Online Diagnosis of Hard Faults in Microprocessors

FRED A. BOWER
Duke University and IBM Systems and Technology Group
and
DANIEL J. SORIN and SULE OZEV
Duke University

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 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.

Categories and Subject Descriptors: B.2 [**Arithmetic and Logic Structures**]: Reliability, Testing, and Fault-Tolerance—*Diagnostics, error checking, redundant design*; B.3 [**Memory Structures**]: Reliability, Testing, and Fault-Tolerance—*Diagnostics, error checking, redundant design*; B.7 [**Integrated Circuits**]: Reliability and Testing—*Error checking, redundant design*; B.8 [**Performance and Reliability**]: Reliability, Testing, and Fault-Tolerance; C.1 [**Processor Architectures**]: General

General Terms: Design, Performance, Reliability

Extension of Conference Paper: Fred A. Bower, Daniel J. Sorin, and Sule Ozev. "A Mechanism for Online Diagnosis of Hard Faults in Microprocessors," In *38th Annual International Symposium on Microarchitecture (MICRO)*, November 2005.
This research was 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.
Authors' addresses: Fred A. Bower, Department of Computer Science, Duke University, PO Box 90129 Durham, NC 27708-0129; email: bowerf@us.ibm.com; Daniel J. Sorin and Sule Ozev, Department of Electrical and Computer Engineering, Duke University, PO Box 90291, Durham, NC 27708; email: {sorin, sule}@ee.duke.edu.

Additional Key Words and Phrases: Hard fault tolerance, processor microarchitecture, fine-grained diagnosis

ACM Reference Format:

Bower, F. A., Sorin, D. J., and Ozev, S. 2007. Online diagnosis of hard faults in microprocessors. Architec. Code Optim. 4, 2, Article 8 (June 2007), 32 pages. DOI = 10.1145/1250727.1250728 http://doi.acm.org/10.1145/1250727.1250728

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 [Srinivasan et al. 2004b], 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 [International Technology Roadmap for Semiconductors 2003]. 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 macroscale 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 [Youngs and Paramandam 1997; Nicolaidis et al. 2003] or buffers [Bower et al. 2004], 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 [Shivakumar et al. 2003; Srinivasan et al. 2004a, 2005, is that modern superscalar microprocessors, particularly simultaneously multithreaded (SMT) microprocessors [Tullsen et al. 1996], 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.

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 preshipment testing instead of online error detection and diagnosis [Shivakumar et al. 2003]. Other work has explored deconfiguration and has left detection and diagnosis as open problems [Srinivasan et al. 2005].

In this paper, we discuss integrated design options for microprocessors that achieve all three of these goals; we also present one particular microprocessor in this design space. First, our microprocessor detects and corrects errors, because of both transient and hard faults, using previously developed DIVA-style [Austin 1999] 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), 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 ones 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 [Shivakumar et al. 2003]. 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 preexisting and new deconfiguration schemes.
- An experimental evaluation that demonstrates that microprocessors 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 toward 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 [Srinivasan et al. 2004b; Jedec Solid State Technology Association 2003], that is, hard faults that occur over the lifetime of the microprocessor. Srinivasan et al. [2004b] determine that electromigration [Tao et al. 1996; Blaauw et al. 2003] and gate oxide breakdown [Dumin 2002] 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 resulting from the creation of a highly conductive path between its gate and its bulk. A newly manufactured oxide contains inherent electron traps because of imperfections in the fabrication process. Over the lifetime of the device, the number of such traps increases as a result of 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 [Carter et al. 2005].

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 macroscale redundancy (e.g., TMR). We do not claim to innovate in this area; we simply seek to use a preexisting 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 [Austin 1999]. In a system with DIVA dynamic verification, a total of n checkers are added at the commit stage of the typical n-way superscalar processor pipeline. These checkers are small, simple, in-order cores. According to Weaver and Austin [2001], 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 reexecute each instruction and compare their results with those of the superscalar core. The original DIVA paper [Austin 1999] 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 attractive options besides DIVA exist for error detection and correction, such as redundant multithreading. With redundant threading, each thread is replicated and executed in parallel with the original. The results of each are compared periodically—at every instruction or more infrequently—and, if they do not match, an error has been detected. The processor can then recover to the most recent instruction whose results were identical for the two threads by flushing pipeline state and reexecuting the instruction that encountered the miscomparison. Because the redundant instructions either execute on different resources (e.g., different adders) or on the same resource, but at different times, this scheme is well-suited to detecting transient errors. It can also detect some errors because of hard faults, but it will not detect a hard fault when both redundant instructions use the same faulty resource at different times. Several redundant threading proposals have appeared in the literature (AR-SMT [Rotenberg 1999], Slipstream [Sundaramoorthy et al. 2000], SRT [Mukherjee et al. 2002; Reinhardt and Mukherjee 2000], and SRTR [Vijaykumar et al. 2002]). We believe that all of these schemes can either provide error detection and correction in our diagnosis framework or can easily be adapted to do so.

We chose DIVA over the alternatives, including redundant threading, 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 [Austin 1999].

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 substructures 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. Because deconfiguration is more easily achieved with this FDU selection, we favor it over an FDU selection that seeks to have equal amounts of logic in each FDU. For the processors that 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 reorder buffer (ROB), individual arithmetic logic units (ALU), and the individual DIVA checkers. While our chosen processor designs have only one of some of the more complex ALUs (for example, the integer multiplier), we include them in our diagnosis evaluation to show that the diagnosis is capable of identifying hard faults in these units. 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 prespecified 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 tradeoff in our design that will allow implementations to select the appropriate FDU granularity/overhead tradeoff. For typical superscalar microprocessor designs, including those that we evaluate in Section 7, roughly 20 bits are required to track this fine-grained FDU utilization information. 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 bits are a small addition, especially since not all of the bits need to traverse the whole pipeline.

For each FDU we track, the processor maintains a small, saturating error counter. The purpose of the error counter is to differentiate hard from soft faults. At the scope of the error detection and correction mechanisms considered (that is, at the instruction granularity), hard faults are not distinguishable from soft faults at the time an error is detected and corrected. For hard faults affecting frequently used structures, we observe an error detection and correction rate that is orders of magnitude higher than that observed for transient faults. Occasional corrections because of soft faults do not trigger diagnosis because they do not saturate the error counter for any given FDU in the system. Periodic clearing of the error counters prevents soft-fault corrections from accumulating to a point where diagnosis is triggered.

4.1.1 *Design Issues.* Using saturating error counters for diagnosis of hard faults presents four challenges. First, after the FDUs have been selected and configured for diagnosis in an implementation of our mechanism, all remaining logic for which the error detection and correction mechanism detects and corrects 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 and any common datapaths that all instructions must traverse.

The second issue with using saturating error counters is that 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 recommend clearing once every 10 s, even though current terrestrial transient fault rates do not approach this frequency. This rate is based upon our experimental results for latency to diagnose hard faults. Our experimental results show that the latency to diagnose a hard fault in the FDUs we evaluate is less than 1/10th of a second at multigigahertz frequencies, even in infrequently used FDUs. By clearing at an interval well above the diagnosis latency of FDUs we care to diagnose, we ensure that we will diagnose hard faults that greatly affect system performance if they are allowed to continue to cause error detection and correction to occur. If diagnosis spans a clearing interval, we are merely temporarily postponing the deconfiguration. 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. To illustrate this, consider the case where we have a single adder and two ROB entries. Assuming we use the adder and one of the ROB entries each cycle, we can observe that a fault in the ALU will cause both ROB entries' error counters to accumulate errors at a rate of one-half that of the adder. To avoid misdiagnosis, we would need the adder's saturation value to be greater than that of an ROB entry, but not more than twice the ROB entry value. Thus, for frequently utilized FDUs, a larger counter value is required to prevent the misdiagnosis of a fault in an up- or downstream structure.

The final challenge is that the chosen FDUs must be used reasonably independently. Otherwise, for example, if every time an instruction uses FDU A it also uses FDU 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 nonuniformly (e.g., higher priority for ALU0) 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 ALU0 could be significantly greater than that of other ALUs and thus preclude hard faults in them from being diagnosed (since the thresholds assume uniform utilization). This scheduling modification is not necessary for resources that are naturally scheduled uniformly, like ROB entries. We found though, that round-robin scheduling alone does not avoid all lockstep allocation of resources. 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 ALU0 always used Checker1, ALU1 was perfectly correlated with Checker2, and ALU2 was perfectly correlated with Checker0. To avoid this lockstep allocation, we introduced a small amount of pseudorandomness into the scheduling of checkers. Every cycle, the first checker to be considered for allocation is determined based on pseudorandom data (e.g., low-order bits of the tick counter) and then subsequent checkers are allocated sequentially (mod width) after the first one. This pseudorandomness, combined with round-robin scheduling, prevents lockstep allocation and achieves reasonably uniform utilization of each set of identical FDUs.

4.1.2 *Heuristics for Choosing Error Counter Values.* Given these four challenges, we developed a heuristic for choosing appropriate threshold values for the saturating error counters. As it is always possible to craft an instruction sequence that leads to saturation of the wrong counter, the best that we can do is to choose saturating values and then verify correct diagnosis operation via simulation. Using this heuristic for the designs we evaluate in Section 7, we will see that it does provide effective threshold values that lead to low-latency

Table I. Error Counter Thresholds

FDU	Threshold	Storage Requirements for Diagnosis
Instruction fetch queue entry	32	5 bits/entry
Reservation station	32	5 bits/entry
Reorder buffer entry	16	4 bits/entry
Load/store queue entry	16	4 bits/entry
Integer ALU	64	6 bits/unit
Floating-point ALU	64	6 bits/unit
Integer multiplier	32	5 bits/unit
Floating-point multiplier	32	5 bits/unit
DIVA checker	64	6 bits/checker
Critical logic (issue, etc.)	128	7 bits

diagnoses of a wide range of FDUs. The heuristic is as follows:

1. Select a minimum power-of-two threshold value well above what transient or intermittent faults would cause in a counter-clearing interval.
2. Segregate FDU types by the population of units for each type. For FDUs that have a population that is not a power of two, round the population to either the next larger power of two, if it is a heavily-utilized resource, or the next smaller if it is a less-heavily utilized resource. At this point, resource utilization information may need to be gathered via simulation of representative workloads. Group like-population FDUs together. Assuming that there is some logic for which error detection and correction can contain a fault, but for which there is no associated FDU, create a singleton group for "critical logic."
3. Assign the minimum threshold chosen in step 1 to the highest-populated FDU group.
4. Assign the next power-of-two as the error counter threshold for the next-most-populated FDU group.
5. Repeat step 4 for all remaining FDU groups, assigning the highest threshold to the "critical logic" group.
6. Simulate the processor with a representative set of workloads and FDU faults to verify that the thresholds chosen cause the diagnosis mechanism to converge on the faulty FDU.
7. Using the simulation results from step 6, reduce the threshold by a factor of two (one bit) for those items whose diagnosis latency is large. If this threshold reduction results in no FDUs with an error counter threshold in the middle of the threshold range, reduce all higher error counter thresholds by a factor of two. This will result in a set of error counters whose bit width is monotonically increasing, without any gaps from lowest to highest. Repeat the simulation to verify correct operation.

In Table I, we list the counter thresholds for the FDUs we consider in this paper, including the per-unit storage cost for each FDU's counter. These values were derived for our three evaluated processor design points using the above heuristic with a minimum threshold value of 16. For resources that are less utilized, such as the floating-point units, our mechanism may take additional

time to diagnose, even with the lower threshold than their more heavily utilized integer counterparts. Any hard fault that gets exercised so rarely as to not exceed our error-counter threshold between periodic counter zeroing is also so rare that it incurs little performance penalty for its infrequent error recoveries. In this situation, simply using DIVA to correct errors because of a hard fault in a lightly utilized FDU is sufficient. The key observation is that our scheme can diagnose hard faults in the highly utilized resources, so that the microprocessor avoids frequent recoveries.

4.1.3 *Discussion.* 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.

There are certain scenarios in which the diagnosis mechanism can temporarily 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. The use of saturating counters for the FDUs within the processor also introduces the possibility that the wrong unit's counter will saturate first for a particular instruction sequence. To address this issue, we use an iterative diagnosis process. Diagnosis is not considered complete until fault rates fall below a hard-wired threshold set by the designer. We set this threshold sufficiently high to allow for all hard faults that we wish to diagnose to be accounted for. The final unit deconfigured before this error rate change is considered faulty, while all other units deconfigured in the affiliated diagnosis cycles are returned to operation. In general, if deconfiguration does not help (i.e., as unit(s) are deconfigured, error counters continue to saturate in close temporal proximity), then the system can reconfigure the previously mapped out unit(s) back into the system (under the common assumption of one hard fault at a time) once the correct unit has been identified and deconfigured. Our evaluation in Section 7 will show that one diagnosis iteration is sufficient a vast majority of the 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.

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 results from 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 pseudorandomly—would be extremely large. BIST requires the processor to be offline for testing to occur. Our online error counting differs from BIST by diagnosing faults via the observation of the execution of actual software with the software's instructions acting as test vectors and the error detection and correction acting as output verifier. This ensures that we always have a test vector that exposes a detected fault. Finally, BIST adds performance overhead because of the extra multiplexers that choose between normal inputs and BIST inputs. Unlike our diagnosis overhead, this overhead is on the critical path of instruction flow through the processor. Since many processors have some form of BIST support already in their design, use of our mechanism presents an opportunity to remove this hardware from the critical path, replacing BIST with our mechanism.

Within our diagnosis mechanism, there are also design options. If, instead of using DIVA, we used redundant threading for error detection and correction, this would also affect our diagnosis mechanism. DIVA assumes that the checker core is always fault-free and thus it can diagnose with only two copies of a given unit (e.g., the multiplier in the out-of-order core and the multiplier in the checker). If a redundant threading scheme is used for detection and correction of hard faults, it must use independent resources for each of the primary and redundant threads in order to guarantee that results are not derived from the same faulty hardware. Since with redundant threading, there is no known-good unit, we need at least three copies of a given unit to ensure forward progress is achieved in the presence of a hard fault. Otherwise, for example, a hard fault in one of two multipliers would cause repeated mismatched results with no way to determine which result is correct. In this case, the instruction would replay continually until a higher-level deadlock detection mechanism activated. With at least three copies of a unit, the two fault-free copies will calculate the correct

result, allowing us to isolate the faulty functional unit and then increment its associated error counter.

Finally, an alternative, related diagnosis mechanism bears mentioning. As an alternative to keeping saturating error counters for each FDU and all logic covered by the chosen error detection and correction mechanism, a microarchitect could opt to have a single, saturating error counter that triggers diagnosis. This counter, when saturated, would lead the system to replay the faulted instruction, deconfiguring and replacing each FDU involved in the last erroneous result until a correct result is obtained. At that point, the currently deconfigured FDU would be deemed faulty and would remain deconfigured from the system, with normal operation resuming. This method presents three drawbacks. First, to use this alternative, the microarchitecture would have to support directed steering of instructions through specific FDUs to allow for multiple replays with only a single suspect removed from the processing of each replay. This would add additional complexity to every stage of the pipeline. Second, if a transient fault happens to cause the diagnosis in this alternative scheme, diagnosis will take the maximum amount of time and will result in no unit deconfigured, requiring a subsequent diagnosis attempt on the next encountered error. Finally, if a transient occurs during diagnostic replay, it will result in either the diagnosis missing the suspect unit, requiring another round of diagnosis, or a double-fault case, which greatly complicates error detection. Given these issues, we chose the use of error counters for each FDU, which leads to a single deconfiguration action upon saturation without requiring any directed replay of instructions.

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 preexisting 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 deconfiguration of a single entry with little additional latency added to access time for the structure [Bower et al. 2004; Shivakumar et al. 2003]. In the method by Bower et al. [2004], 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 in-use, thus removing it from further operation [Shivakumar et al. 2003]. Once again, Bower et al. [2004] assume that cold spares may be available,

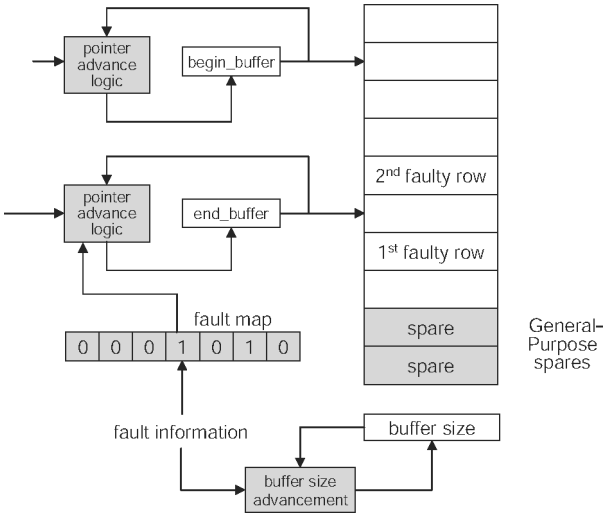

Fig. 1. Deconfiguration of entries in a circular buffer (e.g., reorder buffer). Shading indicates hardware added for entry deconfiguration purposes.

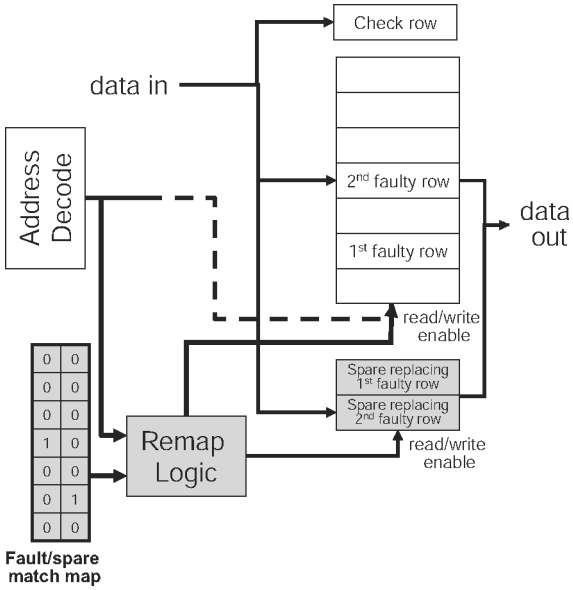

Fig. 2. Deconfiguration of entries in a tabular structure (e.g., reservation station). Shading indicates hardware added for entry deconfiguration purposes.

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 [Shivakumar et al. 2003]. Cold sparing of functional units is possible, but it may require too much hardware area, 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 [Weaver and Austin 2001]. In addition to DIVA, which provides benefits even without our additions, we also add: error counters, wires for tracking each instruction's resource usage, 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 that are used with a majority of instructions, because of ambiguity reasons stated at the end of Section 4.1. Examples of these resources include issue logic and common datapath lines. These singletons are always in lock-step scheduling with each other. Future work will involve designing modular implementations of these currently monolithic structures, so that incremental redundancy is feasible.

Related to this issue is the impact of hard faults in the datapaths and unique logic for each FDU. For some FDUs selected, there is a unique set of logic and data paths that will affect correct execution for a subset of instruction paths through the processor if hard faults are present, but for which diagnosis will lead to deconfiguration of a downstream unit. In these instances, the deconfiguration action results in discontinued use of the faulted portion of the circuit via deconfiguration of the downstream FDU, so the correct thing happens with our diagnosis mechanism despite the problem actually residing in a different FDU.

For example, consider bypass paths between ALUs. A fault in a bypass path will be flagged as a fault in the destination ALU by our mechanism, even though that ALU is able to correctly process instructions when the bypass path is not active. By discontinuing use of the ALU, however, we observe that the bypass path is no longer used, thus eliminating the fault from further activation. To prevent this effect, we could treat bypass paths as separate FDUs, but their deconfiguration would not be straightforward, so we choose to lump them with the ALU FDUs for simplicity of the overall design. The tradeoff here is that a fully functional ALU is deconfigured to prevent the effects of a hard fault in a bypass path.

Second, 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 miscomparison. Without a higher-level recovery scheme, such as checkpointing, this erroneously committed state represents an unrecoverable error. It should be noted that DIVA also can cause silent data corruption when a transient fault affects a checker. Since this is not detectable by DIVA or our diagnosis mechanism, it remains an exposure of any DIVA-based system.

Finally, because we elected to use DIVA in our designs, we are unable to detect and correct problems in uncached loads and stores. This is a limitation of DIVA that we inherit. This adds complexity to recovery, particularly in the case where the checker is at fault. Discussion of techniques to work around this limitation is beyond the scope of this paper. This problem is not new to checkpointing research. If a designer requires containment of this escape in the scheme, an appropriate checkpointing scheme will be required. The use of an alternative error detection and correction mechanism, capable of detecting and correcting these errors, would also correct this issue.

7. EVALUATION

Our evaluation consists of experiments to explore the effectiveness of our diagnosis scheme in a representative sample of processor designs. Our evaluation has the following goals:

- First, we want to show that commodity design points using our reliable architectural extensions 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.
- Fourth, we want to perform a sensitivity analysis for singleton complex, combinational logic units, such as the integer and floating-point multipliers, in order to determine if protection of these units warrants further investigation.
- Finally, taking all three of our chosen design points together, we show the general applicability of the technique to a broad set of designs from the commodity microprocessor design space.

7.1 Methodology

To evaluate our design for proper operation under the fault models considered, we modified sim-mase, as made available by SimpleScalar [Austin et al. 2002]. We model three separate microprocessor designs, each patterned after an existing commodity microprocessor design. The first design, Narrow, is a superscalar processor that is patterned roughly after the original, pre-SMT-enabled Intel Pentium 4 [Boggs et al. 2004; Hinton et al. 2001]. The second design, Deep–Narrow, is a more deeply pipelined implementation of Narrow, patterned on current Intel Pentium 4 designs [Boggs et al. 2004]. Deep–Narrow differs from Narrow in the depth of its pipeline, carrying an additional 11 stages to allow for faster clocking. The final processor configuration, Short-Wide, is inspired by the AMD Athlon/Opteron processor family [AMD 2005; Huynh 2003]. This design point favors a wider, shorter pipeline that, in practice, is clocked at a lower rate than competing designs from Intel. Since the register renaming scheme does not affect our experiments, all of the processor configurations use implicit renaming via the reservation stations (i.e., without an explicit register map table). Table II shows the details of all three configurations, including the overheads for our diagnosis scheme. We utilize the DIVA-style checker capability provided by sim-mase and, in addition, 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 [Sherwood et al. 2002] to sample from execution of each benchmark. The 100-million instruction SimPoints were used, with 100-million instructions of detailed simulation warm-up used prior to simulating the SimPoint for all benchmarks requiring fast forwarding. Since we present results in the rest of this section in terms of normalized performances, we provide baseline error-free IPC results for each of the three processor design points in Figure 3.

The goal of all of our evaluation is to show how the processor behaves in the presence of a hard fault. The likelihood of a hard fault affecting processor operation is highly dependent upon the process used to manufacture the part, the complexity of the design, and the operating environment that the part is deployed in. The discussion of these issues is an active body of research and is beyond the scope of this evaluation.

Table II. Parameters of Target Systems[a]

Feature	Narrow	Deep–Narrow	Short–Wide
Pipeline stages	20	31	12
Width: Fetch/issue/ commit/check	3/6/3/3	3/6/3/3	9/9/9/9
Branch predictor	2-level GShare, 4K Entries	2-level GShare, 4K Entries	2-level GShare, 4K Entries
Instruction fetch queue	64 Entries	64 Entries	72 Entries
Reservation stations	32	32	54
Reorder buffer	128 Entries	128 Entries	216 Entries
Load/store queue	48 Entries	48 Entries	44 Entries
Integer ALUs	2 Units, 1 cycle	2 Units, 1 cycle	3 Units, 5 cycle
Integer multiply/divide	1 Unit, 14 cycle multiply, 60-cycle divide	1 Unit, 14 cycle multiply 1, 60-cycle divide	1 Unit, 8 cycle multiply, 74-cycle divide
Floating point ALUs	2 Units, 1-cycle	2 Units, 1 cycle	3 Units, 5-cycles
Floating point multiply/divide/square root	1 Unit, 1-cycle multiply, 16-cycle divide/square root	1 Unit, 1-cycle multiply, 16-cycle divide/square root	1 Unit, 24-cycle multiply, 26-cycle divide, 35-cycle square root
L1 I-Cache	16 KB, 8-way, 64-byte blocks, 2-cycles	16 KB, 8-way, 64-byte blocks, 2-cycles	64 KB, 2-way, 64-byte blocks, 3-cycles
L1 D-Cache	16 KB, 8-way, 64-byte blocks, 2-cycles	16 KB, 8-way, 64-byte blocks, 2-cycles	64 KB, 2-way, 64-byte blocks, 3-cycles
L2 cache (unified)	1 MB, 8-way, 128-byte blocks, 7-cycles	1 MB, 8-way, 128-byte blocks, 7-cycles	1 MB, 16-way, 128-byte blocks, 20-cycles
Diagnosis: error counters	1249 Bits	1249 Bits	1219 Bits
Diagnosis: FDU tracking	19 Lines	19 Lines	22 Lines

[a]Shaded entries for Deep–Narrow are identical to those of Narrow.

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 one hard fault in a single structure. All injected hard faults manifest as a single-bit stuck-at-1. To accurately account for masking effects, we inject the hard fault at a specific site in the FDU, with the exception of complex FDUs for which we lack a detailed implementation. Our hard fault selection attempts to provide greater masking of fault effects, which leads to a smaller performance penalty and longer diagnosis latency because of fewer error detections and corrections. We do this because it is a pessimistic case for the operation of our mechanism.

Because transient faults are relatively rare for the intervals we are simulating, we expect no more than one transient to occur during a diagnosis interval. To model the effects of this scenario, we ran each simulation with the effect of a single, observed transient added at the beginning of the simulation (that is, one random set[1] of FDUs' counters started with an error count of one, rather than

[1]A set is defined as one of each required FDU type for a particular instruction's processing. Recall that DIVA cannot determine whether an error came from a transient or hard fault and also cannot

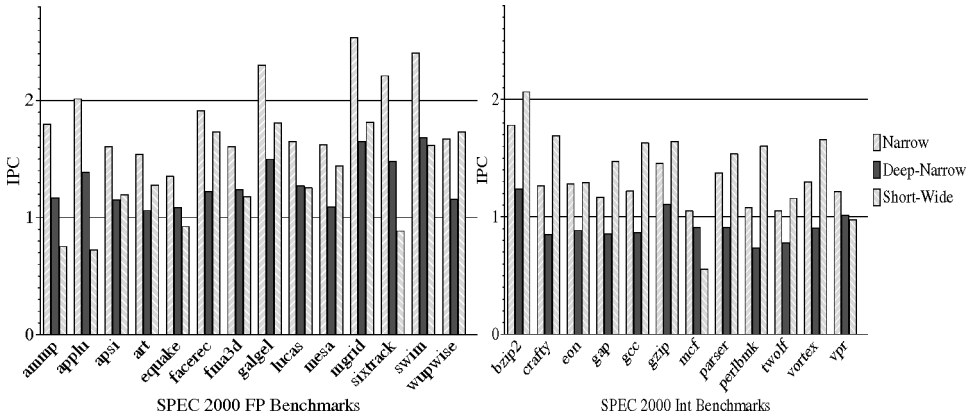

Fig. 3. Error-free performance (SPECfp and SPECint) for each of the three evaluated processor configurations.

zero at the beginning of diagnosis). We observed no difference in the behavior of the diagnosis algorithm for these experiments, leading us to believe that the mechanism is robust in the presence of typical transient faults.

In order to accurately account for masking effects in our simulation environment, we extended SimpleScalar to include detailed simulation of the fault sites we inject errors at. To avoid excessive simulation times, we extended SimpleScalar only in the areas required to sufficiently evaluate the effects of masking for the injected fault. Fault sites were chosen for each of the FDUs in the system with the goal of providing a representative fault for the given structure, with nominal or slightly pessimistic behavior sought to ensure that our study would apply for the broader set of possible faults that could occur in the system.

For storage structures, we selected a representative bit to corrupt for a faulted unit. 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 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.

For combinational logic units, such as the ALUs, corrupting a single bit of output is not an accurate fault model. This is because of the fact that combinational logic differs from storage in that faults may propagate to different outputs or may be functionally masked for different inputs and operations. This

diagnose a fault's source, requiring the diagnosis mechanism to treat all errors detected by DIVA in the same fashion, with the counters for all FDUs involved in the calculation of the erroneous result getting incremented upon DIVA correction. For example, for an integer add instruction, a set would include critical logic, one integer ALU, one reservation station, one ROB entry, one IFQ entry, and one checker.

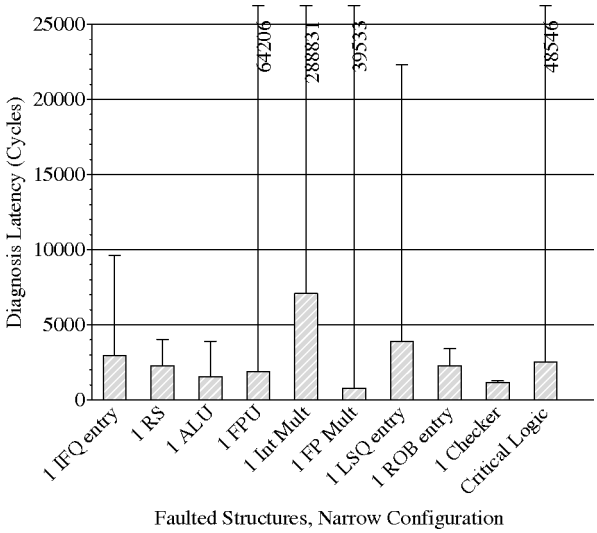

Fig. 4. Hard fault diagnosis latency, averaged over all benchmarks, for Narrow configuration.

requires us to either simulate a gate-level design of the faulted unit or to utilize a statistical fault model.

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. We then simulated all possible inputs and all possible fault locations for the adder to gain intuition on how masking affects observation of fault effects. The gate we selected for fault injection in our simulations 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.

For the integer multiplier, floating-point multiplier, and floating-point ALUs, we used a statistical model for fault injection. In this model, we assume that there is a 50% chance that data masking will mask the injected fault. We use a random number generator to select which instructions observe this data-masking effect.

In all of our experiments, the microprocessor detected and diagnosed the injected hard fault and did not misdiagnose a soft fault as being hard. We measured how many cycles elapsed before an injected hard fault was correctly diagnosed, and we plot the results of this experiment for the worst of the three configurations (Narrow) in Figure 4. The other two configurations exhibited qualitatively similar performance, so are not shown here. Since the results were relatively insensitive to the benchmarks, we present the mean results for the entire SPEC2000 benchmark suite; the error bars in the figure represent one standard deviation above the mean. The results show that most hard faults are diagnosed within fewer than 15,000 cycles, but that there are irregular diagnoses that take significantly more time, leading to a high variance in the data. These irregular diagnoses come from two sources.

Table III. Number of Diagnoses Needed to Identify Correct Failing Unit

Faulted Unit	1 Diagnosis	2 Diagnoses	3 Diagnoses	4 Diagnoses	5 Diagnoses	6+ Diagnoses
Instruction fetch queue entry	>99.99%	<0.01%	0%[a]	0%[a]	<0.01%[a]	<0.01%
Reservation station	>99%	<1%	<0.01%	<0.01%	<0.01%	<0.1%
Integer ALU	>99%	<0.1%	<0.1%	0%[a]	0%	<0.1%
Floating-point ALU	>99%	<0.1%	<0.1%	<0.01%	<0.1%	<0.1%
Integer multiplier	>99.999%	0%	0%[a]	0%[a]	0%	0%[a]
Floating-point multiplier	>99.99999%	0%	0%	0%	0%[a]	0%
Load/store queue entry	100%	0%	0%	0%	0%	0%
Rob entry	>99.99%	0%[a]	0%[a]	0%[a]	0%[a]	0%[a]
DIVA checker	>99%	<1%	<0.01%	0%	0%	0%
Critical logic	>94%	<4%	<1%	<1%	<1%	<1%

[a]Value less than 0.001%, but nonzero value.

The first source is initial misdiagnosis of nonfaulty hardware. To gain intuition on how often this will be a factor in diagnosis latency, we gathered statistics on how many diagnoses are required before converging on the correct diagnosis. In these simulations, the fault was always left active, allowing for continual diagnosis of the same faulty unit. Table III shows the results of these experiments. Because the results for all processor configurations are similar, we combine them in the data presented. While only the load/store queue entry has perfect diagnosis across all configurations, all units except critical logic are diagnosed initially with at least 99% accuracy. With critical logic, the fact that multiple units get deconfigured before the correct problem is identified is unimportant because a fault in the critical logic will require that the processor be shut down. As the latency data in Figure 4 shows, this still happens in a very short period of time. In effect, the counter threshold selection for critical logic allows the greatest opportunity for correct diagnosis of an FDU prior to drawing a conclusion that critical logic has been affected by a hard fault. Since the reaction to such a hard fault is more drastic than deconfiguring a single FDU, we feel that this is a wise design decision.

The second source of variance in diagnosis latency is programmatic phase behavior. The mix of instructions varies throughout the various phases of program operation. During certain phases, FDU utilization patterns will shift, causing diagnosis behavior to vary. In rare circumstances, a string of instructions that

causes the wrong error counter to saturate first will occur (for example, a loop that repeats many times). This can lead to a large number of misdiagnoses before the faulted unit gets properly deconfigured. As mentioned previously, the diagnosis mechanism tolerates these misdiagnoses without significant impact to the performance of the processor. The largest observed latencies were on the order of millions of cycles, which is a small amount of time for a modern microprocessor running at multiple gigahertz clock frequencies.

Our diagnosis latency study shows that the window of vulnerability for a faulty DIVA checker is, on average, around 2000 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 deconfigured, 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 cycles, this performance penalty is still negligible. The key is not incurring that performance penalty over the entire lifetime of the processor, as results in Section 7.4 show.

For the microarchitectures in our experiments, there are no spare units for the integer multiplier or floating-point multiplier. Thus, we are unable to evaluate the effects of deconfiguring these units in Section 7.3, because they are essential to correct operation of the processor. The latency and accuracy data do suggest that considering these units as FDUs is possible. In Section 7.4, we show that protecting these units from hard faults with a diagnosis and deconfiguration strategy is worth considering in future designs.

7.3 Performance after Deconfiguring FDU

The second set of experiments evaluates the performance impact of deconfiguring 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 for each processor configuration. The data show that the performance impact of deconfiguring an FDU is often small. This result, which corroborates prior work [Shivakumar et al. 2003; Srinivasan et al. 2005], is in part because of the fact that the processor configurations we are modeling are overprovisioned for single SPEC benchmarks; both of the Pentium 4-styled configurations (Narrow and Deep–Narrow) are designed to simultaneously run multiple threads and the extreme width of the Athlon-styled configuration (Short–Wide) has it provisioned with multiple units. Thus, resources are often idle in a typical single-threaded workload. There is a non-negligible performance degradation because of deconfiguring an ALU or DIVA checker in the Narrow configuration. This penalty all

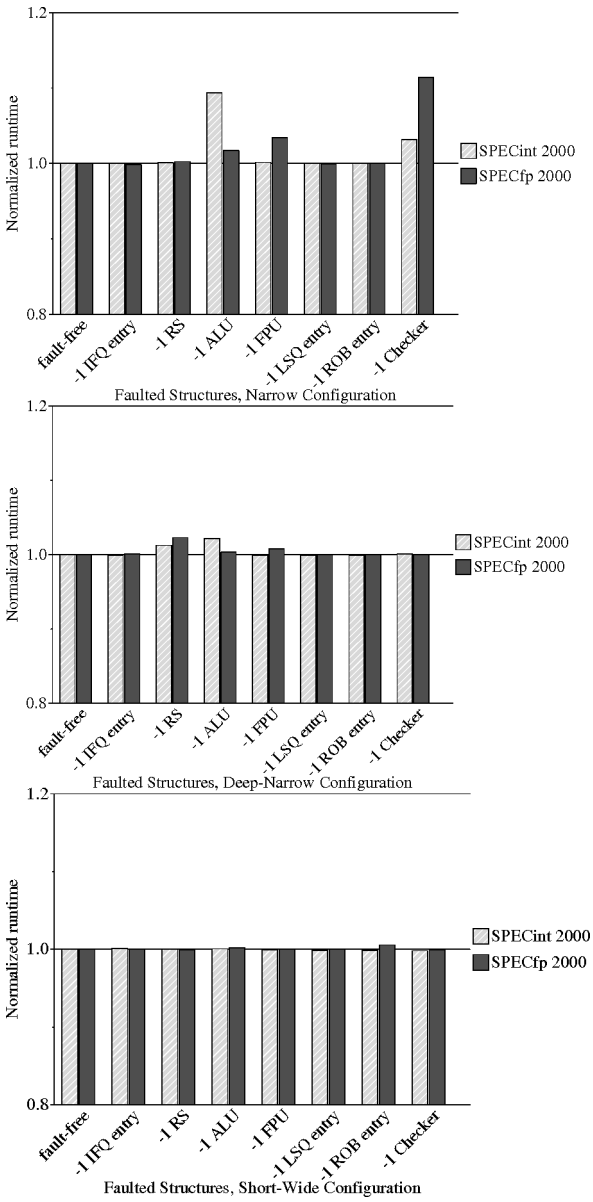

Fig. 5. Performance impact of losing one component to a hard fault for each of the three evaluated processor configurations.

but disappears in the other two configurations. In Deep–Narrow, the longer pipeline suffers more from pipeline flushes, which degrade performance to a point where the performance loss of the execute and commit bandwidth is effectively masked. In Short–Wide, the extra units provisioned to support the width of the processor effectively mask the penalty for removing a single unit. Stated another way, removing a single unit in Short–Wide is removing a smaller

percentage of available computing bandwidth than in the Narrow configurations. All of these faulty systems continue to function correctly and with reasonable performance.

7.4 Performance with Just DIVA Recovery (but No Diagnosis)

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). Figures 6 and 7 show the effects of allowing complex, combinational logic substructures with hard faults to remain in use with the DIVA checkers correcting the errors that they activate for the SPECint and SPECfp benchmarks, respectively. Figure 8 (for SPECint) and Figure 9 (for SPECfp) show the effects of allowing regular array structures with hard faults to remain in use with only DIVA correction. In all four figures, we plot runtimes that are normalized to the error-free case for each configuration, but we do not aggregate results across benchmarks, because there is significant variability. In these figures, the bar order (from left to right) matches the order of items in the legend (from top to bottom) with a full set of bars provided for each of the SPEC2000 benchmarks. We do not inject hard faults into the DIVA checkers because they cannot tolerate them without our diagnosis/reconfiguration.

In the case of the complex combinational logic units, 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 data for the singleton units in our study (the integer and floating point multipliers) shows that, for certain workloads, there is motivation to provide a less costly alternative to pipeline flushing error correction mechanisms.

For the array structures, there are many more units present in typical architectures than there are combinational logic units. Because of their greater population in modern designs, these units are naturally used less often than the combinational logic units. This functional masking effect results in the lessened effects we observe. These units are still used often enough to cause frequent pipeline flushes from DIVA corrections to noticeably, negatively impact performance.

The relative difference in magnitude of the structure-to-structure penalty is directly related to how frequently a given substructure is used by the workload. Benchmark-to-benchmark variation for a given type of FDU is a result of the distribution and frequency of preexisting stall events in a given benchmark. The causes of these events, such as cache misses or branch mispredictions, result in a

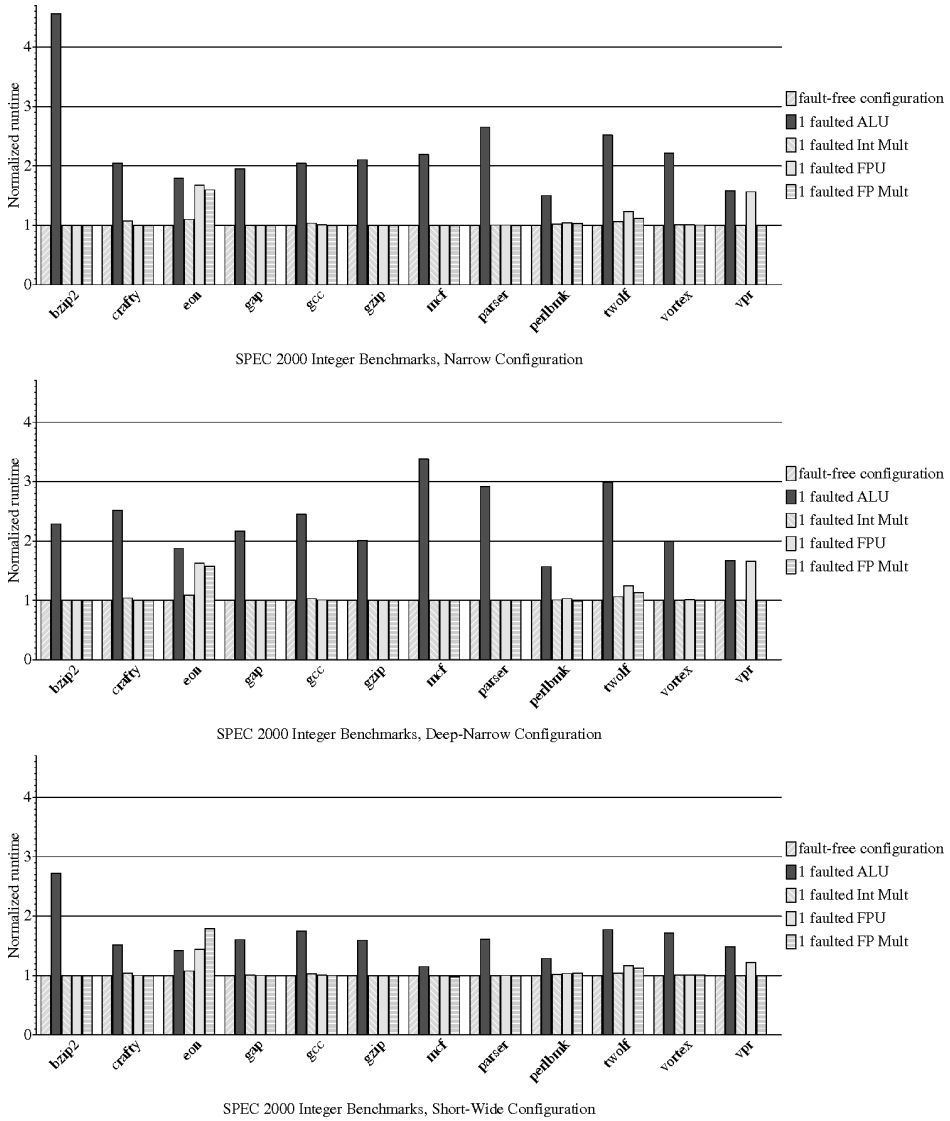

Fig. 6. Performance of DIVA-only correction for combinational logic units (SPECint).

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 resulting from 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

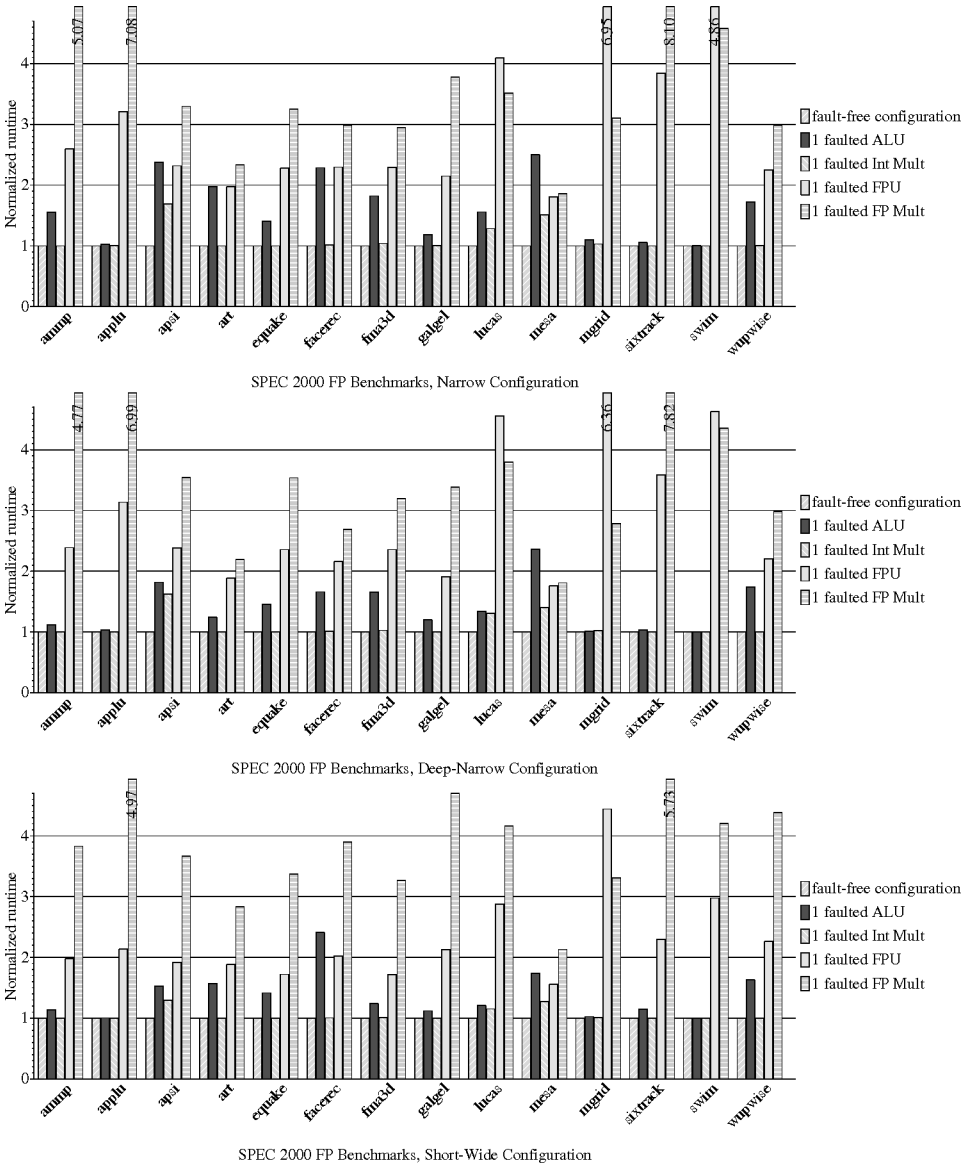

Fig. 7. Performance of DIVA-only correction for combinational logic (SPECfp).

shown that, for a variety of processor configurations, 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. This will lead to decreased yield, higher FIT rates, and lower MTTF for future generation parts. We have shown that use of a diagnosis

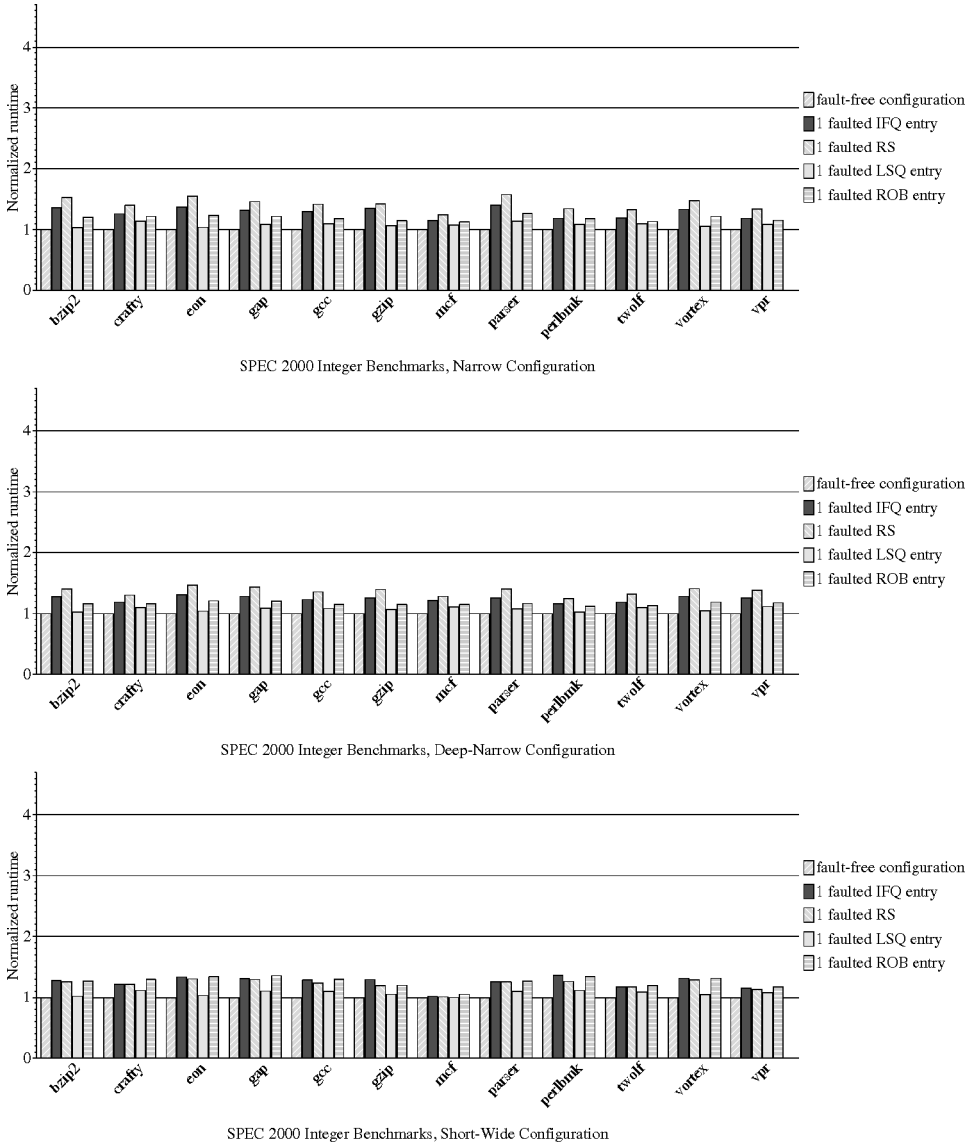

Fig. 8. Performance of DIVA-only correction for array logic units (SPECint).

and deconfiguration mechanism will allow for parts to operate in the presence of hard faults until they begin to experience larger numbers of hard faults near their end of life. This will lead to higher MTTF and lower FIT rates for parts that use this sort of scheme over their unprotected peers. 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 because

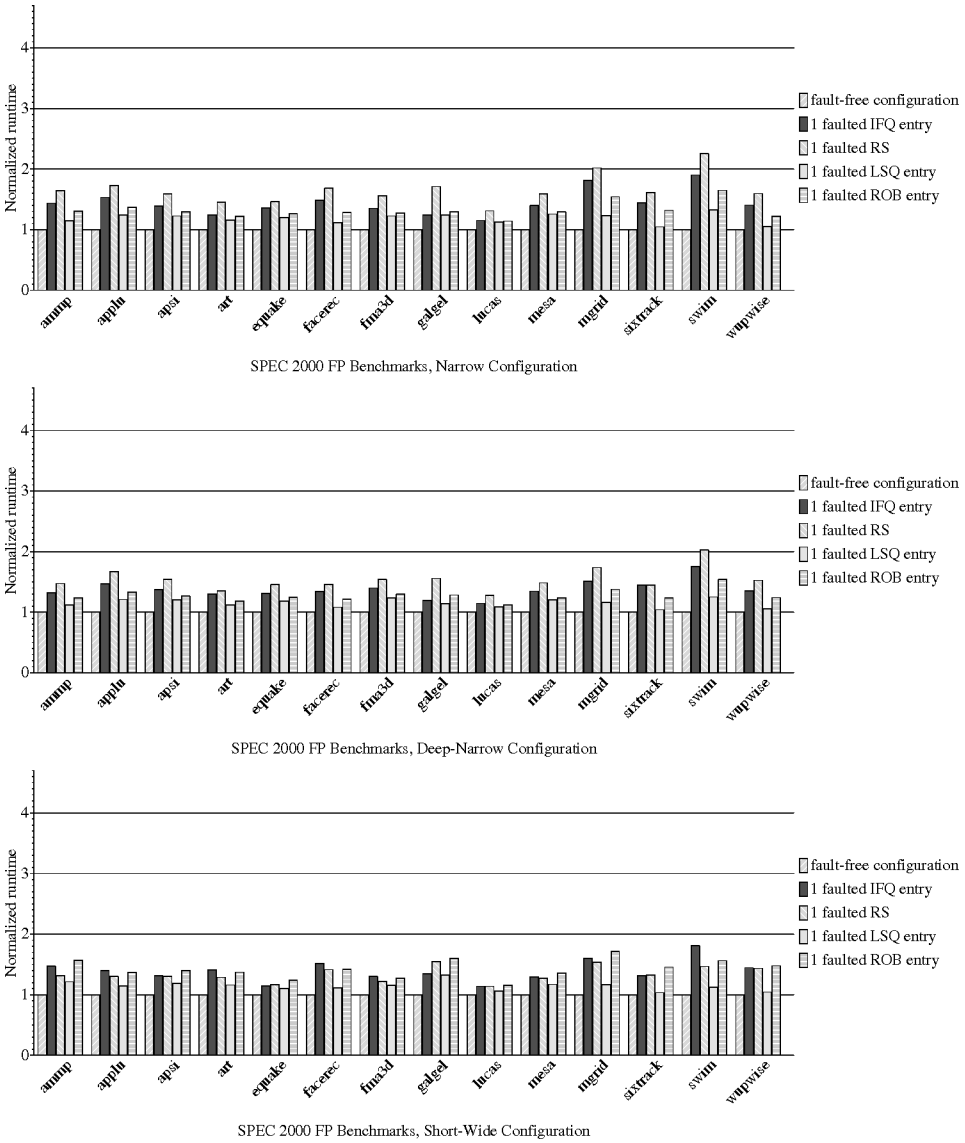

Fig. 9. Performance of DIVA-only correction for array logic units (SPECfp).

of 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. If that is the case, cold sparing of performance-essential FDUs may be employed to effectively increase the MTTF and decrease the FIT rate of a part employing our scheme. As mentioned previously, quantitative analysis of how much MTTF/FIT rate improvement will be gained is dependent upon fault rates, which are dependent upon process and design details that we do not consider in this work. Nevertheless, even without cold spares, 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 [Spainhower and Gregg 1999]. 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 [Wilson 1985] and the Tandem S2 [Jewett 1991], 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 [Culbertson et al. 1996]. 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 [Austin 1999] and redundant thread schemes provide low cost and low power alternatives to heavyweight redundancy. All of the redundant threading schemes (AR-SMT [Rotenberg 1999], Slipstream [Sundaramoorthy et al. 2000], SRT [Mukherjee et al. 2002; Reinhardt and Mukherjee 2000], and SRTR [Vijaykumar et al. 2002]) provide error detection and either use pipeline squashing for error correction or could easily provide error correction via pipeline squashing. 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. [2003] and Srinivasan et al. [2005] 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 (preshipment) 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 [Srinivasan et al. 2004a]. 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, because of 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 [Schuchman and Vijaykumar 2005], 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 noncomprehensive 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 [Spainhower and Gregg 1999]. 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 [Patterson et al. 1988]. 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 [Dell 2002; IBM 1999], and partial failures are tolerated with schemes that map out faulty locations [Chen and Sunada 1992; Mazumder and Yih 1990; Sawada et al. 1989]. For SRAM caches, techniques have been developed to map out defective locations during fabrication [Youngs and Paramandam 1997] and, more recently, during execution [Nicolaidis et al. 2003]. SRAS [Bower et al. 2004] 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.

As technology trends continue to drive higher-complexity designs, implemented with smaller transistor geometries, we believe that the incidence of hard faults will increase, both from manufacturing defects and lifetime wearout effects. In response to this increase in hard faults, commodity microprocessor designs will require that hard fault tolerance be considered in their designs.

Traditional approaches in the fault-tolerant computing space have not been limited by the same cost constraints as the commodity space, making direct application of existing techniques inappropriate. We believe that the commodity microprocessor design space will drive the following constraints into a fault-tolerant design:

- Low-cost implementation in terms of hardware and power consumption characteristics.
- Graceful degradation in performance in the presence of hard faults.
- Effective containment of lifetime-reliability induced defects.

To meet these constraints, fine-grained diagnosis schemes will be required, since coarse-grained solutions tend to incur too much performance penalty per fault tolerated. The present online techniques will have to be adapted to work in concert with existing features in the commodity design space, including low-cost error detection and correction mechanisms.

ACKNOWLEDGMENTS

We thank Alvy Lebeck and the rest of the Duke Architecture Reading Group for helpful feedback on this paper.

REFERENCES

AMD. 2005. Software Optimization Guide for AMD64 Processors. Publication 25112, Rev. 3.06 (Sept.).

AUSTIN, T. M., LARSON, E., AND ERNST, D. 2002. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer* 35,2, 59–67.

AUSTIN, T. M. 1999. DIVA: A reliable substrate for deep submicron microarchitecture design. In *Proceedings of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*. (Nov.). 196–207.

BLAAUW, D. T. ET AL. 2003. Static electromigration analysis for on-chip signal interconnects. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems 22*, 1, 39–48.

BOGGS, D. ET AL. 2004. The microarchitecture of the Intel Pentium 4 processor on 90nm technology. *Intel Technology Journal* 8, 1.

BOWER, F. A., SHEALY, P. G., OZEV, S., AND SORIN, D. J. 2004. Tolerating hard faults in microprocessor array structures. In *Proceedings of the International Conference on Dependable Systems and Networks* (June). 51–60.

CARTER, J. R., OZEV, S., AND SORIN, D. J. 2005. Circuit-level modeling for concurrent testing of operational defects due to gate oxide breakdown. In *Proc. of Design, Automation, and Test in Europe*. (Mar.). 300–305.

CHEN, T. AND SUNADA, G. 1992. An ultra-large capacity single-chip memory architecture with self-testing and self-repairing. In *Proc. of the International Conference on Computer Design (ICCD)*. 576–581, (Oct.).

CULBERTSON, W. B., AMERSON, R., CARTER, R. J., KUEKES, P., AND SNIDER, G. 1996. The teramac custom computer: Extending the limits with defect tolerance. In *Proceedings of the IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems* (Nov.).

DELL, T. J. 2002. A white paper on the benefits of chipkill-correct ECC for PC server main memory. IBM Microelectronics Division Whitepaper (Nov.).

DUMIN, D. J. 2002. *Oxide Reliability: A Summary of Silicon Oxide Wearout, Breakdown and Reliability*. World Scientific Publications. Hacksensack, NJ.

HINTON, G., SAGER, D., UPTON, M., BOGGS, D., CARMEAN, D., KYKER, A., AND ROUSSEL, P. 2001. The microarchitecture of the Pentium 4 processor. *Intel Technology Journal* (Feb.).

HUYNH, J. 2003. The AMD Athlon XP processor with 512KB L2 cache. AMD White Paper (Feb.).

IBM. 1999. Enhancing IBM netfinity server reliability: IBM Chipkill Memory. IBM Whitepaper (Feb.).

INTERNATIONAL TECHNOLOGY ROADMAP FOR SEMICONDUCTORS. 2003.

JEDEC SOLID STATE TECHNOLOGY ASSOCIATION. 2003. Failure Mechanisms and Models for Semiconductor Devices. JEDEC Publication JEP122-B (Aug.).

JEWETT, D. 1991. Integrity S2: A fault-tolerant UNIX platform. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing Systems*. 512–519 (June).

MAZUMDER, P. AND YIH, J. S. 1990. A novel built-in self-repair approach to VLSI memory yield enhancement. In *Proceedings of the International Test Conference*. 833–841.

MUKHERJEE, S. S., KONTZ, M., AND REINHARDT, S. K. 2002. Detailed Design and implementation of redundant multhreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture* (May). 99–110.

NICOLAIDIS, M., ACHOURI, N., AND BOUTOBZA, S. 2003. Dynamic data-bit memory built-in self-repair. In *Proceedings of the International Conference on Computer Aided Design* (Nov.). 588–594.

PATTERSON, D. A., GIBSON, G., AND KATZ, R. H. 1988. A case for redundant arrays of inexpensive disks (RAID). In *Proceedings of 1988 ACM SIGMOD Conference* (June). 109–116.

REINHARDT, S. K. AND MUKHERJEE, S. S. 2000. Transient fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture* (June). 25–36.

ROTENBERG, E. 1999. AR-SMT: A Microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of the 29th International Symposium on Fault-Tolerant Computing Systems* (June). 84–91.

SAWADA, K., SAKURAI, T., UCHINO, Y., AND YAMADA, K. 1989. Built-in self repair circuit for high density ASMIC. In *Proceedings of the IEEE Custom Integrated Circuits Conference*.

SCHUCHMAN, E. AND VIJAYKUMAR, T. N. 2005. Rescue: A microarchitecture for testability and defect tolerance. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture* (June). 160–171

SHERWOOD, T., PERELMAN, E., HAMMERLY, G., AND CALDER, B. 2002. Automatically characterizing large scale program behavior. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Oct.).

SHIVAKUMAR, P., KECKLER, S. W., MOORE, C. R., AND BURGER, D. 2003. Exploiting microarchitectural redundancy for defect tolerance. In *Proceedings of the 21st International Conference on Computer Design* (Oct.).

SPAINHOWER, L., AND GREGG, T. A. 1999. IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective. *IBM Journal of Research and Development 43*, 5/6.

SRINIVASAN, J., ADVE, S. V., BOSE, P., AND RIVERS, J. A. 2004a. The case for lifetime reliability-aware microprocessors. In *Proceedings of the 31st Annual International Symposium on Computer Architecture* (June).

SRINIVASAN, J., ADVE, S. V., BOSE, P., AND RIVERS, J. A. 2004b. The impact of technology scaling on lifetime reliability. In *Proceedings of the International Conference on Dependable Systems and Networks* (June).

SRINIVASAN, J., ADVE, S. V., BOSE, P., AND RIVERS, J. A. 2005. Exploiting structural duplication for lifetime reliability enhancement. In *Proc. of the 32nd Annual International Symposium on Computer Architecture* (June).

SUNDARAMOORTHY, K., PURSER, Z., AND ROTENBERG, E. 2000. Slipstream processors: Improving both performance and fault tolerance. In *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems* (Nov.). 257–268.

TAO, J., CHEN, J. F., CHEUNG, N. W., AND HU, C. 1996. Modeling and characterization of electromigration failures under bidirectional current stress. *IEEE Transactions on Electron Devices 43*, 5, 800–808.

TULLSEN, D. M., ET AL. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. of the 23rd Annual Int'l Symp. on Computer Architecture* (May). 191–202.

VIJAYKUMAR, T. N., POMERANZ, I., AND CHUNG, K. K. 2002. Transient fault recovery using simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture* (May). 87–98.

WEAVER, C. AND AUSTIN, T. 2001. A fault tolerant approach to microprocessor design. In *Proceedings of the International Conference on Dependable Systems and Networks* (July). 411–420.

WILSON, D. 1985. The stratus computer system. In *Resilient Computer Systems*. 208–231.

YOUNGS, L. AND PARAMANDAM, S. 1997. Mapping and repairing embedded-memory defects. *IEEE Design & Test of Computers* (Jan.–Mar.) 18–24.