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. Current online testing mechanisms focus on testing all components in a microprocessor, including hardware that has 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 affect application correctness the most, and it achieves high coverage while incurring minimal performance overhead. We also introduce a new metric, Application-Aware Fault Coverage, measuring a 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 of online testing by more than a factor of 2 when compared to solutions oblivious to application's behavior. Specifically, we found that our solution can achieve 95% fault coverage while maintaining a minimal performance overhead (1.3%) and area impact (0.4%).

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 jeopardize transistor robustness [2].

Reliability issues may be triggered by a wide range of causes: from manufacturing problems, such as optical proximity effects and processing material defects, to 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, including oxide breakdown, hot carrier injection, negative bias temperature instability and electromigration. The problem is further exacerbated by aggressive mainstream testing techniques such as burn-in [16], 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 affects all transistors, often shortening their lifetime.

The impact of runtime transistor failures on processors varies greatly: from causing disruptive software behavior to corrupting computation without providing any warning signs [8]. Faults that cause silent alterations to the output of software applications are particularly concerning. For instance, silent data corruptions leading to system outputs that 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 [13]. As the size of transistors decreases with technology, expected fault rates are projected to increase drastically, causing concerns on the correctness of computation by any computer system [2].

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 into epochs and normal execution is periodically suspended to run a battery of tests on the microprocessor [5]. Periodic testing of microprocessors can be accomplished through the addition of ad-hoc hardware testing components [5, 11] and/or through the execution of high-quality software test sequences [4, 9]. 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% [4]. 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. In contrast, our work proposes a novel approach to online testing of processors, emphasizing application sensitivity to runtime hardware faults.

1.1 Contributions

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

- We propose an Application-Aware testing framework - A^2Test - that **dynamically tunes a test to focus on the hardware units exercised by software**. As a result, we can limit testing overhead, while providing close and careful monitoring of hardware units that, if faulty, might have corrupted software outputs.
- We rely on a **hybrid hardware/software solution** to deliver this online approach: software tests check the integrity of the underlying hardware, while lightweight hardware observation points are inserted to improve test coverage.
- We propose a **new metric called *Application-Aware Fault Coverage*** **(A^2FC)** which measures test quality, while accounting for dynamic hardware usage. Thus, A^2FC expresses the effective fault protection experienced by the user.

We evaluated test quality, performance overhead and area impact of our diagnosis mechanism on a processor based on an OpenSPARC T1 core [17]. We found that the area overhead of our design is negligible (approximately 0.4%), and it can provide 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 detail the fault coverage achievable by software testing on a multi-threaded microprocessor.

2. RELATED WORK

Classic runtime testing techniques focus on achieving high area fault coverage, regardless of processor utilization. These solutions

incur a constant runtime overhead due to periodic testing. Below we overview the two groups of solutions.

Hardware-based detection mechanisms insert extra microarchitectural features in order to detect faulty transistors on the chip. For example, Constantinides, *et al.* [5] and Mehrara, *et al.* [11] propose embedding Built-In-Self-Test units and checkers to test the integrity of VLIW processors. 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 solution to perform runtime fault detection [4, 9] and it is the solution that can achieve the highest fault coverage in digital systems. However, area and performance overheads limit its viability for online reliability schemes. Structural testing requires significant area overhead, approximately 6%, 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, a few seconds at a time for the solution proposed in [9]. **Software-based** techniques utilize software routines providing high fault coverage with no hardware additions [1, 14]. Their focus is to achieve an elevated fault coverage across all structures of the CPU, posing a significant performance overhead, regardless of the dynamic usage and health of the processor. In contrast, our hybrid solution takes advantage of the low cost of software testing, but improves its coverage through the addition of extra observation points. Gupta, *et al.* [7] suggested tuning the execution of functional tests to the health of the silicon elements within the processor. They estimate hardware health through in-situ oxide breakdown and NBTI sensors spread throughout the silicon, thus increasing chip area by 2.6%. Gupta, *et al.* focus only on a single type of silicon defect: extending their approach to handle other sources of silicon degradation would require the deployment of different specialized sensors.

All these solutions are oblivious to application behavior; our hybrid solution, in contrast, proposes to monitor processor's utilization. Doing so, we can tune our tests to target those hardware structures that have been subjected to high activity, and thus are at higher risk of corrupting the application.

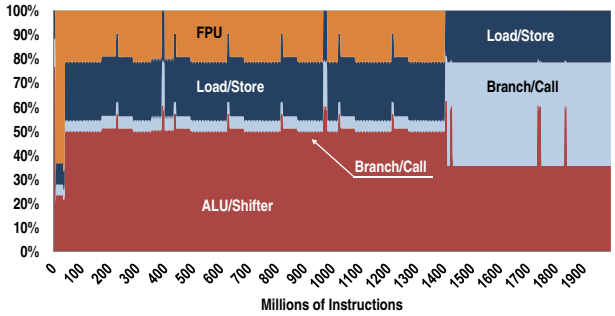

Figure 1: Dynamic instructions in the Nas FT benchmark. This application solves differential equations using Fourier transforms. The figure shows the type of dynamic instructions executed over a window of 2 billion instructions and their distribution by type.

3. APPLICATION-AWARE 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 vary 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, Fourier Transform from the Nas suite [6]. Note that the execution is characterized by long phases executing instructions which require only a portion of the hardware units. These patterns of utilization are not unique to this benchmark, but are common to most applications.

Research has shown that applications are only susceptible to faults that occur in the hardware units contributing to the computation of the program's outcome [8]. 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. The metric that we propose below is inspired by software testing techniques such as operational profile [12], where the code is profiled at design time to determine the software functions that are executed often by users. 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 user's population. This approach is very powerful and widely adopted in the software industry due to its benefit-to-cost ratio. Online hardware testing, on the other hand, is typically oblivious to hardware utilization, thus targeting 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* - A^2FC. A^2FC measures the quality of a test with respect to its ability to detect a fault in hardware units that the application has exercised. For instance, if an application only uses the integer pipeline of a processor, a test that detects faults occurring exclusively in the floating point unit (FPU) would provide an A^2FC 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 A^2FC of 40%. In other words, instead of measuring the fraction of transistors that a test covers, A^2FC 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 our A^2FC metric. Let P_f represent the probability of a single transistor permanently 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 a hardware unit comprising n transistors, all subject to the same switching activity. Assuming that transistor failures are independent events, then the probability of at least one error occurring in a unit after s toggles is $1 - (1 - P_f)^{sn}$. Using Taylor binomial expansion, this expression becomes: $snP_f + \binom{sn}{2}P_f^2 - \cdots$

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 had 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. As 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. Assuming a negligible probability of having two faults manifesting in the processor within the time frame delimited by two subsequent tests, the probability of a chip incurring a fault is given by the sum of the individual probabilities that any of its modules incurred 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 the *Application-Aware Fault Coverage* of a test to be the ratio between faults that can be detected by the test and all possible occurring faults. Since P_f is common to all terms, it can be removed from the expression, yielding:

$$A^2FC = \frac{\sum_i (s_i n_i c_i)}{\sum_i (s_i n_i)}$$

Note that our A^2FC 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 area fault coverage, 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 dynamic behavior into account, our A^2FC metric would report a 90% coverage for the first test, against a 95% for the second. We use the A^2FC metric in this work to evaluate how effective a test is in protecting a system against failures that can affect the correctness of software application.

4. APPLICATION-AWARE DIAGNOSIS

Our diagnosis framework takes advantage of dynamic program behavior to reduce the overhead required for periodic testing without affecting A^2FC. 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. Similarly, a test that optimizes performance without affecting A^2FC 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 underlying hardware is assessed through hardware counters, called *activity monitors*. An activity monitor is associated with each functional unit in the processor. Every time an instruction exercises a particular functional unit, the corresponding counter is incremented. To characterize the dynamic behavior of an application, the activity monitors are reset at the beginning of each epoch. At the end of the epoch, our proposed framework evaluates the monitor's counters to determine which hardware units should be tested. To optimize the overhead imposed by our detection mechanism, we developed a fully-adaptive framework, so that unit-focused tests are triggered on demand. Specifically, during each testing phase, we execute only test routines exercising functional units that were utilized by the software application during the last execution interval. Units which did not experience utilization, based on the information from the activity monitors, are not tested since they would not improve overall A^2FC. This approach is beneficial for two reasons. First, 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.

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 performance 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. In addition, the same hardware counters used as activity monitors, can be used during testing to collect data from the observation points and their final value is used to verify test success. We found experimentally that with our integrated approach, we can expose the vast majority of microprocessor's faults and, in particular, the ones applications are most sensitive to, without incurring the high cost of traditional testing mechanisms such as BIST and scan-chains.

As a case study we analyzed the behavior of the application Nas FT using an epoch of 20 million cycles. During normal computation, every time an instruction exercised a functional unit, the corresponding activity monitor was incremented. At the end of the epoch, the activity monitor associated with the floating point unit and the divider reported an utilization of 5.9 million and 1 million instructions, respectively. We had setup a 10% utilization threshold for this case study and, consequently, our A^2Test triggered hardware tests for the integer pipeline and for the floating point unit, but not for the divider.

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 [14]. Several techniques has been proposed in the literature to build software test routines to this end [3]. In our framework, we choose 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 those tests most effective in detecting faults for a given unit. We accomplish this goal by formulating an integer linear programming (ILP) problem, such that its solution 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 front end, and stream processing unit. Solutions for the ILP problems generated are computed only once at design time and used to select the routines that should test each module at runtime.

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)

b)

t_i = { 1, if T_i ($0 \le i \le 4$) is selected; 0, otherwise }

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

Constraint inequalities:

$t_1 + t_2 \ge f_0$ | $t_2 + t_4 \ge f_3$

$t_0 + t_1 \ge f_1$ | $t_0 + t_3 + t_4 \ge f_4$

$t_0 + t_2 \ge f_2$ | $t_1 + t_4 \ge f_5$

c)

	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($\sum_i t_i c_i$)

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

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). One inequality is also added for each possible fault location. The binary variable that represents a test i, t_i, is set to 1 if and only if the associated test is selected for execution. A variable associated with a fault location j, f_j, is greater than 0 only if the fault is exposed by at least one of the tests that will be executed. Since the integer pipeline is active all the time while the system is operational, the corresponding test is selected in each testing session. As a result, this test has the most impact on test execution and, consequently, we set a hard constraint on its time budget. In contrast, the test time for all other units 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 both 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 preset 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 this ILP instance 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 test execution time. Instead, we add a constraint requiring that overall coverage is above a user-specified threshold (Figure 2.c).

The ILP problem for the integer pipeline in a complex processor 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. Note that solution to each ILP problem must only be computed once when the microprocessor is designed; thus, although these problems are resource-consuming, their complexity is well manageable, even for a modern processor such as the T1.

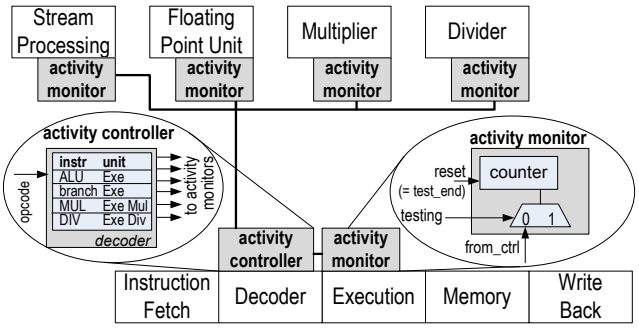

Figure 3: Activity monitors track the use of each processor unit so that tests can be adapted to target those activated during the last execution interval. Monitors' counters are incremented by a controller based on the instruction flow and reset at the beginning of each execution interval.

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 its 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 counters' values. Functional unit testing can be triggered when a functional unit's utilization rises above a preset threshold. In our framework, we assume that users could configure the desired trigger thresholds dynamically, so to trade-off performance overhead with A^2FC.

4.3 Microprocessor Observability Extensions

To boost the coverage provided by the test routines we augment the processor's logic with observability points. 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 the test's outcome.

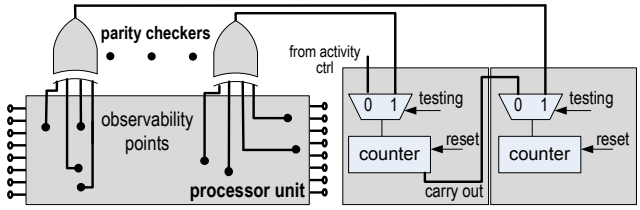

Figure 4: Observability extensions. Each processor's unit is augmented by a set of observability points, compressed through a parity checker and fed to local counters. Counters' values are then evaluated determine test correctness.

We selected tests capable of controlling the vast majority of fault locations in the design. By analyzing the non-observable nodes in the gate-level netlist of the processor, we found that these are often grouped in cones of logic. From the processor netlist, we built a graph connecting all non-observable locations in the design. We then identified the cones of logic rooted at each of these locations through a breadth-search-first algorithm. The non-observable nodes corresponding to cones containing at least four other non-observable locations were selected to be instrumented with an observation point. Through this selection, we extended system's observability without significantly affecting its area.

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 at the beginning of the testing sequence. After the test completes, the counter is compared against a reference value and it is considered successful only if the difference between these two values is within an acceptable range (which we set at 10%). Counter value variations below the threshold are considered within the normal range for a complex processor executing a same test multiple times. Indeed we noted, by experimental evaluation, that the occurrence of a fault causes a significant difference in the counters values (greater than 10%). Figure 4 shows the schematic of our compression circuit. Note that we can use the same counters

for both our compression circuitry and for the activity monitors, discussed in the previous section.

5. EVALUATION

We evaluated the quality of our solution on a Sun's OpenSPARC T1 processor [17] and compared against traditional non-adaptive testing solutions in terms of performance overhead, fault coverage, A^2FC and area impact. At the end, in Table 3, we provide a direct comparison of our evaluation with a number of previously published solutions. The processor implements the SPARC V9 ISA and supports 4-way fine grain multi-threading. 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. The Nas parallel benchmark suite was used to estimate the performance overhead on CPU-intensive programs [6]. In addition, we evaluated our solution on I/O intensive benchmark suites such as Bonnie [18] and Stream [10]. To estimate performance on a benchmark that relies on both CPU and I/O, SPECWeb was also considered [15]. Statistics on functional unit utilization were collected through Simics simulations. Performance was measured in number of committed instructions and impact of our design was evaluated against three epoch lengths: 20, 50, and 100 million cycles. Our experiments focus on stuck-at faults and do not account for faults either marked as undetectable by an automatic test pattern generator or within the design-for-test structures. Because all memory structures are protected with either parity bits or error-correcting codes, single permanent faults in memory are detected by mechanisms already present in the design [17]. Finally, 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.

OpenSPARC T1 Unit	Area (%)	Test Coverage (%)					
		No limit	5.0M cycles	2.5M cycles	1.25M cycles	0.5M cycles	1.25M w/ obs
Instr. 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 Detect.	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 Proc.	3	93.7	89.1	84.8	79.9	60.5	
FP Front End	4	91.5	90.0	85.3	77.9	67.7	
Memory	51	100.0	100.0	100.0	100.0	100.0	
Total *(w/ Memory)*	100	96.3	95.5	94.9	93.6	91.0	94.1
A^2FC		96.6	95.9	95.7	94.8	93.2	95.5

Table 1: Fault coverage achieved by integer pipeline tests. For each module in the OpenSPARC T1, we report area occupied and fault coverage attained for test groups targeting the integer pipeline. The last two rows indicate total area coverage attainable and A^2FC for an application relying exclusively on the integer pipeline.

5.1 Fault Coverage

We first determined the maximum fault coverage achievable when using Sun Microsystems' functional verification software routines. Because the overhead introduced by running all these programs sequentially is very high, we partitioned the processor into several functional units and grouped the test routines based on the functional units for which they provide high coverage. The functional units are listed in Table 1. 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 group of tests that yields the highest fault coverage within a given time budget. Table 1 shows the coverage attained for each module when running our integer pipeline test battery, over a range of execution budgets. The last two rows in the table report the total fault coverage attainable for the system, and the Application-Aware Fault Coverage for an application workload that relies exclusively on the integer pipeline.

The area-based fault coverage attainable by executing all tests in the integer pipeline group is extremely high, 96.3%. However, this comes at a very high cost: the test sequence requires nearly 26 million cycles. Thus we deemed necessary to select a subset of tests that would still lead to high fault coverage but within a limited time budget. As shown in Table 1, when the time budget is reduced, fault coverage for the integer pipeline modules is not effected as significantly as for other functional units. Among the T1, the load-store unit and the trap logic unit suffer of limited testability and, in an effort to increase their fault coverage, we enhanced them with 869 and 738 observation points, respectively. This led to a 3% improvement in the area-based fault coverage of these units, as indicated in the last column of Table 1.

Note that the coverage for the other functional units plummets as the time budget decreases, since the test group is focused only on the integer pipeline. For the other functional units, distinct test groups were selected by solving dedicated ILP problems: target fault coverage for the multiplier and the divider was set to 98%, and the cycles necessary to complete the corresponding test group are 27,383 and 290,715, respectively. For the floating point front end and for the stream processing unit, the target fault coverage was set to 96%, requiring a runtime of the 230,033 cycles for the first test group and 1,572,807 cycles for the second. These test groups target a very high fault coverage but are very time consuming. For instance, performing a thorough integrity check on the stream processing unit is extremely expensive, accounting for more than 1.5 million cycles. However, the unit is utilized infrequently, indeed none of the benchmarks made use of it. This observation further supports the hypothesis that, in order to maintain low performance overhead, tests should only be triggered on the hardware units that could have impacted computation results. We then compared the degradation in fault coverage observed when running an application-oblivious test vs. an Application-Aware test, over a range of application programs. In this experiment the epoch considered was 100M instructions long and we used a threshold of 1 instruction to trigger unit tests. From our experiments, we report that the area-based fault coverage is reduced by 1.7% on average when going from an oblivious test to an adaptive one; correspondingly the A^2FC metric is only reduced by less than 0.1% on average.

5.2 Performance Overhead

To evaluate the performance overhead of our solution, we set a bound of 1.25 million cycles for the integer pipeline, since it provides a good compromise between area-based fault coverage and test runtime. For all the considered epoch lengths, the average of the performance overhead of our Application-Aware solution was more than 50% lower when compared against an online testing solution that is oblivious to application behavior. The runtime overhead of our adaptive test system was also measured for several individual benchmarks against different test trigger thresholds. In Figure 5, we plot the runtime overhead of our proposed technique against an oblivious testing solution for some representative applications and for the average among all the considered applications. In particular, we compare the variation of A^2FC in our system against the A^2FC obtained by an oblivious solution. The epoch

length for this experiment is set at 100 million instructions. Again, several test trigger thresholds to activate the module tests are considered (from 1 instruction to 20% of the instructions executed in the epoch). Note that for the benchmark Nas IS, the A^2FC achievable by our adaptive system saturates when the test trigger reaches 1% of the committed instructions. This behavior is common among several applications that only rely on few CPU functional units at a time. In addition, the runtime required to perform the proposed A^2Test on the OpenSPARC T1 is extremely small compared to the results obtained by techniques such as those in [4] and [9]. As expected, by using test adaptation, the performance overhead of online testing decreases when the threshold increases. Moreover, Application-Aware testing reduces the performance overhead by a factor of two over oblivious solutions.

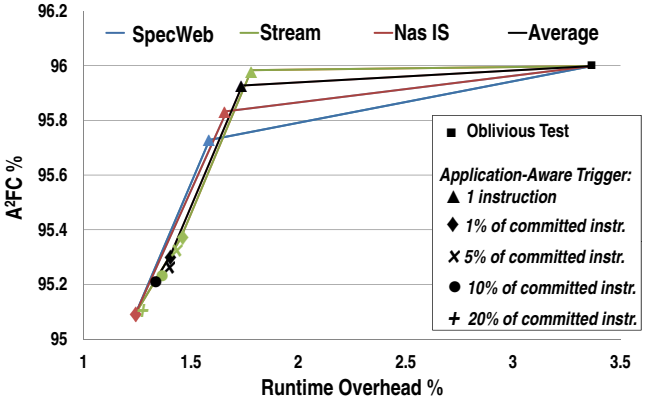

Figure 5: Trade-off between runtime overhead and A^2FC for an epoch length of 100M cycles. This figure shows the impact on performance and A^2FC of our Application-Aware adaptive mechanism for some significant benchmarks. The markers in the graph report different instructions thresholds triggering the functional test, from 0 (oblivious solution), to 20% of the committed instructions.

5.3 Area Overhead

Our diagnosis mechanism requires additional hardware for counters, which are used at different times as activity monitors and as fault detectors for the observation points. In the design considered, only five 64-bit hardware counters were required, yielding a total area overhead of 0.4%. We assume that the five hardware counters can be split in eight 8-bit counters. If these 8-bit counters are time multiplexed three times during the test, no further area additions are needed for our framework. Our framework can take advantage of counters already present in silicon for other purposes, such as post-silicon validation or design for testability (DFT), in which case our solution will not have any hardware impact. We estimated that the addition of the parity compressors for the extra observation points requires an additional area overhead of 0.4%.

To put our results into context, Table 3 compares all three aspects of the evaluation for a range of previously published solutions: note how our solution achieves the best coverage for the smallest area overhead within a reasonable performance impact.

6. CONCLUSIONS

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 impact while improving software protection, since the tests target potential faults that are most likely to corrupt computations.

	Test Technique	Test Coverage (%)	Core Downtime (Cycles)	Area Overhead (%)
A^2Test	Hybrid	95 - 96	1.3 M - 3.4 M	0.8
ACE [4]	Structural	100	5.4 M	5.8
Bulletproof [5]	BIST	88.6	1.5 K	6
Bulletproof2 [11]	BIST	95.2	600 - 3.3 K	14
CASP [9]	Structural	99.5	240 M	6
Health Adapt. [7]	Functional	0 - 97	0 - 690 K	2.6
SW-Based [14]	Functional	90	44 K	0

Table 2: Comparison of online testing techniques. For each technique we report the fault coverage achieved for stuck-at faults, the number of cycles required for testing and the area impact. Note that A^2Test, ACE, and CASP provide results for the OpenSPARC T1, while the other results are based on different architectures.

We also introduced A^2FC, Application-Aware Fault Coverage, a new metric indicating 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 the running application, we showed that considering dynamic application behavior is very beneficial, yielding a reduction of test overhead greater than 50%, without severely affecting A^2FC.

Acknowledgments. The authors would like to acknowledge the support of the Gigascale Systems Research Center.

7. REFERENCES

[1] A. Benso, A. Bosio, P. Prinetto, and A. Savino. "An on-line software-based self-test framework for microprocessor cores." In *Proc. of DTIS* , Sep 2006.
[2] S. Borkar, N. Jouppi, and P. Stenstrom. "Microprocessors in the era of terascale integration." In *Proc. of DATE*, Apr 2007.
[3] L. Chen, S. Ravi, A. Raghunathan, and S. Dey. "A Scalable Software-Based Self-Test Methodology for Programmable Processors." In *Proc. of DAC*, Jun 2003.
[4] K. Constantinides, O. Mutlu, T. Austin, and V. Bertacco. "Software-based defect tolerance for chip-multiprocessors." In *Proc. of MICRO*, Dec. 2007.
[5] 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.
[6] A. Ferrari, A. Filipi-Martin, and S. Viswanathan, "The NAS Parallel Benchmark Kernels in MPL." Technical Report, *Dept. of Computer Science, Univ. of Virginia*, Dec 1995.
[7] S. Gupta, A. Ansari, S. Feng, and S. Mahlke. "Adaptive online testing for efficient hard fault detection." In *Proc. of ICCD*, Oct 2009.
[8] 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.
[9] Y. Li, M. Samy, and S. Mitra. "CASP: Concurrent autonomous chip self-test using stored test patterns." In *Proc. of DATE*, Mar 2008.
[10] J. McCalpin. "STREAM: Sustainable Memory Bandwidth in High Performance Computers." University of Virginia, 2007.
[11] 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.
[12] J. Musa. "Operational profiles in software-reliability engineering." *IEEE Software*, Mar 1993.
[13] A. Pellegrini, V. Bertacco, and T. Austin. "Fault-based attack of RSA authentication." In *Proc. of DATE*, Mar 2010.
[14] 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.
[15] Standard Performance Evaluation Corp. "SPECweb2005." http://www.spec.org/, 2005.
[16] A. Strong, E. Wu, R.-P. Vollertsen, J. Sune, G. LaRosa, and T. Sullivan. "Reliability Wearout Mechanisms in Advanced CMOS Technologies." Wiley Press, 2009.
[17] Sun Microsystems Inc. "OpenSPARC T1 microarchitecture specification." Aug 2006.
[18] Textuality Services, Inc. "Bonnie file system benchmark." http://www.textuality.com/bonnie/, 1996.