Low-Cost Run-time Diagnosis of Hard Delay Faults in the Functional Units of a Microprocessor*

Mahmut Yilmaz, Sule Ozev, Daniel J. Sorin
Dept. of Electrical and Computer Engineering
Duke University

Abstract

This paper addresses the run-time diagnosis of delay faults in functional units of microprocessors. Despite the popularity of the stuck-at fault model, it is no longer the only relevant fault model. The delay fault model —- which assumes that the faulty circuit element gets the correct value but that this value arrives too late — encompasses many of the actual in-field wearout faults in modern microprocessors. In-field wearout faults, such as time-dependent dielectric breakdown and electromigration, cause signal propagation delays which may be missed during production test time. These defects progress exponentially over time, potentially causing a catastrophic failure. Our goal is to diagnose hard delay faults (i.e., identify them as hard faults, not transients) during run-time before they lead to catastrophic chip failures. Results show that we can diagnose all injected delay faults and that prior diagnosis mechanisms, which target only stuck-at faults, miss the majority of them.

1 Introduction

The vast majority of microprocessor fault tolerance mechanisms have targeted stuck-at faults. The stuck-at fault model assumes that a faulty circuit element (e.g., a wire or the output of a gate) is stuck at either zero or one. This fault model applies to both transient and permanent faults. The stuck-at model is popular because it is easy to use and it has, at least historically, represented a large class of actual physical defects (e.g., short or open circuits). This model assumes that the fault will manifest itself in the same way when the same input is applied.

Despite the popularity of the stuck-at fault model, it is no longer the only relevant fault model. The delay fault model assumes that the faulty circuit element gets the correct value but that this value arrives too late. The delay fault model covers many physical phenomena that are becoming more prevalent in modern microprocessors, including the effects of temperature, process variability, and the onset of physical wear-out [5, 10, 15]. For instance, at the beginning of wearout, the only impact is an increase in delay [4, 10], which will eventually cause a delay fault. Delay faults on any circuit path, not just critical paths, can cause an error since the delay amount can easily exceed the timing slack on even short paths [24]. In addition to the effects of normal wearout, the circuit may contain innate defects that manifest as delay faults. While there is extensive research in production testing for delay faults, production test coverage is still low, particularly when these defects cause only small delays, the effects of which get masked at nominal clock frequencies [1, 27].

We want to diagnose a hard delay fault (i.e. identify it as a hard fault) during run-time before its underlying cause, the beginning of physical wearout, becomes catastrophic. If undiagnosed, wear-out can lead to oxide breakdown or other failures that can also ruin nearby circuitry [10] and lead to chip failures.

The majority of fault tolerance techniques for commodity microprocessors have focused on the detection of and recovery from transient faults [2, 16, 19, 20]. Most of these techniques are based on instruction replay and rely on the already existing recovery mechanisms originally designed for speculative execution. These techniques do not diagnose the fault location or the type of the fault and thus cannot reconfigure or gracefully shut down to avoid catastrophic failure. A limited number of approaches aim at diagnosing the hard faults [8, 9, 25]. Most of these techniques target stuck-at faults and rely on the fault manifesting itself in the same way when the same input is applied. Blome et. al. [6] target delay faults, but they focus on faults at the end of the bathtub curve and paths that are critical or near critical. Thus, their technique misses innate defects and faults that are not on the critical path.

Our contribution with respect to this previous work is a run-time diagnosis mechanism for delay faults occurring in the functional units (FUs) of a microprocessor. To determine if an error was due to a hard delay fault, we must be able to reconstruct the circuit switching sequence that led to the error. Thus, we keep a very short history of recent inputs at FUs and use them to determine if the error repeats itself.

*This material is based upon work supported by the National Science Foundation under Grants No. CCF-0444516, CCR-0309164, CCF-0545456, CCF-0540994, and EIA-9972879, the National Aeronautics and Space Administration under grant NNG04GQ06G, an equipment donation from Intel Corporation, and a Duke Warren Faculty Scholarship.

Consistent manifestation of the error indicates a hard delay fault and requires system reconfiguration to take the faulty component out of circulation. Our mechanism can diagnose all injected delay faults (that are not masked), whereas the vast majority of delay faults cannot be diagnosed by previous methods.

2 Related Work

Prior work has explored hard fault diagnosis, timing speculation, and delay fault testing. None of this work, however, diagnoses hard delay faults at run-time.

Hard Fault Diagnosis Reliable servers, such as IBM mainframes [23], detect and diagnose hard faults by using vast amounts of redundancy. For example, in a server with triple modular redundancy (TMR) at the FU-level, a fault in one FU can be diagnosed when the other two FUs repeatedly out-vote it. SRAS [8] uses much less hardware than TMR to diagnose hard stuck-at faults in a microprocessor's array structures (e.g., reorder buffer, load/store queue). DIVA [2] is a lightweight redundancy based technique originally intended for transient errors. DIVA uses simple in-order processors to check for errors in the execution of a complex out-of-order core. While DIVA is designed for error detection only, it could diagnose hard stuck-at faults (but not hard delay faults) if it assumed after repeated error detections that the checkers are correct and that the aggressive core is faulty (i.e., there is no third party to break the tie). We do not believe that this assumption is viable, even if the checkers are implemented with more robust technology. Bower et al. [9] diagnose hard stuck-at faults in microprocessor components, including FUs. For every instruction that is determined to be erroneous (detected by DIVA), they increment error counters for every component that instruction used. If, despite periodic zeroing of the error counters, a particular counter reaches its threshold, then that component is considered to have a hard fault. Their approach only applies to hard stuck-at faults, since it cannot replicate the input sequences necessary to diagnose delay faults, as we will show in Section 4. Blome et al. [6] propose to detect the onset of wearout in a microprocessor by sampling and averaging line delays over time. Once the sampled delay of a circuit reaches a critical limit, a wearout detection unit predicts a catastrophic failure. This method has the ability the diagnose delay faults on critical lines when the circuit approaches the end of its lifetime. Thus, this method is not suitable for faults due to production defects, early onset wearout, or faults that lie on a non-critical path.

Timing Speculation Razor [12] cleverly enables timing speculation, and it might appear that Razor could be used to also diagnose delay faults (even though that is not what it was intended for). Razor flip-flops enable timing speculation (e.g., potentially unsafe dynamic voltage scaling) by combining a normal latch with a shadow latch that is clocked on the next half cycle. Regular flip-flops are replaced with the more expensive Razor flip-flops for those critical and near-critical circuits that could potentially be clocked too fast for certain cases. If the two latches hold different states, then there was a timing mis-speculation and they recover. Razor could diagnose a delay fault after repeated mis-speculations. The key difference between timing mis-speculation and delay faults is that a delay fault can occur on any path at any time. Using Razor to tolerate delay faults would require replacing all flip-flops with Razor flip-flops. However, Razor flip-flops cannot be used when multiple paths intersect in which any one path might be less than half of a clock period [12]. Moreover, Razor would not be capable of detecting delay faults that cause delays of more than half of a clock cycle (it is common for faults to cause gross delays that are nearly as long as the system clock cycle [24]).

Testing for Delay Faults Delay faults have been extensively researched in the context of VLSI testing. Studies show that delay faults are an appreciable percentage of failures and require at-speed or faster-than-at-speed testing [21]. Most prior work on delay fault testing (e.g., [11, 17]) has focused on test pattern generation and detection techniques for off-line manufacturing testing. Our work differs in that it detects and diagnoses delay faults at run-time. We enable diagnosis by repeating the input conditions that result in an error. In off-line test, one has to come up with all the sequences that potentially lead to the manifestation of fault as an error. In our case, the error has already manifested itself, so repeating the input conditions (including the transition conditions) enables diagnosis.

3 Error Detection and Recovery

In this section, we describe error detection and recovery mechanisms. First, we discuss what we expect from a checker (Section 3.1). Then, we list the available error detection mechanisms that can be used with our diagnosis scheme (Sections 3.2). Last, we discuss how to recover from errors (Section 3.3).

3.1 Required Aspects for Detection

Our diagnosis mechanism requires an error detection mechanism for FUs of the microprocessor. We can either use redundant units or specialized checkers, which may complete zero or more cycles after the functional unit completes. If the design constraints require very low area and power overheads, a smaller checker can be implemented at the expense of increased checker latency. The selected

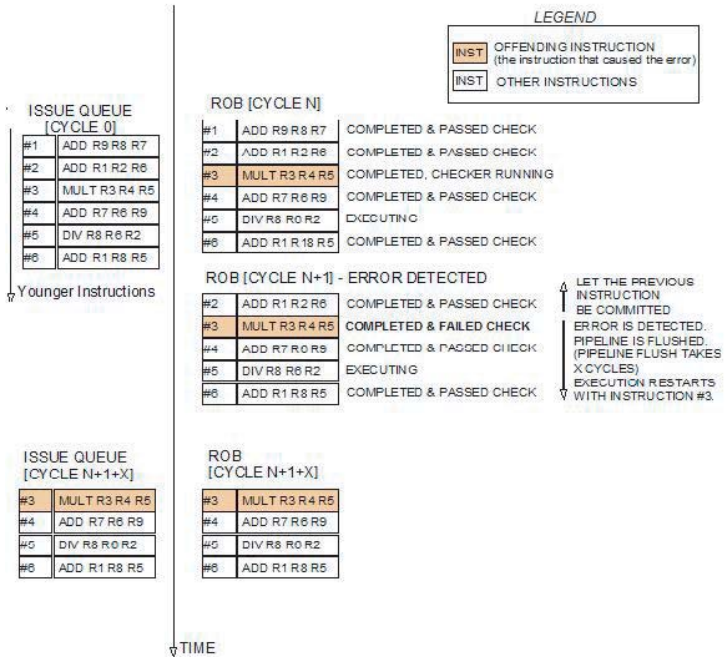

Figure 1. The recovery mechanism using the microprocessor's branch misprediction mechanism

checker does not change the way our diagnosis mechanism works. However, selecting a self-checking checker decreases the probability of marking a non-faulty FU as faulty.

Independent of the selected checker and the FU, we require that each FU makes its results immediately available upon completion, before the checker completes. Instruction completion is often on the critical path, since an instruction's result is often the input to one or more subsequent instructions. However, we commit the FU results only after the checker finishes. Instruction commit is only critical if the reorder buffer (ROB) or load-store queue (LSQ) fills up.

3.2 Options for Error Detection

Error detection for microprocessors and functional units is an extensively researched area. Since we do not require that error detection is instantaneous (i.e., within the same clock cycle), there are many error detection options we can use. Simple replication of FUs enables error detection, since a mismatch in the results of the FUs reveals an error. In this case, the check operation will be completed in the next clock cycle after the FU completion. This is because the comparison of results will take one more clock cycle.

There are light-weight approaches for checking most of the microprocessor pipeline including the FUs. One of them is to use redundant threads [16, 19, 26]. Another one is DIVA [2], which detects errors by checking an aggressive out-of-order core with simple in-order checker cores that sit right before the aggressive core's commit stage. If the result of the aggressive core does not match that of the checker, an error is detected and the aggressive core is flushed and replays from the last uncommitted instruction.

There are also efficient implementations of specialized FU checkers. Vasudevan et al. [25] detect errors in carry select adders by checking partial results and the carry state. The authors report 16% area overhead for a 32-bit adder. This scheme includes a totally self-checking, 2-rail checker which can detect all faults except the ones in the primary inputs and primary outputs. Yilmaz et al. [28] detect errors in a recursive multiplier by using a modulo-3 checker. The authors report 26% area overhead and almost 99% fault coverage for this scheme. Self diagnosis for the checker is enabled by swapping the inputs of the checker after diagnosing a hard fault. The same scheme can also be applied to a Booth-encoded multiplier, which is more common in industrial applications [14, 18].

3.3 Recovery

If the result of the FU does not match that of the checker, the recovery process is started. We rely on the the microprocessor's branch-misprediction recovery mechanism to recover from errors. The recovery mechanism that we use is illustrated in Figure 1. Before the checker triggers a pipeline flush, we let all the instructions which are older than the offending instruction (i.e., the instruction that led to the error) commit. Thus, after the flush, the first instruction to enter the ROB will be the one that failed its check.

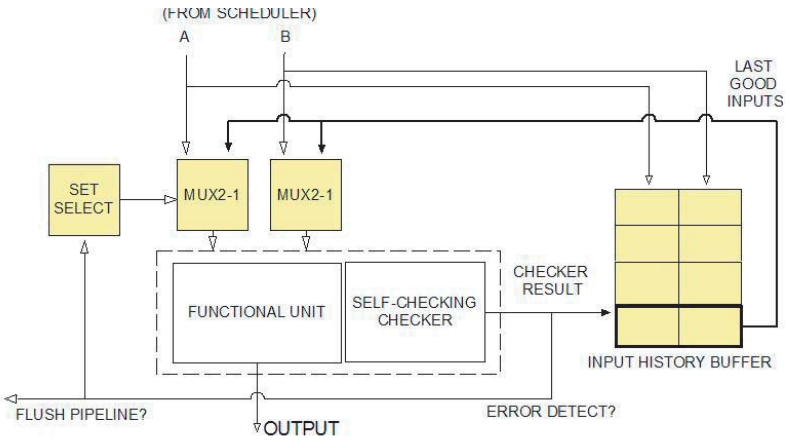

Figure 2. Block diagram of Diagnosis for Functional Units

4 Run-time Diagnosis of Hard Delay Faults

In this section, we describe our diagnosis mechanism in more detail. Although we discuss only integer adders and multipliers, the diagnosis aspect of our research applies just as easily to any other FU. If the fault is hard, we must diagnose it (Section 4.1). After diagnosis, we may choose to perform some limited reconfiguration (Section 4.2).

4.1 Diagnosing Hard Delay Faults (and Stuck-At Faults)

We now explain how we diagnose hard faults — both hard stuck-at and hard delay faults. Our explanation focuses on the diagnosis of hard delay faults, since that is a new contribution, but our scheme also diagnoses hard stuck-at faults at the same time.

Intuition A delay fault, unlike a stuck-at fault, is activated by an input transition rather than by a specific input. Thus, diagnosing a delay fault for an FU requires replaying a specific transition from one set of inputs to another set of inputs. We can replay an input transition by recording the last inputs to the FU and, when an error is detected, applying these inputs to the FU before re-trying the inputs that triggered the error. If the error re-occurs after replaying the input transition, we have an indication of a possible hard fault — replaying the input transition is sufficient for exercising both a delay fault or a stuck-at fault (we only need the second input for stuck-at faults, but we need both for delay faults).

When an error is encountered, it may be either due to a transient fault or due to a hard fault. Upon the detection of an error, we first try flushing the pipeline and recovering to the instruction that triggered the error. We also increment a 2-bit saturating error counter associated with that FU. This counter is reset upon any successful execution of that FU. Before restarting the pipeline, we restore the state of the FU by applying its previous inputs. If the error is caused by a transient fault, there will be a successful re-execution before the counter is saturated, and the instruction execution will continue. If the counter saturates, we conclude that the error is due to a hard fault.

Implementation and Operation Figure 2 shows a high level block diagram of our fault diagnosis scheme. Each FU keeps a history of its recent inputs in its own FIFO Input History Buffer (IHB). The IHB should keep the most recent L+1 inputs that have been sent to the checker. L is equal to the number of clock cycles in which the checker completes an instruction check. For example, if an adder checker completes a check in 3 clock cycles (2 cycles for the addition and one cycle for the comparison to the result of the primary adder), the size of the IHB should be 4. In this case, the IHB keeps the input sets of the last three unchecked additions, as well as the input set of the last checked addition. Each entry of the IHB keeps the two inputs of the adder. When an addition is checked, its input set is pushed to the end of the IHB and marked as the last good input set. The flow of events is shown in Table 1.

Furthermore, each checker must buffer the old results produced by the primary FU if the checker is slower than the primary FU. Using the same example, if the adder checker takes 2 clock cycles for the addition operation, there is a delay of one clock cycle between the adder checker and the primary adder, as shown in Table 1. The primary adder results arrive at the checker's comparator in the cycle after the primary adder finishes execution. As a result, the adder result buffer needs to keep only one result at a time. If the checker works as fast as the primary FU, buffering the old results is not necessary.

Table 1. Execution of Adder with Diagnosis

	Primary Adder		Adder Checker				
cycle	**input**	**output**	**adder inputs**	**adder output**	**comparator inputs**	**checked**	**last good inputs**
0	A0+B0	-	A0+B0	-	-	-	-
1	A1+B1	S0	A1+B1	-	-	-	-
2	A2+B2	S1	A2+B2	S0'	S0 and S0'	-	-
3	A3+B3	S2	A3+B3	S1'	S1 and S1'	A0+B0=S0	A0 and B0
4	A4+B4	S3	A4+B4	S2'	S2 and S2'	A1+B1=S1	A1 and B1
5	A5+B5	S4	A5+B5	S3'	S3 and S3'	A2+B2=S2	A2 and B2

If the comparison check is not successful, we first start the recovery mechanism as described in Section 3.3. Second, we apply the last good input set to the primary FU before any other input can be applied. We do not increment or clear the error counters during this FU operation. Replaying the last good input set puts the primary FU back in the state that led to a potential delay error. Then, the scheduler sends the offending instruction (the instruction that caused the error) to the FU. We add a small amount of logic to enable the scheduler to choose an FU based on feedback from an FU checker. If no error is detected this time, we clear the error counter and execution proceeds normally. If an error is detected, we increment the error counter and re-try the process again, starting with the pipeline flush and applying the last good input set to the primary FU.

If there is a hard fault in a primary FU or in its checker - either a delay fault or stuck-at fault - the error counter for that FU will saturate.

4.2 Post-Diagnosis Reconfiguration

Once we have diagnosed that there is a hard fault within an FU combination (a primary FU and its checker), we have a few options for dealing with it. The simplest solution is to deconfigure the FU — we mark it as permanently busy in the scheduler and prevent it from being used again. However, particularly for singleton units, such as the multiplier, it may be preferable to determine whether the primary FU or the checker is faulty, and continue executing without being checked. Implementing a self-checking checker [25, 28] is sufficient in this case.

5 Evalution

We have two goals for this experimental evaluation. First, we want to show that our scheme diagnoses injected delay faults and that it is indeed necessary for diagnosing delay faults; that is, without it, many of them would go undiagnosed and potentially lead to catastrophic chip failure. Second, we want to evaluate the hardware and power costs of our proposed mechanisms.

5.1 Experimental Methodology

To evaluate our design, we used a modified version of a cycle-accurate architectural simulator, SimpleScalar [3]. Table 2 shows the details of our configuration, which was chosen to be similar to that of the Intel Pentium4 [7, 13].

For benchmarks, we use the complete SPEC2000 benchmark suite with the reference input set. To reduce simulation time, we used SimPoint analysis [22] to sample from the execution of each benchmark.

5.2 Selected Functional Units

To evaluate our mechanism, we implemented a 32-bit CLA adder and a pipelined 32-bit Booth Encoded Multiplier in both Verilog and HSpice (using 0.18μm process parameters). The operational latencies of the functional units are 1 clock cycle and 4 clock cycles, respectively. We inserted 2x1 multiplexors in front of the functional units to enable them to select inputs either from the scheduler or from the IHB. There is also a "Set Select" logic, which simply changes the select signal of the multiplexors depending on the checker result.

The added multiplexors cause around 2% delay overhead per cycle for the multiplier and 2.7% delay for the adder, and this extra delay may require increasing the clock period by up to this amount.

5.3 Diagnosability of Delay Faults

In this experiment, we inject delay faults and show that our mechanism diagnoses them. We also determine how many delay faults would not be diagnosed in a system that did not have our mechanism. We focus on the adder, since we need to be able to simulate it at the transistor level of detail, and this is overly time-consuming for the multiplier. Since the results of our experiment depend on the functional input transitions of the adder, we use the benchmark behavior as the basis for our analysis.

First, for each SPEC benchmark, we use SimpleScalar to collect traces of additions performed at each of the three adders in the microprocessor. The total number of additions across all three traces is 50,000. These additions include

Table 2. Parameters of the Target System

Feature	Details
pipeline stages	20
width: fetch/issue/commit/check	3/6/3/3
branch predictor	2-level G-Share, 4K entries
instruction fetch queue	64 entries
reservation stations	32 entries
reorder buffer	128 entries
load/store queue	48 entries
integer ALU	3 units, 1-cycle
integer multiply/divide	1 unit, 14-cycle mult, 60-cycle div
floating point ALU	2 units, 1-cycle
floating point multiply/divide	1 unit, 1-cycle mult, 16-cycle div
L1 I-Cache	16KB, 8-way, 64-byte blocks, 2-cycles
L1 D-Cache	16KB, 8-way, 64-byte blocks, 2-cycles
L2 (unified)	1MB, 8-way, 128-byte blocks, 7-cycles

those performed for add instructions as well as for other instructions, such as computing an address to be loaded. For each of the 50,000 additions, the trace records the input operands and the time (cycle count) at which that addition occurred.

Then, in each of 50 separate experiments, we injected a single delay fault in a randomly-chosen gate within the adder in the Verilog implementation. The delay fault causes an extra delay that is always long enough to exceed the timing constraints, but it could still be masked (e.g., if the delay fault affects a gate that is on a path that is not switching). We simulated the operation of each faulty adder executing the three traces, and these simulations were performed with a Verilog simulator at the gate level, using gate timing values produced by HSpice simulation.

For each addition that is erroneous due to a delay fault, we explore the behavior of both our diagnosis mechanism as well as a system without our diagnosis mechanism. With our diagnosis mechanism, we can always restore the prior state of the adder since we directly read this state from the IHB. Having restored the prior input state, we replay the transition for which the delay fault caused an error. Thus, our mechanism diagnoses all un-masked delay faults. Without our mechanism, the diagnosis of the delay fault is not guaranteed since it depends on what state the adder input was left at the time of error detection. Our experiment can be explained by a simple example: Assume an instruction order as given in Table 1. Assume that there is an error in the addition for input operands A1+B1. Thus, there is a fault that manifests itself when the adder inputs transition from A0+B0 to A1+B1. Also assume that the error due to this fault is detected at cycle 4 (i.e., the checker has a 3-cycle latency). Based on this scenario, just before the error is detected, operands A4+B4 are at the input of the adder. With our diagnosis mechanism, we first set the adder inputs to A0+B0 before replaying the error-causing instruction, ensuring that the inputs transition from A0+B0 to A1+B1, thus re-generating the same input transition that caused the error. Without our diagnosis mechanism, the inputs will transition from A4+B4 to A1+B1. While this transition may also result in the delay fault manifestation, this outcome is not guaranteed. In order to determine how often the delay fault is not diagnosed, for every addition in the trace, we first set the input signals of the adder to the pattern at the time of error detection and then re-play the erroneous addition. If this addition executes correctly during replay, the injected delay fault cannot be diagnosed without our mechanism. In Figure 3, we show the results of this experiment, averaged over all three adders in the microprocessor. We observe that, for all benchmarks, the vast majority of delay faults would not be diagnosed without our mechanism.

5.4 Hardware and Power Costs

In this section, we evaluate the hardware and power costs of our delay fault diagnosis mechanism. Because we cannot possibly simulate every possible input transition for the adder and multiplier and because HSpice simulations take so long, our power results are based on 3000 simulations with different random input transitions. The power results are averaged across all of the time for all of the simulations for each FU, and they include dynamic and static (leakage) power.

Since we assume that there is a built-in error detection mechanism, we do not consider the cost of the checker. The cost of our mechanism consists of the IHB, a small control logic, the multiplexors, and the ”Set Select” logic. The major cost factor is the IHB. The area and power overhead of the IHB depends on the operational latency of the FU,

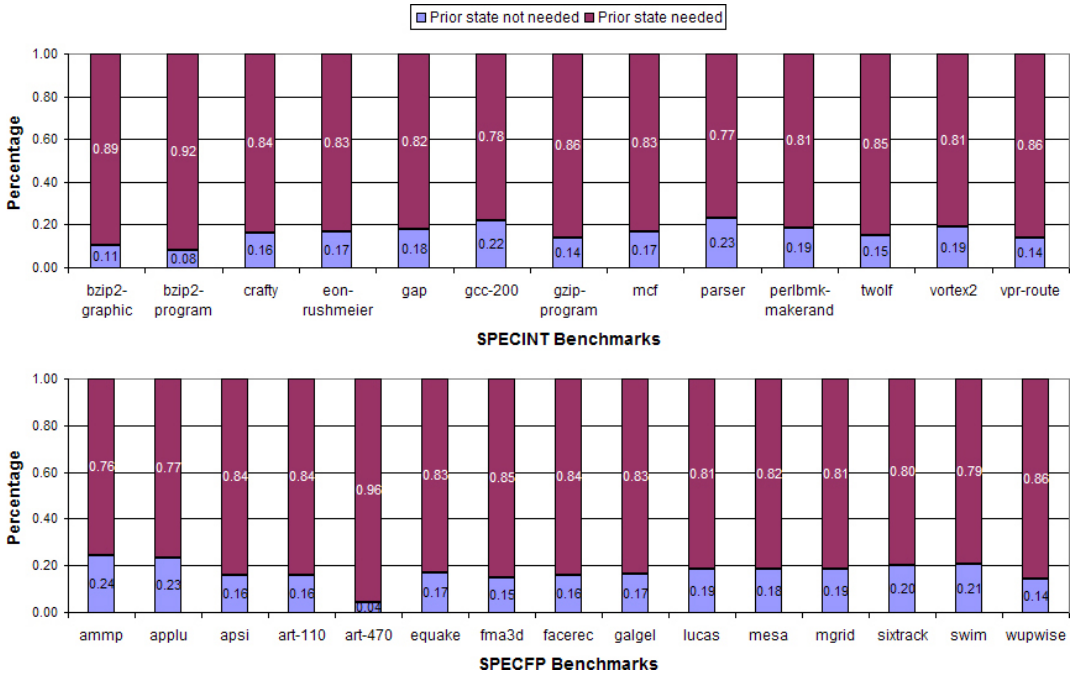

Figure 3. How often our diagnosis mechanism is needed

the latency of the checker (L), and the input operand width (e.g., 32-bit, 64-bit).

Table 3 summarizes the overhead results for our implementation of the 32-bit integer adder and the 32-bit multiplier. The overheads for other functional units can be estimated considering the relative size of the FU to the adder or the multiplier. We approximate the area of a module in terms of the number of transistors. We assumed that we use A-phase 10-transistor latches for our IHB while calculating these numbers.

As can be seen in Table 3, the size of the IHB compared to the primary adder increases with increasing L. The total area overhead is around 80% for L=3, and 62% for L=2. In order to keep the area and power overhead of the IHB low, a checker which can run at least as fast as the primary adder can be selected. For the multiplier, the area overhead is lower because of the larger size of the multiplier compared to the adder. The IHB for the multiplier has more entries because the primary multiplier has a latency of 4 cycles (L=5). The total area overhead for the multiplier is 12.6%. We could reduce the amount of buffering, if we were willing to change the design such that the scheduler provided the last inputs. However, we decided against that design, because it would increase the complexity of the scheduler and require us to add a new wiring path between the scheduler and each FU. The power overhead results correspond closely to the area overhead results.

6 Conclusions

We have developed a low-cost mechanism for diagnosing hard delay faults in functional units of a microprocessor. Our scheme uses already existing error detection mechanisms and leverages the built-in recovery mechanism in the microprocessor. Our diagnosis mechanism requires just a small amount of hardware to buffer previous functional unit state in order to re-create the switching sequence that led to the error. We showed that our diagnosis mechanism can diagnose all injected delay faults and that previous diagnosis schemes, which target stuck-at faults, miss the majority of delay faults. We believe that the low cost — in terms of hardware, power, and performance — makes our approach viable for this important problem.

References

[1] N. Ahmed, M. Tehranipoor, and V. Jayaram. Timing-based Delay Test for Screening Small Delay Defects. In *Proc. of IEEE DAC*, pages 320–325, Jul 2006.

[2] T. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proc. of IEEE/ACM MICRO*, pages 196–207, Nov 1999.

[3] T. Austin, E.Larson, and D.Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *IEEE Computer*, 35(2):59–67, Feb 2002.

[4] A. Avellan and W. Krautschneider. Impact of Soft and Hard Breakdown on Analog and Digital Circuits. *IEEE Tran. on Device and Materials Reliability*, 4(4):676–680, Dec 2004.

Table 3. Hardware and Power Overheads for Adder and Multiplier

Module	Size (# of tran.)	% of primary FU	Power(mW)	% of primary FU
Primary Adder	3488	100	1.56	100
IHB (L=2 / L=3)	1922/2562	55/73	0.9/1.2	57/77
Muxes and select logic	258	7.4	$<$0.001	negligible
Primary Multiplier	32633	100	27.6	100
IHB	3842	11.8	1.9	6.9
Muxes and select logic	258	0.8	$<$0.001	negligible

[5] M. Azimane and A. Majhi. New Test Methodology for Resistive Open Defect Detection in Memory Address Decoders. In *Proc. of IEEE VTS*, pages 123–128, Apr 2004.

[6] J. Blome, S. Feng, S. Gupta, and S. Mahlke. Online Timing Analysis for Wearout Detection. In *Workshop on Architectural Reliability*, Nov 2006.

[7] D. Boggs, A. Baktha, J. Hawkins, D. Marr, J. Miller, P. Roussel, R. Singhal, B. Toll, and K. Venkatraman. The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology. *Intel Technology Journal*, 8(1), Feb 2004.

[8] F. A. Bower, P. G. Shealy, S. Ozev, and D. J. Sorin. Tolerating Hard Faults in Microprocessor Array Structures. In *Proc. of IEEE Int. Conf. on Dependable Systems and Networks*, pages 51–60, Jun 2004.

[9] F. A. Bower, D. J. Sorin, and S. Ozev. A Mechanism for Online Diagnosis of Hard Faults in Microprocessors. In *Proc. of IEEE/ACM MICRO*, pages 197–208, Nov 2005.

[10] J. R. Carter, S. Ozev, and D. J. Sorin. Circuit-Level Modeling for Concurrent Testing of Operational Defects due to Gate Oxide Breakdown. In *Proc. of IEEE DATE*, pages 300–305, Mar 2005.

[11] K. Y. Chung and S. Gupta. Low-Cost Scan-Based Delay Testing of Latch-Based Circuits with Time Borrowing. In *Proc. of IEEE VTS*, pages 8–15, Apr 2006.

[12] D. Ernst, N. S. Kim, S. Das, S. Pant, R. Rao, T. Pham, C. Ziesler, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *Proc. of IEEE/ACM MICRO*, pages 7–18, Dec 2003.

[13] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Intel Pentium 4 Processor. *Intel Technology Journal*, Feb 2001.

[14] R. Kaivola and N. Narasimhan. Formal Verification of the Pentium4 Floating-Point Multiplier. In *Proc. of IEEE DATE*, pages 20–27, May 2002.

[15] J. Li, C.-W. Tseng, and E. McCluskey. Testing for Resistive Opens and Stuck Opens. In *Proc. of IEEE ITC*, pages 1049–1058, Oct 2001.

[16] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Implementation of Redundant Multithreading Alternatives. In *Proc. of IEEE ISCA*, pages 99–110, May 2002.

[17] S. Natarajan, S. Patil, and S. Chakravarty. Path Delay Fault Simulation on Large Industrial Designs. In *Proc. of IEEE VTS*, pages 16–23, Apr 2006.

[18] S. Oberman. Floating Point Division and Square Root Algorithms and Implementation in the AMD-K7TM Microprocessor. In *Proc. of IEEE Symposium on Computer Arithmetic*, pages 106–115, Apr 1999.

[19] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proc. of IEEE ISCA*, pages 25–36, Jun 2000.

[20] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proc. of IEEE FTCS*, pages 84–91, Jun 1999.

[21] J. Saxena, K. Butler, J. Gatt, R. Raghuraman, S. Kumar, S. Basu, D. Campbell, and J. J. Berech. Scan-Based Transition Fault Testing - Implementation and Low Cost Test Challenges. In *Proc. of IEEE ITC*, pages 1120–1129, Oct 2002.

[22] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proc. of Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct 2002.

[23] L. Spainhower and T. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(5/6), Sep 1999.

[24] N. Tendolkar, R. Molyneaux, C. Pyron, and R. Raina. At-speed Testing of Delay Faults for Motorola's MPC7400, A PowerPC microprocessor. In *Proc. of IEEE VTS*, pages 3–8, 2000.

[25] D. P. Vasudevan and P. Lala. A Technique for Modular Design of Self-checking Carry-Select Adder. In *Proc. of IEEE DFT*, pages 325–333, Oct 2005.

[26] T. Vijaykumar, I.Pomeranz, and K. Chung. Transient Fault Recovery Using Simultaneous Multithreading. In *Proc. of IEEE ISCA*, pages 87–98, May 2002.

[27] S. Wang and S. Chakradhar. A Scalable Scan-path Test Point Insertion Technique to Enhance Delay Fault Coverage for Standard Scan Designs. *IEEE Tran. on CAD*, 25(8):1555–1564, Aug 2006.

[28] M. Yilmaz, D. Hower, S. Ozev, and D. J. Sorin. Self-Checking and Self-Diagnosing 32-bit Microprocessor Multiplier. In *Proc. of IEEE ITC*, Oct 2006.