A Mechanism for Online Diagnosis of Hard Faults in Microprocessors

Fred A. Bower[1,3], Daniel J. Sorin[2], and Sule Ozev[2]

[1]Department of Computer Science, Duke University

[2]Department of Electrical and Computer Engineering, Duke University

[3]IBM, Research Triangle Park

Abstract

We develop a microprocessor design that tolerates hard faults, including fabrication defects and in-field faults, by leveraging existing microprocessor redundancy. To do this, we must: detect and correct errors, diagnose hard faults at the field deconfigurable unit (FDU) granularity, and deconfigure FDUs with hard faults. In our reliable microprocessor design, we use DIVA dynamic verification to detect and correct errors. Our new scheme for diagnosing hard faults tracks instructions' core structure occupancy from decode until commit. If a DIVA checker detects an error in an instruction, it increments a small saturating error counter for every FDU used by that instruction, including that DIVA checker. A hard fault in an FDU quickly leads to an above-threshold error counter for that FDU and thus diagnoses the fault. For deconfiguration, we use previously developed schemes for functional units and buffers, and we present a scheme for deconfiguring DIVA checkers. Experimental results show that our reliable microprocessor quickly and accurately diagnoses each hard fault that is injected and continues to function, albeit with somewhat degraded performance.

1 Introduction

As technological trends continue to lead toward smaller device and wire dimensions in integrated circuits, the probability of hard (permanent) faults in microprocessors increases. These faults may be introduced during fabrication, as defects, or they may occur during the operational lifetime of the microprocessor. Well-known physical phenomena that lead to operational hard faults are gate oxide breakdown, electromigration, and thermal cycling. Microprocessors become more susceptible to all of these phenomena as device dimensions shrink [28], and the semiconductor industry's roadmap has identified both operational hard faults and fabrication defects (which we will collectively refer to as "hard faults") as critical challenges [13]. In the near future, it may no longer be a cost-effective strategy to discard a microprocessor with one or more hard faults, which is what, for the most part, we do today.

Traditional approaches to tolerating hard faults have masked them using macro-scale redundancy, such as triple modular redundancy (TMR). TMR is an effective approach, but it incurs a 200% overhead in terms of hardware and power consumption. There are some other, lightweight approaches that use marginal amounts of redundancy to protect specific portions of the microprocessor, such as the cache [36, 18] or buffers [5], but none of these are comprehensive.

Our goal in this work is to create a microprocessor design that can tolerate hard faults without adding significant redundancy. The key observation, made also by previous research [25, 27, 29], is that modern superscalar microprocessors, particularly simultaneously multithreaded (SMT) microprocessors [32], already contain significant amounts of redundancy for purposes of exploiting ILP and enhancing performance. We want to use this redundancy to mask hard faults, at the cost of a graceful degradation in performance for microprocessors with hard faults. In this paper, we do not consider adding extra redundancy strictly for fault tolerance, because cost is such an important factor for commodity microprocessors. The viability of our approach depends only on whether, given a faulty microprocessor, being able to use it with somewhat degraded performance provides any utility over having to discard it.

To achieve our goal, the microprocessor must be able to do three things while it is running.

- It must detect and correct errors caused by faults (both hard and transient).
- It must diagnose where a hard fault is, at the granularity of the field deconfigurable unit (FDU).
- It must deconfigure a faulty FDU in order to prevent its fault from being exercised.

Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)
0-7695-2440-0/05 $20.00 © 2005 IEEE

While previous work in this area has explored aspects of this problem, none has developed an integrated solution. Some work has used deconfiguration to tolerate strictly fabrication defects and thus assumed pre-shipment testing instead of online error detection and diagnosis [25]. Other work has explored deconfiguration and has left detection and diagnosis as open problems [29].

In this paper, we discuss integrated design options for microprocessors that achieve all three of these goals, and we present one particular microprocessor in this design space. First, our microprocessor detects and corrects errors, due to both transient faults and hard faults, using previously developed DIVA-style [2] dynamic verification. Second, it uses a newly developed mechanism to diagnose hard faults as the system is running. Third, after diagnosing a hard fault, the microprocessor deconfigures the faulty FDU in an FDU-specific fashion. In this paper, we present and evaluate previously developed deconfiguration schemes for functional units and portions of array structures (e.g., reorder buffer, load/store queue, etc.), and we show that our integrated approach also enables the microprocessor to deconfigure faulty DIVA checkers.

Our experimental results show that our new diagnosis mechanism quickly and accurately diagnoses hard faults. Moreover, our reliable microprocessor can function quite capably in the presence of hard faults, despite not using redundancy beyond that which is already available in a modern microprocessor. This technique can turn otherwise useless microprocessors into microprocessors that can function at a gracefully degraded level of performance. This capability can improve reliability by tolerating operational hard faults. We can improve yield by shipping microprocessors with defects that we have tolerated—it is as if they are regular microprocessors that will get "binned" into a lower performance bin. Although binning is typically by clock frequency, recent proposals have suggested more general performance binning [25]. As long as these bins are not so low-performing as to be useless, then our improvement in yield is a benefit. Our scheme also vastly outperforms a system with only DIVA or a comparable recovery-based scheme, since the performance cost of recoveries is quite high for hard faults that get exercised frequently; moreover, our scheme can tolerate a hard fault in a DIVA checker.

The contributions of this work are:

- A dynamic, comprehensive hardware mechanism for diagnosing hard faults in microprocessors, including faults in DIVA checkers.
- A microprocessor design that integrates our new hard fault diagnosis mechanism with DIVA error detection and a mix of pre-existing and new deconfiguration schemes.
- An experimental evaluation that demonstrates that a microprocessor with our enhancements can tolerate hard faults with a graceful degradation in performance.

In Section 2, we discuss hard faults and why they concern microarchitects. In Sections 3, 4, and 5, we describe error detection and correction, hard fault diagnosis, and deconfiguration of faulty components, respectively. Section 6 discusses the costs and limitations of our particular implementation. Section 7 presents our experimental evaluation. We discuss related work in Section 8 and conclude in Section 9.

2 Hard Faults in Microprocessors

In this section, we discuss the hard faults that motivate this work. In particular, we focus on the technological trends that are leading towards greater incidences of these faults. With increasingly smaller device and wire dimensions and higher temperatures, these trends lead us to conclude that hard fault rates will increase.

There have been several recent studies of operational hard faults [28, 14], that is, hard faults that occur over the lifetime of the microprocessor. Srinivasan et al. [28] determine that electromigration [31, 3] and gate oxide breakdown [10] are likely to be the two dominant phenomena that cause operational hard faults. Electromigration results in highly resistive interconnects or contacts and eventually leads to open circuits. Electromigration increases as wire dimensions shrink and as temperatures increase. Gate oxide breakdown (OBD) results in the malfunction of a single transistor due to the creation of a highly conductive path between its gate and its bulk. A newly manufactured oxide contains inherent electron traps due to imperfections in the fabrication process. Over the lifetime of the device, the number of such traps increases due to electric field stress and electron tunneling. At some point, the electron traps may line up and constitute a conductive path between the gate and the bulk of the device, eventually leading to OBD. OBD rates increase as oxide thicknesses shrink and temperatures increase. Since OBD increases switching delay, it can lead to delay faults that manifest themselves as bit flips [6].

Defects introduced during chip fabrication are another source of hard faults. Their causes differ from those of operational hard faults, but they often manifest themselves in a similar fashion. For example, a fabrication defect could result in a discontinuity in a wire,

which is equivalent to the situation in which electromigration leads to an open circuit. A fabrication defect could also lead to the growth of an insufficiently thick gate oxide, which is functionally equivalent to OBD. The impact of technology trends on fabrication defects is less clear than it is for operational faults. In general, though, smaller wire and device dimensions are more prone to defects, since the margin for error is smaller.

3 Error Detection and Correction

There are numerous ways to detect and correct errors in microprocessors. For our target design space, the best error detection candidates are the recently developed techniques that are both comprehensive (i.e., not tailored to one specific error model) and less costly than macro-scale redundancy (e.g., TMR). We do not claim to innovate in this area; we simply seek to use a pre-existing solution that is well-suited to our diagnosis and deconfiguration mechanisms.

We choose DIVA to comprehensively detect and correct errors using dynamic verification with checker processors [2]. In a system with DIVA dynamic verification, a total of *k* checkers are added at the commit stage of the typical *k*-way superscalar processor pipeline. These checkers are small, simple, in-order cores. According to Weaver and Austin [34], a checker's size is less than 6% of an Alpha 21264 core, which is far less than the 200% overhead of TMR. These checkers re-execute each instruction and compare their results with those of the superscalar core. The original DIVA paper [2] assumes that the checkers, because of their small size, can be made resilient to physical faults; thus, a mismatch in the result of an instruction signifies an error in the superscalar core and leads the checker to correct the error by committing its results and squashing the superscalar pipeline.

In the original DIVA design, a hard fault in a checker is undetectable and uncorrectable—this is a limitation that we overcome later in this paper by detecting and diagnosing hard faults in checkers, so that a system can stop producing erroneous results and, if backward error recovery (BER) is available, recover from erroneous data that was committed before the checker was diagnosed as faulty.

Other options besides DIVA exist, such as redundant multithreading, and they present different engineering tradeoffs. A thorough discussion of all of the alternatives is outside the scope of this paper, but we provide a summary of alternatives and their capabilities in Section 8. We chose DIVA over the alternatives because the opportunity cost and power consumption of using the alternatives exceeded the small amount of overhead introduced by DIVA. We also believe that DIVA checkers offer better hard fault correction capability. Detailed studies of the implementation of DIVA dynamic verification have shown it to provide performance nearly on par with an unprotected processor in the error-free case, with minor performance degradation until error rates reach the error-per-thousand-instruction range [2].

4 Fault Diagnosis

DIVA checkers do not provide fault diagnosis. They are only capable of detecting and correcting errors, not determining their underlying causes. For transient faults, this is appropriate, since the desired remedy never involves altering the configuration of the core. For hard faults, however, we show in Section 7 that it is often desirable to deconfigure part of the superscalar core in order to prevent frequent errors and the performance penalty that frequent pipeline flushes from DIVA corrections (or redundant thread corrections) would require.

We define sub-structures within the processor core that we wish to be able to deconfigure as field deconfigurable units (FDUs). To diagnose hard faults in the processor core, we first have to select the FDU granularity at which we wish to be able to diagnose. Many structures are replicated within a typical superscalar core, and the granularity of replication represents a natural FDU granularity.

The choice of FDU is a design decision for a given implementation. For the processor we model in our evaluation, the identified FDUs for which we track diagnosis information are: individual entries in the instruction fetch queue (IFQ), individual reservation stations (RS), individual entries in the load-store queue (LSQ), individual entries in the re-order buffer (ROB), individual arithmetic logic units (ALU), and the individual DIVA checkers. We have chosen a fairly fine FDU granularity, but one could choose coarser or even finer granularities if so desired; we discuss this engineering tradeoff later. The hardware bounds of our diagnosis mechanism are the components in which the selected error checker (in our design, DIVA) can detect a fault. Therefore, we do not consider the register file, because DIVA cannot recover from errors in it.

4.1 A New Online Diagnosis Mechanism

We propose in this paper to dynamically attribute errors to FDUs as the system is running. Given an error detection mechanism, if an instruction (or micro-op, in the case of IA-32) is determined to be in error, the system records which FDUs that instruction used during its lifetime. If, over a period of time, more than a pre-specified threshold of errors has been attributed to a given FDU, it is very likely that this resource has a hard fault.

To track each instruction's FDU usage, bits are carried with each instruction from the point of FDU usage to commit. For those structures that the instruction owns at commit, this information is already implicitly available and no extra wires are needed to carry this resource usage info through the pipeline. In our modeled processor, the ROB entries and DIVA checkers use implicit tracking. For the remaining FDUs, the number of bits required is a function of the size of the structure and the granularity into which we are allowing it to be subdivided for later deconfiguration. This represents an engineering trade-off in our design that will allow implementations to select the appropriate FDU granularity/overhead trade-off. With the configuration used in our evaluation in Section 7, each instruction carries 19 bits of usage information: 5 bits for RS, 6 bits for LSQ, 6 bits for IFQ, and 2 bits for ALUs. Carrying these extra bits through the pipeline incurs two costs: pipeline latches will be marginally wider and there will be more wires to route through the pipeline. However, compared to the 64-bit operands that are carried through the pipeline, these extra 19 bits are a small addition, especially since not all 19 bits need to traverse the whole pipeline. For each FDU we track, the processor maintains a small, saturating error counter.

There are four challenges with this approach. First, after the FDUs have been selected and configured for diagnosis in an implementation of our mechanism, all remaining logic for which the checker detects errors must also be tracked by our diagnosis scheme. For our design, this critical logic includes all logic that is not within an FDU but that is in the portion of the superscalar core for which DIVA is capable of detecting errors. This includes instruction issue, any singleton arithmetic logic units (ALUs) (for example, a floating point multiply/divide unit), floating point ALUs, and any common datapaths that all instructions must traverse while in-flight.

Second, transient errors must not lead to above-threshold error rates. Thus, we must have error counter thresholds that are not too small, and the microprocessor must periodically clear the error counters to prevent transient errors from accumulating past the hard fault threshold. The frequency of counter clearing is an adjustable parameter that depends on expected transient error rates. Counter clearing is a low-cost operation, so we choose to clear the counters once per second in our experiments, even though current terrestrial transient fault rates do not approach this frequency. Also, if a hard fault is detected and deconfiguration is activated, the deconfiguration process clears the error counters.

Third, the error rate threshold for a resource must be related to its usage. For example, a very high threshold for a resource that is rarely used will preclude the system from ever diagnosing a hard fault in it. Thus, for frequently utilized FDUs, a larger counter value is required to prevent the mis-diagnosis of a fault in an upstream or downstream structure. In Table 1, we list the counter thresholds for the FDUs we consider in this paper. For resources that are very rarely used, such as the floating point units, our mechanism might never be able to diagnose hard faults in them. However, any hard fault that gets exercised so rarely as to not exceed our error counter threshold is also so rare that it incurs little performance penalty for its infrequent error recoveries. In this situation, simply using DIVA to correct errors due to a hard fault in a rarely used FDU is sufficient. Results (not shown due to space constraints) confirm that, even for the SPEC floating point benchmarks, a faulty FPU does not significantly degrade performance. Thus, we do not consider FPUs to be FDUs. The key observation is that our scheme can diagnose hard faults in the highly utilized resources, so that the microprocessor avoids frequent recoveries.

Table 1. Error counter thresholds

FDU	threshold	quantity	total bits used
instruction fetch queue entry	32	64	320
reservation station	32	32	160
reorder buffer entry	16	128	512
load/store queue entry	16	48	192
integer ALU	64	3	18
DIVA checker	64	3	18
critical logic (issue, etc.)	128	1	7
			1227

The final challenge is that the resources must be used reasonably independently. Otherwise, for example, if every time an instruction uses resource A it also uses resource B, then the diagnosis mechanism will not be able to distinguish between a hard fault in A and a hard fault in B. To guarantee that instructions take many different and independent paths through the pipeline, we slightly change the scheduling of resources that are normally scheduled non-uniformly (e.g., higher priority for ALU_0) to add a "round-robin" aspect to it. For example, instead of always allocating the lowest-numbered ALU that is available, the microprocessor allocates available ALUs in a round-robin fashion. Otherwise, the usage of ALU_0 could be significantly greater than that of other ALUs and thus preclude hard faults in them from being diagnosed (since the thresholds assume uniform utiliza-

tion). This scheduling modification is not necessary for resources that are naturally scheduled uniformly, like ROB entries. We found that round robin scheduling alone does not avoid all lockstep allocation of resources, though. For example, with three ALUs and three DIVA checkers, we found that a long string of instructions that all used ALUs led to undiagnosable errors. In one particular scenario, an instruction that used ALU_0 always used $Checker_1$, ALU_1 was perfectly correlated with $Checker_2$, and ALU_2 was perfectly correlated with $Checker_0$. To avoid this lockstep allocation, we introduced a small amount of pseudo-randomness into the scheduling of checkers. Every cycle, the first checker to be considered for allocation is determined based on pseudo-random data (e.g., low order bits of the tick counter), and then subsequent checkers are allocated sequentially (mod 3) after the first one. This pseudo-randomness, combined with round-robin scheduling, prevents lockstep allocation and achieves reasonably uniform utilization of each set of identical FDUs.

We include the DIVA checkers in the error diagnosis design, so that we can enable the microprocessor to tolerate hard faults in the checkers. Since a *k*-way superscalar microprocessor requires approximately *k* checkers to avoid having the checkers become a bottleneck, we would like to be able to tolerate a hard fault in one of them by leveraging their redundancy.

Using DIVA for error detection and correction provides three unique issues related to diagnosis and deconfiguration of a hard-faulted unit. First, uncached loads and stores commit without any redundant check of the operation, making them undiagnosable. A fault affecting the logic unique to these operations will not be covered by our mechanism. The system will perform exactly as it would if it only had DIVA checkers active. Second, the microprocessor is vulnerable to transient errors in DIVA checkers, but DIVA assumes that small checkers can be designed to be more resilient to transient faults by using more robust feature sizes. Third, because the microprocessor trusts a DIVA checker until its error counter exceeds its threshold, the microprocessor is vulnerable to incorrect execution in the window between when a hard fault occurs in a checker and when it diagnoses that the checker is the culprit. We further discuss this window of vulnerability in Section 6.2.

4.2 Alternative Design Options

There exist other ways to perform fault diagnosis. The most obvious approach is to use TMR—if two modules produce one result and the third module produces a different result, then the system diagnoses the third module as faulty (assuming a single-fault model). TMR, however, has a 200% hardware and power overhead.

Another well-known diagnosis approach is built-in self-test (BIST). After detecting an error and determining that it is due to a hard fault (e.g., by detecting it repeatedly), systems with dedicated BIST hardware can test themselves in order to diagnose the location of the hard fault. To its advantage, unlike our new diagnosis mechanism, BIST does not have to worry about the statistical nature of online error counting. BIST can be applied to a microprocessor like the ones we study, and one concurrent BIST mechanism can be used for all components in the path, although the number of BIST test vectors to generate—either deterministically or pseudo-randomly—would be extremely large. The BIST-based scheme cannot be applied to single modules such as the instruction queue or the decode logic, since taking these structures offline for testing would leave the microprocessor temporarily unable to function. Moreover, online error counting has the advantage over BIST of diagnosing faults via the observation of the execution of actual software and not needing to analyze test outputs. BIST also adds performance overhead due to the extra multiplexers that choose between normal inputs and BIST inputs.

5 Deconfiguring Faulty Components

After an FDU has been diagnosed as having a hard fault present, deconfiguring the faulty FDU is desired to avoid the frequent pipeline flushes that DIVA would trigger due to continued manifestation of the fault. In this section, we describe several pre-existing methods for deconfiguring typical microprocessor structures, plus a new way to deconfigure a faulty DIVA checker.

For circular access array structures—such as the instruction fetch queue (IFQ), reorder buffer (ROB), and load/store queue (LSQ)—previous work has shown how to add a level of indirection to allow for de-configuration of a single entry with little additional latency added to access time for the structure [5, 25]. In the method by Bower et al. [5], each structure maintains a fault map. This fault map information feeds into the head and tail pointer advancement logic, causing the advancement logic to skip an entry that is marked as faulty. If cold spares are available, as assumed by Bower et al. and shown in Figure 1, the structure size can be maintained at the original processor design point. If no spares are provisioned, which is what we assume in this paper, then the structure size must be updated when the fault map is updated.

For some tabular (i.e., directly addressed) structures—such as reservation stations, register files, etc.—a simple solution is to permanently mark the resource as

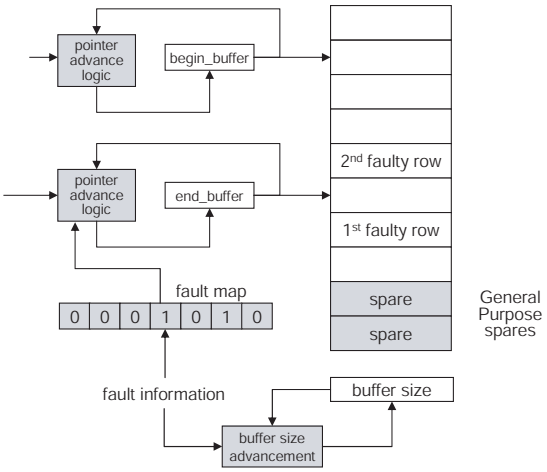

Figure 1. Deconfiguration of entries in a circular buffer (e.g., reorder buffer). Shading indicates hardware added for entry deconfiguration purposes.

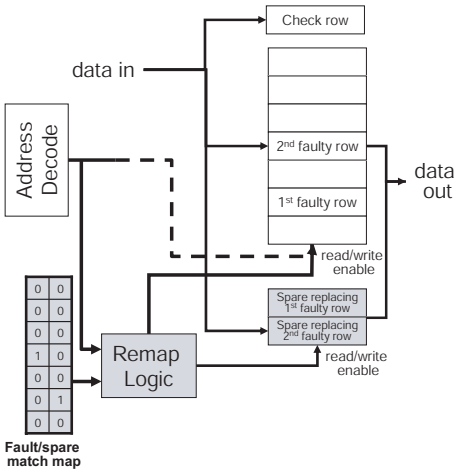

Figure 2. Deconfiguration of entries in a tabular structure (e.g., reservation station). Shading indicates hardware added for entry deconfiguration purposes.

in-use, thus removing it from further operation [25]. Once again, Bower et al. [5] assume that cold spares may be available, and we illustrate this previously developed design in Figure 2, even though we assume no provisioning of cold spares in this paper.

For a functional unit (ALU, etc.), similar to a reservation station, we can mark the resource as permanently busy, preventing further instructions from issuing to it [25]. Cold sparing of functional units is possible, but it may require too much die space, as functional units are relatively large compared to individual ROB entries or reservation stations. We focus on using existing redundancy, since the cost of adding extra redundancy may be too great for commodity microprocessors.

For one of the multiple DIVA checkers, we can map it out if we diagnose it as being permanently faulty. Depending on how DIVA checkers are scheduled, deconfiguration is just as simple as for ALUs; just marking a faulty checker as permanently busy will deconfigure it. Prior work has not looked into deconfiguring DIVA checkers, because no fault diagnosis schemes prior to this paper could diagnose hard faults in a checker.

6 Costs and Limitations

The design that we have presented in Sections 3-5 is not free, nor is it without limitations. In this section, we present its hardware costs and limitations.

6.1 Hardware Costs

We add hardware to an unprotected microprocessor to achieve hard fault tolerance. The largest, single addition to the processor is the DIVA checkers, each of which has been estimated at 6% of the size of an Alpha 21264 core [34]. In addition to DIVA, which provides benefits even without our additions, we also add: error counters (1227 bits total), wires for tracking each instruction's resource usage (19 wires in total), and logic for deconfiguring FDUs. None of these additional hardware costs are large; moreover, they can all be reduced at the expense of a coarser granularity of diagnosis and deconfiguration. For example, we can share one error counter and one wire among k entries in the instruction window, at the cost of having to deconfigure all k entries if any of them incurs a hard fault.

6.2 Limitations

We now discuss three limitations of our current implementation and approaches for addressing them in the future. First, there are certain structures that we either cannot protect or that are very difficult to protect. Our current implementation cannot protect the register file, because it is part of the recovery point for DIVA recovery. We cannot diagnose faults in singleton resources, due to ambiguity reasons stated at the end of Section 4.1. Singleton resources include issue logic, common datapath lines, and similar components. Singletons are always in lock-step scheduling with other singletons. Future work will involve designing modular implementations of these currently monolithic structures, so that incremental redundancy is feasible.

Second, there are certain scenarios in which the system can deconfigure a fault-free FDU. A transient or hard fault in our added hardware—error counters, wires for tracking resource usage, and deconfiguration logic—could lead to deconfiguring a fault-free component. In general, if deconfiguration does not help (i.e., immedi-

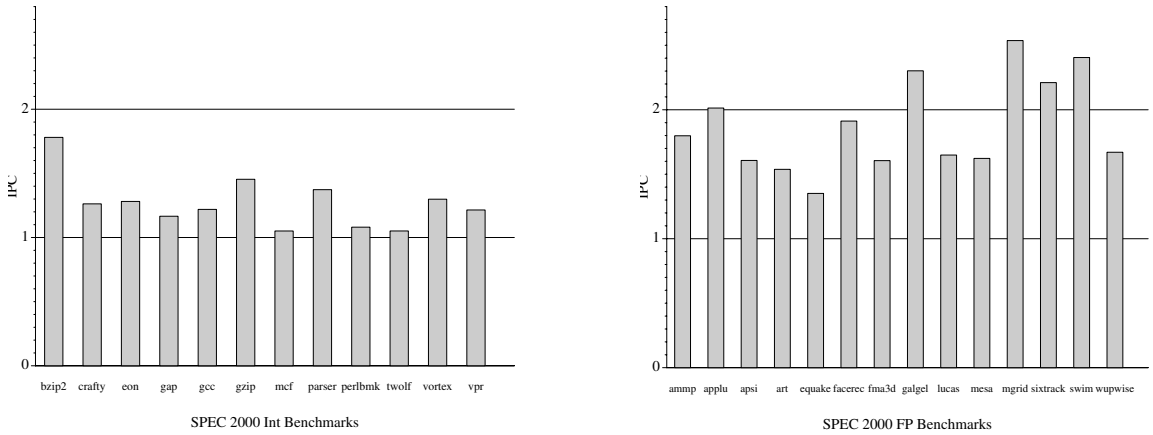

Figure 3. Error-free performance (SpecINT and SpecFP)

ately after deconfiguration, another error counter saturates), then the system can reconfigure the previously mapped out unit back into the system (under the common assumption of one hard fault at a time). The microprocessor also tolerates faults in the error counters by testing them. After clearing the counters, it checks that they are indeed all zero. It also uses a small amount of hardware to periodically test that the counters can be incremented correctly. If a counter is faulty, the corresponding FDU is then permanently either configured or deconfigured, based upon whether it is mapped back in or left deconfigured. Mapping it back in leaves the system vulnerable to a hard fault in this FDU, but leaving it deconfigured is potentially a loss of useful hardware.

Third, there is a window of vulnerability in which a faulty microprocessor can unwittingly produce erroneous results. Being able to deconfigure a faulty DIVA checker enables the microprocessor to improve reliability by preventing the fault from continuing to silently corrupt system state; in a DIVA-only system, it would go unnoticed until visible data corruption was recognized by a downstream entity. However, there is still a window of vulnerability between when the hard fault occurs in the checker and when it is diagnosed and deconfigured. In that window, a number of instructions equal to the error counter threshold for the checker times the number of DIVA checkers could have been committed in error, since DIVA checkers assume they are correct in the case of a mis-comparison. Without a higher-level recovery scheme, such as checkpointing, this erroneously committed state represents an unrecoverable error.

7 Evaluation

There are three goals of this evaluation. First, we want to show that our reliable microprocessor can quickly and correctly detect and diagnose hard faults, even in the presence of transient faults. Second, we want to demonstrate that, after our scheme deconfigures a permanently faulty FDU, the microprocessor's performance is still good enough to be useful. Third, we want to compare our scheme against a microprocessor that simply relies on DIVA checkers to tolerate hard faults; while DIVA was designed primarily for soft faults, it can also tolerate hard faults, and we want to determine if our scheme outperforms this simpler solution.

7.1 Methodology

To evaluate our design for proper operation under the fault models considered, we modified sim-mase, as made available by SimpleScalar [1]. We model a superscalar processor that is patterned roughly after the original, pre-SMT-enabled Intel Pentium 4 [11, 4] with DIVA checkers modelled, as supported by sim-mase. Since the register renaming scheme does not affect our experiments, the processor uses implicit renaming via the reservation stations (i.e., without an explicit register map table). Table 2 shows the detailed configuration of the processor we model. We modified SimpleScalar to allow for hard fault injection.

For benchmarks, we use the complete SPEC2000 benchmark suite with the reference input set. To reduce simulation time, we used SimPoint analysis [24] to sample from execution of each benchmark. Since the results in the rest of this section are presented in terms of normalized performances, we provide baseline error-free IPC results in Figure 3.

7.2 Detection and Diagnosis of Hard Faults

Our first set of experiments explores how accurately and quickly our scheme detects and diagnoses hard faults. In each experiment, we injected periodic transient faults in various structures and one hard fault in a single structure. We injected transient faults in a Poisson

Table 2. Parameters of Target System

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

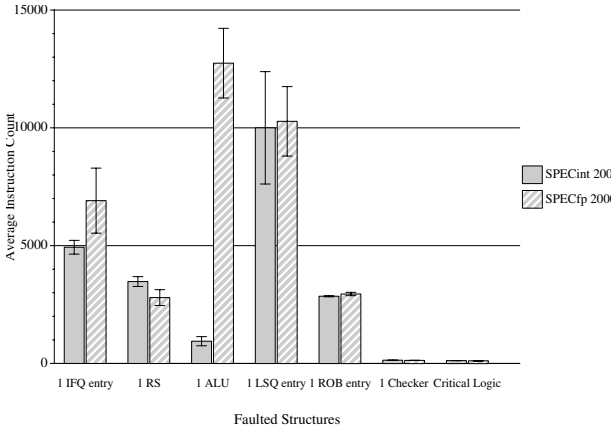

Figure 4. Hard fault diagnosis latency

distribution with a mean of one transient per billion instructions. All injected hard faults manifest as a single bit stuck-at-1. For the ROB, we inject the fault into the least-significant bit (LSB) of the data result. This causes the common value of 1 to provide data masking for the injected fault. For the RS and IFQ, we corrupt the LSB of the register identifier for the second argument of the instruction. This causes single-argument instructions to functionally mask this error and gives an even probability that two-argument instructions will experience data-

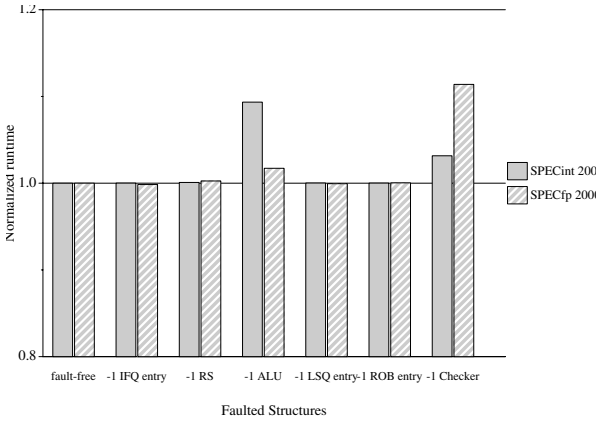

Figure 5. Performance impact of losing one component to a hard fault

masking for the injected fault. For the LSQ, we inject the fault in bit 16 of the address. This prevents data misalignment exceptions and provides an average-case data masking scenario. Finally, for the integer ALUs, we model faults as manifesting in the adder. We used a gate-level design for a 32-bit adder and selected a representative gate whose output is stuck-at-1 when the fault is injected. We performed a thorough gate-level fault simulation of the adder, and the gate we selected for fault injection represents the nominal masking case with a shading toward more masking, as this is a pessimistic assumption in our experiments. Masking was then evaluated for every instruction that accessed the ALU with the faulty adder.

In all of our experiments, the microprocessor detected and diagnosed the injected hard fault and did not mis-diagnose a soft fault as being hard. We measured how many instructions were executed before an injected hard fault was diagnosed, and we plot the results of this experiment in Figure 4. Since the results were relatively insensitive to the benchmarks, we present the mean results for SpecINT and SpecFP; the error bars in the figure represent one standard deviation above and below the mean. The results show that most hard faults are diagnosed within a few thousand instructions and that all of them are diagnosed within 15 thousand instructions. From this data, we also observe that the window of vulnerability for a faulty DIVA checker is about 200 instructions, which is easily within the recovery capabilities of typical hardware and software backward error recovery (BER) mechanisms. The different diagnosis latencies for different FDUs are a function of the relative usages of these structures as well as their error counter thresholds. Nevertheless, for all structures other than the DIVA checkers, the diagnosis latency is relatively unimportant, since between when the fault occurs and when it is diagnosed and the FDU deconfig-

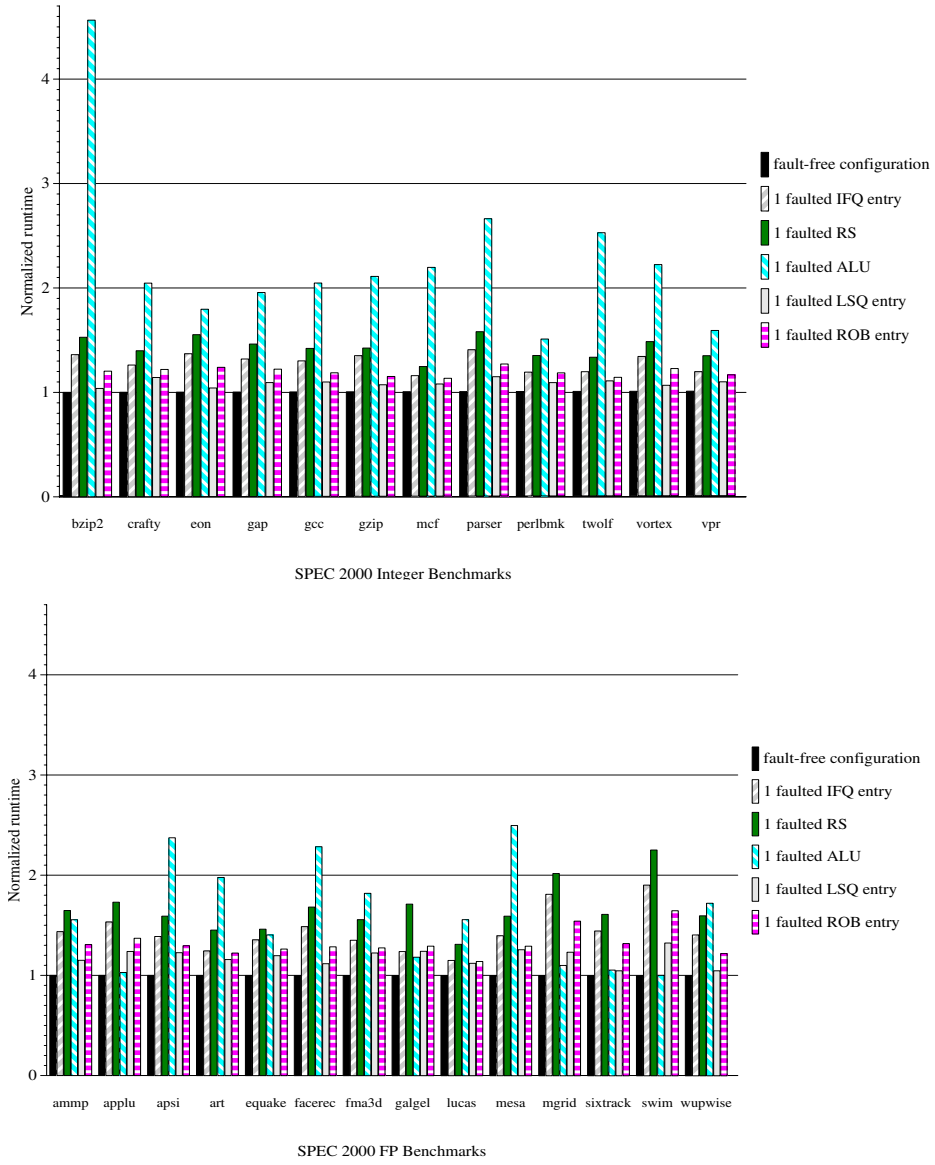

Figure 6. Performance comparison to DIVA-only (SpecINT and SpecFP)

ured, the checkers mask its effect with only a performance penalty caused by the number of pipeline flushes equal to the error counter threshold for the faulty FDU. Over the course of even thousands of instructions, this performance penalty is still unimportant. The key is not incurring that performance penalty over the entire lifetime of the processor, as results in Section 7.4 show.

7.3 Performance After Deconfiguring FDU

The second set of experiments evaluates the performance impact of de-configuring an FDU after having diagnosed it as being permanently faulty. In each of these experiments, we remove one of each type of FDU that we study. Figure 5 plots the runtime for each of these experiments, normalized to the error-free (fully-configured) case. Since there is little variation in the results across benchmarks, we plot the average results (geometric means of normalized runtimes) across the SpecINT and SpecFP benchmarks. The data show that the performance impact of deconfiguring an FDU is often small. This result, which corroborates prior work [25, 29], is in part due to the fact that the processor configuration we are modeling is over-provisioned for single SPEC benchmarks; the Pentium 4 is designed to run multiple threads simultaneously. Thus, resources are often idle in a typical single-threaded workload. There is, however, a non-negligible performance degradation due to deconfiguring an ALU or DIVA checker. Nevertheless, all of these faulty systems continue to function correctly and with reasonable performance.

7.4 Performance with Just DIVA

In this last set of experiments, we evaluate the performance of a microprocessor that relies strictly on the DIVA checkers to tolerate hard faults. While DIVA was designed primarily for soft faults and thus this is not a basis for a perfectly fair comparison, DIVA can tolerate hard faults and it is instructive to compare against this option. A DIVA-only system is also similar to a system that uses redundant threads for error detection and flushes the pipeline to recover from errors (assuming forward progress can be ensured). Figure 6 shows the effects of allowing sub-structures with hard faults to remain in use with the DIVA checkers correcting the errors that they activate. Once again, we plot runtimes that are normalized to the error-free case, but we do not aggregate results across benchmarks because there is significant variability across benchmarks. We do not inject hard faults into the DIVA checkers because they cannot tolerate them without our diagnosis/reconfiguration. Because the structures into which we are injecting faults are used frequently and are critical to the correctness of the processor, the results show that hard faults have a drastic impact on system performance when DIVA is forced to correct the errors they create. The performance of the DIVA-only system is far worse than the performance we demonstrated for our system in Section 7.3. Technology trends toward deeper pipeline implementations will only serve to make the performance penalty for each error's recovery (i.e., pipeline flush) more severe. The relative difference in magnitude of the structure-to-structure penalty is directly related to how frequently a given sub-structure is used by the workload. Benchmark-to-benchmark variation for a given type of FDU is a result of the distribution and frequency of pre-existing stall events in a given benchmark. The causes of these events, such as cache misses or branch mispredictions, result in a percentage of corrected errors falling in the shadow of another pipeline-clearing event, thus diminishing the penalty associated with the error correction. For example, a benchmark with many branch mispredictions is less sensitive to pipeline flushes due to errors, if the errors tend to occur soon after branch mispredictions, since there is less state that gets flushed by the error.

7.5 Summary and Discussion of Results

The experimental results in this section confirm that existing microprocessors have redundancy that can be exploited to tolerate hard faults. We have also shown that we can accurately and quickly diagnose hard faults and reconfigure around faulty FDUs to provide a microprocessor that performs only slightly worse than a fault-free microprocessor. Moreover, it vastly outperforms the alternative of just relying on DIVA.

Technological and architectural trends drive this work and encourage further work in this area. The incidences of hard faults and fabrication defects will continue to increase. Also, as microarchitects try to exploit ever more ILP and thread level parallelism, there will be even more redundancy that can be leveraged for improving reliability and yield. In particular, emerging SMT processors will have more redundant hardware and fewer singleton resources. Thus the advantages of our approach will increase due to these trends. The caveat is that, as workloads evolve to take advantage of this extra hardware, the performance impact of having to deconfigure an FDU will increase. Nevertheless, even a heavily loaded microprocessor will continue to function correctly and with better performance than just DIVA in the presence of operational hard faults and fabrication defects.

8 Related Work

In this section, we present prior research in tolerating hard faults and fabrication defects. A canonical design for tolerating hard faults is the IBM mainframe [26]. Mainframes not only have redundant processors, but they also incorporate redundancy within the processor in order to seamlessly tolerate hard faults. The IBM G5 microprocessor, for example, has redundant units for fetch/decode and for instruction execution. Some other traditional fault-tolerant computers, such as the Stratus [35] and the Tandem S2 [15], simply replicate entire processors. An even more extreme case of using redundancy to tolerate fabrication defects and, to a lesser extent, operational hard faults, is the Teramac [8]. The Teramac is designed to make use of components that are likely to be faulty, and it is motivated by expected defect rates in nanotechnology. While these systems all provide excellent resilience to hard faults, such heavyweight redundancy incurs significant costs in terms of hardware and power consumption.

DIVA [2] and redundant thread schemes provide low cost and low power alternatives to heavyweight redundancy. Among the redundant thread schemes, the ones that perform recovery as well as error detection include AR-SMT [21], Slipstream [30], SRT [20, 17], and SRTR [33]. All of these schemes were designed for transient faults and thus share the same drawback as DIVA, with respect to hard faults, since they incur a pipeline squash (and its corresponding performance and energy penalty) every time a fault manifests itself. For hard faults in frequently-used microprocessor structures, fault manifestation is too frequent and the performance of these schemes suffers.

There are lightweight approaches by Shivakumar et al. [25] and Srinivasan et al. [29] that, similar to our work, leverage existing redundancy in microprocessors. Shivakumar et al.'s work differs in that it is strictly for tolerating fabrication defects and does not extend to hard faults that occur during execution. They combine offline (pre-shipment) testing and diagnosis of microprocessors with deconfiguration capabilities to improve effective yield. Our approach combines deconfiguration with online error detection and fault diagnosis to improve both yield and reliability. Srinivasan et al.'s work does not address error detection or fault diagnosis.

A recent approach to improving microprocessor reliability in the presence of operational hard faults (but not fabrication defects) is to use dynamic reliability management [27]. In this approach, the processor dynamically adapts, based on a model of its estimated lifetime, in order to achieve a desired lifetime. In particular, if the processor is running too hot, due to a particular workload, it may use dynamic voltage scaling to cool down and improve its reliability. This approach is orthogonal and complementary to ours.

A recent scheme for tolerating only fabrication defects, called Rescue [23], utilizes circuit transformations to improve testability and enable coarse-grain diagnosis of defective components (ways of a superscalar processor). The finer grain diagnosis in our research enables us to discard less fault-free hardware, and it may enable us to tolerate more hard faults before failure.

There are other non-comprehensive approaches to tolerating hard faults in specific parts of a computer system. One option for storage structures is to protect them with error correcting codes (ECC), as in IBM mainframes [26]. Combining ECC for arrays with DIVA avoids costly DIVA recoveries. However, ECC protection of arrays is on the critical path for array access (both read and write), and it will thus add to the microprocessor's critical path and degrade its performance in the fault-free case. Storage structures can also be protected by using a level of indirection to map out faulty portions of the structure. Whole disk failures were addressed by RAID [19]. For disk faults that did not incapacitate the entire disk, the solution was to map out faulty portions at the sector granularity. Similar approaches have been developed for DRAM main memory. Whole chip failures are tolerated by chipkill memory and RAID-M [9, 12], and partial failures are tolerated with schemes that map out faulty locations [7, 16, 22]. For SRAM caches, techniques have been developed to map out defective locations during fabrication [36] and, more recently, during execution [18]. SRAS [5] uses a similar technique to map out defective rows in microprocessor array structures, such as the reorder buffer and branch history table.

9 Conclusions

To address the emerging problem of operational hard faults and fabrication defects in microprocessors, we have developed a microprocessor design that leverages the existing redundancy in current microprocessors. This redundancy, which exists to improve performance by exploiting ILP and thread level parallelism, can be used to mask hard faults. Our microprocessor design integrates DIVA-style error detection with a new mechanism for diagnosing hard faults. After diagnosis, it deconfigures the faulty FDU and continues operation. Experimental results demonstrate that our scheme can accurately and quickly diagnose hard faults and reconfigure around faulty FDUs to provide a microprocessor that performs only somewhat worse than a fault-free system.

Acknowledgments

This material is based upon work supported by the National Science Foundation under grants CCR-0309164 and CCF-0444516, the National Aeronautics and Space Administration under Grant NNG04GQ06G, a Duke Warren Faculty Scholarship (Sorin), and donations from Intel Corporation. We thank Derek Hower for modeling faults in the adder. We thank Alvy Lebeck and rest of the Duke Architecture Reading Group for helpful feedback on this paper.

References

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

[2] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proc. of the 32nd Annual IEEE/ACM Int'l Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[3] D. T. Blaauw, C. Oh, V. Zolotov, and A. Dasgupta. Static Electromigration Analysis for On-Chip Signal Interconnects. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems*, 22(1):39–48, Jan. 2003.

[4] D. Boggs et al. The Microarchitecture of the Intel Pentium 4 Processor on 90nm Technology. *Intel Technology Journal*, 8(1), Feb. 2004.

[5] F. Bower, P. Shealy, S. Ozev, and D. Sorin. Tolerating Hard Faults in Microprocessor Array Structures. In *Proc. of the Int'l Conference on Dependable Systems and Networks*, pages 51–60, June 2004.

[6] J. Carter, S. Ozev, and D. Sorin. Circuit-Level Modeling for Concurrent Testing of Operational Defects due to Gate Oxide Breakdown. In *Proc. of Design, Automation, and Test in Europe (DATE)*, pages 300–305, Mar. 2005.

[7] T. Chen and G. Sunada. An Ultra-Large Capacity Single-Chip Memory Architecture with Self-Testing and Self-Repairing. In *Proc. of the Int'l Conference on Computer Design (ICCD)*, pages 576–581, Oct. 1992.

[8] W. B. Culbertson et al. The Teramac Custom Computer: Extending the Limits with Defect Tolerance. In *Proc. of the IEEE Int'l Symposium on Defect and Fault Tolerance in VLSI Systems*, Nov. 1996.

[9] T. J. Dell. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory. IBM Microelectronics Division Whitepaper, Nov. 1997.

[10] D. J. Dumin. *Oxide Reliability: A Summary of Silicon Oxide Wearout, Breakdown and Reliability*. World Scientific Publications, 2002.

[11] G. Hinton et al. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Feb. 2001.

[12] IBM. Enhancing IBM Netfinity Server Reliability: IBM Chipkill Memory. IBM Whitepaper, Feb. 1999.

[13] International Technology Roadmap for Semiconductors, 2003.

[14] JEDEC Solid State Technology Association. Failure Mechanisms and Models for Semiconductor Devices. JEDEC Publication JEP122-B, Aug. 2003.

[15] D. Jewett. Integrity S2: A Fault-Tolerant UNIX Platform. In *Proc. of the 21st Int'l Symposium on Fault-Tolerant Computing Systems*, pages 512–519, June 1991.

[16] P. Mazumder and J. S. Yih. A Novel Built-In Self-Repair Approach to VLSI Memory Yield Enhancement. In *Proc. of the Int'l Test Conference*, pages 833–841, 1990.

[17] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed Design and Implementation of Redundant Multhreading Alternatives. In *Proc. of the 29th Annual Int'l Symposium on Computer Architecture*, pages 99–110, May 2002.

[18] M. Nicolaidis, N. Achouri, and S. Boutobza. Dynamic Data-bit Memory Built-In Self-Repair. In *Proc. of the Int'l Conference on Computer Aided Design*, pages 588–594, Nov. 2003.

[19] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. of 1988 ACM SIGMOD Conference*, pages 109–116, June 1988.

[20] S. K. Reinhardt and S. S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proc. of the 27th Annual Int'l Symposium on Computer Architecture*, pages 25–36, June 2000.

[21] E. Rotenberg. AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors. In *Proc. of the 29th Int'l Symposium on Fault-Tolerant Computing Systems*, pages 84–91, June 1999.

[22] K. Sawada et al. Built-in Self Repair Circuit for High Density ASMIC. In *Proc. of the IEEE Custom Integrated Circuits Conference*, 1989.

[23] E. Schuchman and T. N. Vijaykumar. Rescue: A Microarchitecture for Testability and Defect Tolerance. In *Proc. of the 32nd Annual Int'l Symposium on Computer Architecture*, pages 160–171, June 2005.

[24] T. Sherwood et al. Automatically Characterizing Large Scale Program Behavior. In *Proc. of the Tenth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[25] P. Shivakumar, S. W. Keckler, C. R. Moore, and D. Burger. Exploiting Microarchitectural Redundancy For Defect Tolerance. In *Proc. of the 21st Int'l Conference on Computer Design*, Oct. 2003.

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

[27] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The Case for Lifetime Reliability-Aware Microprocessors. In *Proc. of the 31st Annual Int'l Symposium on Computer Architecture*, June 2004.

[28] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *Proc. of the Int'l Conference on Dependable Systems and Networks*, June 2004.

[29] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. Exploiting Structural Duplication for Lifetime Reliability Enhancement. In *Proc. of the 32nd Annual Int'l Symposium on Computer Architecture*, June 2005.

[30] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream Processors: Improving both Performance and Fault Tolerance. In *Proc. of the Ninth Int'l Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–268, Nov. 2000.

[31] J. Tao, J. F. Chen, N. W. Cheung, and C. Hu. Modeling and Characterization of Electromigration Failures Under Bidirectional Current Stress. *IEEE Trans. on Electron Devices*, 43(5):800–808, May 1996.

[32] D. M. Tullsen et al. Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor. In *Proc. of the 23rd Annual Int'l Symposium on Computer Architecture*, pages 191–202, May 1996.

[33] T. N. Vijaykumar, I. Pomeranz, and K. K. Chung. Transient Fault Recovery Using Simultaneous Multithreading. In *Proc. of the 29th Annual Int'l Symposium on Computer Architecture*, pages 87–98, May 2002.

[34] C. Weaver and T. Austin. A Fault Tolerant Approach to Microprocessor Design. In *Proc. of the Int'l Conference on Dependable Systems and Networks*, pages 411–420, July 2001.

[35] D. Wilson. The Stratus Computer System. In *Resilient Computer Systems*, pages 208–231, 1985.

[36] L. Youngs and S. Paramanandam. Mapping and Repairing Embedded-Memory Defects. *IEEE Design & Test of Computers*, pages 18–24, January-March 1997.