, D. Blaauw, and D. Sylvester. "Compact in-situ sensors for monitoring NBTI effect and oxide degradation." In *Proc. of the Solid-State Circuits Conference*, Sep 2008.

[12] M.-L. Li, P. Ramachandran, S. Sahoo, S. Adve, V. Adve, and Y. Zhou. "Understanding the propagation of hard errors to software and implications for resilient system design." In *Proc. of ASPLOS*, Mar 2008.

[13] Y. Li, M. Samy, and S. Mitra. "CASP: Concurrent autonomous chip self-test using stored test patterns." In *Proc. of DATE*, Mar 2008.

[14] J. Lienig. "Interconnect and current density stress - an introduction to electromigration-aware design." *Proc. of the Workshop on System Level Interconnect Prediction*, Apr 2005.

[15] M. Mehrara, M. Attarian, S. Shyam, K. Constantinides, V. Bertacco, and T. Austin. "Low-cost protection against SER upsets and silicon defects." In *Proc. of DATE*, Apr. 2007.

[16] A. Meixner and D. Sorin. "Detouring: Translating software to circumvent hard faults in simple cores." In *Proc. of DSN*, Jun 2008.

[17] J. Musa. "Operational profiles in software-reliability engineering." *IEEE Software*, Mar 1993.

[18] M. Nafría, J. Suñé, and X. Aymerich. "Breakdown of thin gate silicon dioxide films–a review." *Microelectronics and Reliability*, July 1996.

[19] B. Panzer-Steindel. "Data Integrity" Technical Report, *CERN*, Apr 2007.

[20] I. Parulkar, A. Wood, J. Hoe, B. Falsafi, S. Adve, J. Torrellas, and S. Mitra. "OpenSPARC: An open platform for hardware reliability experimentation." In *Proc. of the Workshop on Silicon Errors in Logic - System Effects*, Mar 2008.

[21] A. Pellegrini, V. Bertacco, and T. Austin. "Fault-based attack to RSA authentication." *Proc. of DATE*, Mar 2010.

[22] M. Psarakis, D. Gizopoulos, M. Hatzimihail, A. Paschalis, A. Raghunathan, and S. Ravi. "Systematic software-based self-test for pipelined processors." In *Proc. of DAC*, Jul 2006.

[23] E. Rosenbaum, R. Rofan, and C. Hu. "Effect of hot-carrier injection on n- and pMOSFET gate oxide integrity." *IEEE Electron Device Letters*, Nov 1991.

[24] D. Schroder. "Negative bias temperature instability: What do we understand?" *Microelectronics Reliability*, Jun 2007.

[25] B. Schroeder, E. Pinheiro and W.-D. Weber. *"DRAM errors in the wild: a large-scale field study"*. *Proceedings of the conference on Measurement and modeling of computer systems*, Jun 2009.

[26] J. Srinivasan, S. Adve, P. Bose, and J. Rivers. "The impact of technology scaling on lifetime reliability." *Proc. of DSN*, Jun 2004.

[27] A. Strong, E. Wu, R.-P. Vollertsen, J. Sune, G. LaRosa, and T. Sullivan. *"Reliability Wearout Mechanisms in Advanced CMOS Technologies*." Wiley Press, 2009.

[28] Sun Microsystems Inc. "OpenSPARC T1 microarchitecture specification." Aug 2006.