COPYRIGHT NOTICE

Autonomic Microprocessor Execution via Self-Repairing Arrays

Fred A. Bower, *Student Member*, *IEEE*, Sule Ozev, *Member*, *IEEE*, and Daniel J. Sorin, *Member*, *IEEE*

Abstract—To achieve high reliability despite hard faults that occur during operation and to achieve high yield despite defects introduced at fabrication, a microprocessor must be able to tolerate hard faults. In this paper, we present a framework for autonomic self-repair of the array structures in microprocessors (e.g., reorder buffer, instruction window, etc.). The framework consists of three aspects: 1) detecting/diagnosing the fault, 2) recovering from the resultant error, and 3) mapping out the faulty portion of the array. For each aspect, we present design options. Based on this framework, we develop two particular schemes for self-repairing array structures (SRAS). Simulation results show that one of our SRAS schemes adds some performance overhead in the fault-free case, but that both of them mask hard faults 1) with less hardware overhead cost than higher-level redundancy (e.g., IBM mainframes) and 2) without the per-error performance penalty of existing low-cost techniques that combine error detection with pipeline flushes for backward error recovery (BER). When hard faults are present in arrays, due to operational faults or fabrication defects, SRAS schemes outperform BER due to not having to frequently flush the pipeline.

Index Terms—Logic design reliability and testing, microprocessors, and microcomputers.

1 Introduction

As computers comprise ever more of society's infrastructure, microprocessor reliability becomes increasingly crucial. Unfortunately, the same technological trends that are enhancing microprocessor performance are also undermining reliability and yield. As transistors and wires continue to shrink in accordance with Moore's Law, the probabilities of several types of faults increase. In this paper, we consider hard (i.e., permanent) faults due to both fabrication defects as well as operational phenomena such as electromigration and gate oxide breakdown [36], [7], [25], [17]. Although burn-in testing can quickly uncover some defects which cause reliability risks [28], [4], it is typically only used on small sample sizes due to its high cost. Moreover, for deep submicron devices, the rates of early life failures, even after burn-in, are increasing due to the progressive nature of such defects [4]. In this paper, we focus on hard faults in microprocessor array structures, since these structures are the single-largest consumer of microprocessor core die area, comprising up to 33 percent of the area of microprocessor core (i.e., not including caches) in recent microprocessor designs [34].

To improve fabrication yield and to achieve high reliability in the presence of hard faults, a microprocessor must be able to detect and diagnose them and then mask their effects. Previous work has explored how to tolerate hard faults in microprocessors, but it has suffered in terms of performance and implementation costs. The classic approach is to completely replicate the microprocessor (or components within it), as in IBM mainframes [35]. This approach is effective, but quite costly in terms of hardware and power. As a lighter weight alternative, DIVA dynamic verification [3] can tolerate hard faults in the microprocessor (but not in its checker logic). However, DIVA, which was designed primarily for handling soft faults and corner-case design defects, incurs a large performance penalty for every error from which it must recover, which is a significant problem for a hard fault in a frequently accessed structure.

In this paper, we discuss the design space for self-repairing microprocessor array structures (SRAS) and we present two specific designs. Array structures include the reorder buffer, load-store queue, instruction queue, branch history table, etc. Protecting the combinational logic, such as ALUs, is a complementary but orthogonal problem that we discuss briefly but leave for future work. Our goal is to develop self-repairing arrays that enable autonomic execution. In both of our SRAS designs, spare rows are built into each array structure and are mapped in to replace faulty rows using a level of indirection. This approach is similar to how disks map out faulty sectors and how hard faults in DRAMs can be tolerated with schemes that map out faulty locations [10], [21], [31]. Our first design, SRAS-CheckRow (SRAS-CR) [8], uses dedicated check rows to detect and diagnose hard faults. SRAS-CR relies upon DIVA to recover from transient errors and errors due to hard faults that have not yet been classified as hard. Our second design, SRAS-EDC, uses error detecting codes (EDC) for error detection/diagnosis, and it uses the preexisting branch misprediction recovery mechanism to recover from transient errors and errors due to hard faults that have not yet been classified as hard. After a hard fault has been diagnosed and mapped out, neither SRAS-CR nor SRAS-EDC incurs a performance penalty due to that

- *F.A. Bower is with IBM and the Department of Computer Science, Duke University, PO Box 90129, Durham, NC 27708. E-mail: fredb@cs.duke.edu.*
- *S. Ozev and D.J. Sorin are with the Electrical and Computer Engineering Department, Duke University, PO Box 90291, Durham, NC 27708. E-mail: {sule, sorin}@ee.duke.edu.*

Manuscript received 30 Aug. 2004; revised 14 Oct. 15; accepted 19 Oct. 2005; published online 3 Nov. 2005.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0126-0804.

fault, unlike lightweight schemes that incur a costly recovery for every manifestation of a hard fault.

Our experimental results show that SRAS-EDC adds some performance overhead in the fault-free case, but that both SRAS-CR and SRAS-EDC mask hard faults 1) without the hardware costs of high-level redundancy (e.g., IBM mainframes) and 2) without the per-error performance penalty of existing low-cost techniques (e.g., DIVA). When hard faults are present in arrays, due to operational faults or fabrication defects, then our SRAS schemes outperform low-cost techniques that require a pipeline recovery per error. Given the increasing frequencies of fabrication defects and operational hard faults, the likelihood of wanting to be able to operate correctly with one or more hard faults makes array self-repair appealing.

The rest of this paper is as follows: Section 2 presents a survey of related work, focusing on techniques that might be applied to the problem of hard faults in the microprocessor core. Section 3 provides background on our hard fault model. The underlying physical phenomena that lead to hard faults are discussed in some detail to familiarize the reader with these mechanisms as well as to further motivate the case for providing hard-fault tolerance in coming microprocessor core designs. Section 4 presents SRAS-CR and SRAS-EDC in detail, explaining the mechanisms, how they operate, and their limitations and advantages in the application of providing hard-fault tolerance to microprocessor core array structures. Section 5 presents a detailed performance evaluation of SRAS-CR and SRAS-EDC, and the paper concludes with Section 6.

2 Related Work

There is a historical progression of designs that has led up to this point and motivated the work in this paper. A canonical design for autonomic operation is the IBM mainframe [35]. 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 [46] and the Tandem S2 [18], simply replicate entire processors. While these systems all provide excellent reliability, such heavyweight redundancy incurs significant costs in terms of hardware and power consumption.

As a low cost and low power alternative to heavyweight redundancy, DIVA [3] dynamically verifies an aggressive microprocessor core with a simple, provably correct checker core. DIVA sacrifices some amount of reliability in order to greatly reduce these costs. DIVA's small amount of redundancy is far less costly than mainframe redundancy, but it incurs significant performance and energy penalties for each error that it must correct. Each error detected and corrected by the checker core triggers a pipeline flush of the aggressive core. Since DIVA was designed primarily for soft faults (not the hard faults we address in this paper), these flushes are not a performance problem. However, permanent faults in frequently accessed structures, such as the reorder buffer, will frequently manifest themselves as errors and will thus greatly degrade performance. Researchers have also proposed using redundant threads to achieve lightweight redundancy, primarily for soft faults. Of these schemes, the ones that perform recovery as well as error detection include AR-SMT [30], Slipstream [38], and SRTR [42]. These schemes share the same drawback as DIVA, with respect to hard faults, since they incur a pipeline squash every time a hard fault manifests itself. Redundant thread schemes, unlike DIVA, may not be able to guarantee forward progress in the presence of hard faults.

One option for array structures is to protect them with error correcting codes (ECC), as in IBM mainframes [35]. Combining ECC for arrays with DIVA avoids costly recoveries. However, ECC protection of arrays is on the critical path for array access (both read and write). Current ECC implementations can calculate ECC on a representative datum in four cycles on a 2 GHz Itanium2 [45]. At the 4 GHz speeds at which current commodity microprocessor designs run, this becomes a 7-cycle overhead. Since ECC must be calculated on the microprocessor's critical path, a 7-cycle penalty per ECC calculation results in highly-degraded performance, even in the fault-free case. This lost performance makes ECC inappropriate for application in the timing-critical pipeline.

With the advent of chip multiprocessing (CMP) in commodity microprocessor designs, another hard-fault tolerance option is to disable any core that is detected to have a hard fault. While this works, we seek to provide a more cost-effective option than to lose 1/Nth (for an N-core design) of the chip's capacity for each hard fault that is detected. Shivakumar et al. [34] propose a more cost-effective solution that utilizes inherent redundancy in CMP and SMT designs. This work is limited to manufacturing-time detection (i.e., testing) and deconfiguration, whereas SRAS provides a means for both manufacturing-time and in-situ operational detection and deconfiguration of subunits within the microprocessor core.

Table 1 summarizes all of these techniques, including our SRAS-CR and SRAS-EDC designs. Included are the original fault-tolerance targets of the techniques (soft, hard, or design) and notes on the limitations of using these in a commodity microprocessor design. Each technique has certain advantages and certain disadvantages which make it more or less appropriate for a given design space. As our results show in Section 5, SRAS-CR and SRAS-EDC provide a performant solution in the performance-conscious commodity microprocessor design space.

3 Hard Faults in Submicron CMOS Technology

In this section, we present existing high-level models for hard faults (Section 3.1) and then we delve into the underlying physical phenomena that cause hard faults (Section 3.2). In this process, we show that existing fault models are applicable to the physical faults we consider in this paper.

3.1 Fault Models

To facilitate fault tolerant design and testing for physical faults that lead to errors at the circuit level, several structural fault models have been developed for logic circuits and storage components over the past few decades

TABLE 1
Fault Tolerance Techniques: Design Points and Limitations

Technique	Fault Target(s)	Limitations of Use for Hard-Fault Tolerance in Microprocessor
DIVA	Soft, Design	Excessive performance penalty for frequent pipeline flushes due to faults in frequently-accessed structures
Redundant Multithreading	Soft	May be subject to livelock. Excessive performance penalty for frequent pipeline flushes due to faults in frequently-accessed structures
Triple Modular Redundancy (TMR)	Soft, Hard	Over 3x cost in terms of die area and power consumption over unprotected core design point
CMP Core Sparing	Hard	High performance penalty per hard fault (1 of N cores) in designs where N is relatively small
ECC	Soft, Hard	Adds excessive latency to critical path of microprocessor. Only localized fault tolerance
Microarchitectural Redundancy Exploitation	Hard	Manufacturing-time only, as described in [34]. Only localized fault tolerance
SRAS-CR	Hard	Requires fault detection mechanism to trigger hard-fault tolerance. Only localized fault tolerance
SRAS-EDC	Hard	Adds latency to fault-free operation of microprocessor. Only localized fault tolerance

[1]. The *stuck-at fault model* is the most commonly used model in VLSI testing and fault tolerance schemes. In this model, a physical defect manifests itself as a signal consistently having a certain value (either zero or one) independent of the input. For example, an unintended short circuit between the two inputs of an XOR gate results in a stuck-at-zero fault at the output signal. The *coupling fault model*—in which a write to a certain memory location always prompts a write to a neighboring location or locations—has been defined for storage components [9]. The recently defined *transition fault model* represents a slow charging or discharging of a circuit node [26], [32], [41]. This delay can cause incorrect logic values to be latched. Next, in Section 3.2, we see that the stuck-at and coupling fault models will be sufficient for the hard faults that we consider.

3.2 Underlying Physical Phenomena

The reliability of electronic devices under discrete environmental stress, such as radiation [37], and continuous functional stress due to the applied electric field [29], [39], [6] has been a topic of vast research since the early days of semiconductor manufacturing. Extensive research has been conducted on the failure-causing physical phenomena, such as electromigration [19], [39], [6] and transistor gate oxide breakdown (OBD) [12]. Electromigration results in highly resistive interconnects or contacts and eventually leads to open circuits. Such defects are typically modeled as transition faults during manufacturing testing, but they become stuck-at faults during operation due to their progressive nature.

OBD results in the malfunction of a single transistor due to the creation of a highly conductive path between its gate and its bulk. The onset of this phenomenon is called a soft breakdown (SBD); however, after several SBD incidents, the oxide layer diffuses and highly conductive melted metal fills the void and solidifies into a consistent path. This phenomenon is called hard breakdown (HBD). Similar to the electromigration case, the initial circuit level manifestation of SBD is a transition fault, whereas the effect of the subsequent HBD is a stuck-at fault. OBD defects are potentially more dangerous than electromigration defects due to the consistent path between a charged node and ground or supply. Thus, detection and isolation of memory locations with OBD defects is essential for the operational health of computing devices.

Both the electromigration and OBD defects are progressive in nature. The mean time to failure (MTTF) for both defects depends on the thickness and the initial health of the structure. Reported laboratory data on OBD indicates that MTTF is on the order of four million seconds (around 46 days) for 15Å gate oxides under constant stress of 2.1V [20] (scaling the supply voltage down to 1.0V, we can estimate the MTTF for this oxide thickness to be 375 days). In the early stages of the progression of both electromigration defects and OBD defects, bit errors only occur if the defects are sequentially excited. However, in later stages, both defects resemble stuck-at faults. Moreover, in addition to affecting the output node to which the defective transistor is connected, the OBD defects may result in coupling faults due to their current driving nature. Thus, in the experiments in Section 5, we inject stuck-at faults and coupling faults since they correspond to the manifestations of electromigration and OBD defects.

4 Self-Repairing Arrays

Technology and microprocessor architecture trends are leading toward larger array structures within microprocessors. These structures include the instruction queue, reorder buffer (ROB), register file, reservation stations, register map table, branch history table (BHT), etc. We would like to protect these structures from hard faults as the probability of hard faults continues to increase, but we cannot afford to fully replicate these structures. Thus, our SRAS schemes protect array structures in a fashion similar to the way in which existing techniques protect large memory storage

structures. The basic idea is to use a level of indirection to map out faulty portions of the structure. Especially as structures grow larger, the probability of a hard fault within them increases. For DRAM main memory, whole chip failures are tolerated by chipkill memory and RAID-M [11], [15], and partial failures are tolerated with schemes that map out faulty locations [10], [21], [31]. For SRAM caches, techniques have been developed to map out defective locations during fabrication [48] and, more recently, during execution [24]. While providing insight for the use of spare memory locations for repair, direct application of the aforementioned methods to array structures within the processor bears little hope due to the performance criticality within microprocessors.

In the rest of this section, we discuss the types of arrays that we will protect (Section 4.1), and we present the design space for self-repairing arrays (Section 4.2). We then present two specific implementations (Section 4.3 and Section 4.4). We finally discuss the details of applying SRAS to certain specific microprocessor structures (Section 4.5).

4.1 Microprocessor Array Structures

We can classify array structures within the microprocessor core into two categories: nonaddressable buffers for which the data location is determined at the time of access, and randomly addressable tables for which the data location is determined before access. In order to allow timing efficient implementation of the repair logic, we exploit these distinct features of each type of array structures. Without loss of generality, we focus the discussion of SRAS on one specific array structure from each of the two categories: the reorder buffer (ROB) and the branch history table (BHT). The ROB and BHT are representative of the kinds of array structures found in modern microprocessors, and, thus, the arguments and results here apply broadly. We discuss the details of other structures, as well as the impact of content-addressability in the ROB, in Section 4.5.

4.1.1 Reorder Buffer

The ROB is a circular buffer that is used in dynamically scheduled (also known as "out-of-order") processors to implement precise exceptions by ensuring that instructions are committed in program order. We focus on processors that perform implicit register renaming with reservation stations—such as the Intel PentiumPro, IBM PowerPC, and AMD K6—in which an ROB entry contains the physical register tags for the destination register and the data result of the instruction. Alternative ROB designs exist, in which ROB entries do not hold the data results of completed instructions (data is instead held in the physical registers). Designing SRAS for these alternative designs is straightforward and actually simpler (but not discussed in this paper). ROB sizes are on the order of 32-128 entries, which is large enough to have a nonnegligible probability of a hard fault. The ROB has a high architectural vulnerability factor [23], in that a fault in an entry is likely to cause an incorrect execution. A fault in an ROB entry is not guaranteed to cause an incorrect execution for its instruction, though, since the fault might not change the data (i.e., logical masking) or the ROB entry might correspond to a squashed instruction (i.e., functional masking).

4.1.2 Branch History Table

The BHT is a table that is accessed during branch prediction. Common two-level branch predictor designs [47] use some combination of the branch program counter (PC) and the branch history register (BHR) to index into a BHT. The BHR is a k-bit shift register that contains the results of the past k branches. The indexed BHT entry contains the prediction (i.e., taken or not taken, but not the destination). In this paper, we focus on the gshare two-level predictor [22], in which the BHT is indexed by the exclusive-OR of the branch PC and a global BHR. Since the BHT is a table, our remapper implementation for it is fairly similar to the logical abstraction presented earlier. The BHT has an architectural vulnerability factor of zero, in that no fault in it can ever lead to incorrect execution. However, a BHT fault can lead to incorrect branch predictions, which can degrade performance.

4.2 Design Space

Self-repairing arrays require three features, and the designs of each collectively comprise the design space:

1. **Detection of errors and diagnosis of faults**. How does the hardware detect an error in an array, and then how does it isolate which part of the array is faulty? While there are several schemes for dynamically verifying microprocessor execution as a whole [30], [27], [3], they sacrifice diagnosis capability in order to not degrade performance.
2. **Recovery from errors**. How does the hardware recover from an error such that it can ensure that the error does not propagate corrupted data into committed architectural state? The most basic option for recovery is to halt the system when an error is detected (fail-stop), thereby protecting system state from being corrupted, at the cost of more downtime and, thus, less availability. Other alternatives exist, such as using the microprocessor's branch misprediction recovery mechanism.
3. **Mapping out faulty subarrays**. Once the faulty subarray (e.g., row or column) has been diagnosed, how does the hardware map it out and thus avoid future manifestations of this fault? The design choices for this aspect mainly involve the granularity of mapping, e.g., row, column, or even the whole array. Another design decision is the number of spares to provide. These design decisions may be influenced by the array's position in the microprocessor pipeline, particularly if accessing the array is on the critical path and performance is thus crucial.

There are numerous design decisions for each of these three aspects, but the decisions for each aspect are not completely independent. For example, ECC protection of arrays would serve as the detection and recovery mechanism, and it does not require remapping, provided that the errors do not exceed the correction abilities of the chosen correction code.

The design decisions, particularly for the recovery mechanism, also determine which array structures can be protected. For example, since SRAS-EDC uses the

misprediction recovery mechanism, it cannot tolerate errors in the recovery state (i.e., committed architectural state, such as the register file or condition codes).

4.3 SRAS-CheckRow (SRAS-CR)

The first SRAS design that we present, SRAS-CheckRow (SRAS-CR), uses dedicated check rows to detect and diagnose errors in array rows. SRAS-CR protects each array structure in isolation, i.e., the decision to protect an array with SRAS does not affect the decision to protect any other array. We will see in Section 4.4 that SRAS-EDC differs in that it is an integrated approach for protecting multiple arrays.

4.3.1 Detection and Diagnosis

SRAS-CR uses DIVA for end-to-end error detection and correction. However, DIVA cannot isolate the row or even the structure that is faulty. Thus, SRAS-CR combines DIVA with a simple scheme for detecting row errors and diagnosing which row is faulty. SRAS-CR adds a handful of check rows (some are spares, which are used to avoid a single point of failure) to each structure we wish to protect. For buffer structures such as the ROB, each time an entry is allocated, initialization data is written to both the entry and the check row. This initialization data consists of the available target data for the entry (for example, the source and destination register tags for an ROB entry) and pseudorandom data for the parts of the entry that will be written later (for example, the actual result value for an ROB entry). Where pseudorandom data is needed, the hardware tick counter (which records the machine's current uptime) is used, with appropriate scaling to provide the proper number of bits to fully populate the entry. For tables, every write to a location will have a mirrored write to the structure's affiliated check row. Any partial write to a row must be implemented as a read-modify-write (RMW) action in order to support SRAS-CR checking. The issue here is that the check row and array entry to be checked must have identical data written into their contents in order for a meaningful comparison to be made. Immediately after the two writes, both locations are read and their data are compared (all off the critical path of execution). If the data differ, then one of the rows is faulty. The converse is not true, however. Transient errors that occur between writing the row and using it will not be detected in this way; however, this does not affect SRAS-CR, since DIVA will detect and correct these transient errors, and we do not want to diagnose this row as having a hard fault.

Several options exist for determining which one is faulty, and we will explain a simple one after we first describe the mechanism we exploit for distinguishing hard faults from soft faults. SRAS-CR maintains small saturating counters for each row, which are periodically reset, and a counter value above a threshold identifies a hard fault. Now, to determine if the operational row or the check row is faulty, we can simply increment both of their counters in the case of a mismatch in their values, as long as we initially set the threshold for check row counters to be much higher than that for operational rows.

Detection and diagnosis is the same for both tables and buffers. While we logically need only k check rows in a k-way superscalar processor to detect and diagnose faults, the SRAS-CR implementation may necessitate having even more check rows. Having only k check rows could lead to an unreasonably long delay to transfer the data along wires from one end of the array to the other. Wire delays are already a problem in multi-GHz microprocessors—for example, the Intel Pentium4 has multiple pipeline stages allocated strictly to wire delay—and we cannot ignore them in our design. A simple option is to divide the array into subarrays, each of which has k check rows.

4.3.2 Recovery

If an error is detected, but the hard fault threshold has not yet been reached, then the fault is considered transient and is tolerated with a DIVA recovery and its associated performance penalty. If the detected error raises the counter to the threshold, then DIVA also tolerates this fault, but the system then repairs itself so as to prevent this hard fault from being exercised again.

4.3.3 Mapping Out Faulty Subarrays

We logically add a level of indirection that can map out faulty rows in microprocessor array structures. The remapper serves as the interface between the array and the rest of the microprocessor. The repair actions taken depend on whether the faulty row is a noncheck row or a check row. If it is a noncheck row, then it can be immediately mapped out and a spare row can be mapped in to take its place. The spare row can get the correct data from the check row. If the faulty row is a check row, then SRAS-CR maps in a spare check row. While remapping with a level of indirection is straightforward in the abstract, implementing it in a high-performance pipeline requires careful consideration. We now present remapper implementations for the ROB and BHT.

ROB Remapper. In buffer structures, the address of the data to be accessed is determined at the time of the access. Typically, two pointers are used to mark the head and the tail of the active rows. When a new ROB entry is allocated, the tail pointer is advanced and the corresponding address becomes the physical address of the data. Similarly, when an entry is removed, the head pointer is advanced. Thus, the physical as well as logical address of the data is abstracted and all rows have the same functionality. The faulty row can easily be mapped out by modifying the pointer advancement logic when a hard fault is detected. Fig. 1 illustrates the implementation of the remapping mechanism for buffers, with SRAS-CR hardware shaded in gray. SRAS-CR uses a shifted fault map bit-array to track faulty rows. If a row is determined to contain a hard fault, the faulty bit in the previous row is set to 1. The fault map is used by the pointer advancement circuit to determine how far the pointer needs to be advanced. Upon the reception of a dispatch signal, the pointer is advanced by one or two depending on whether the next row is fault or not. The shifted faulty row information enables the preprocessing of the pointer advance logic. Upon the reception of the commit signal, the head pointer is advanced in the same manner. Once the pointer is updated accordingly, reads and writes of the buffer entries proceed unmodified. Since the preprocessing for pointer advancement can be done off the critical path, the proposed modification does not impact the read or write access time.

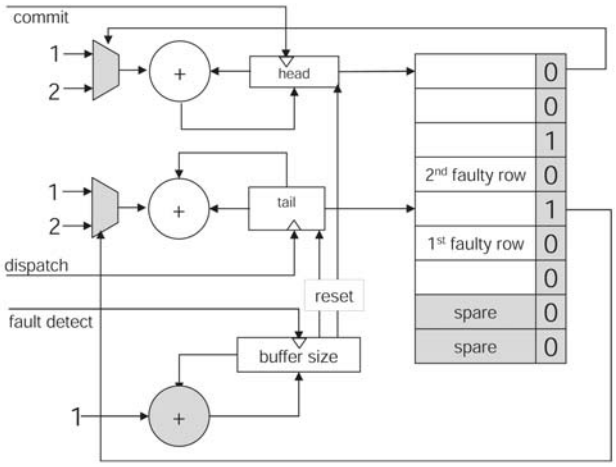

Fig. 1. Self-repair for buffers.

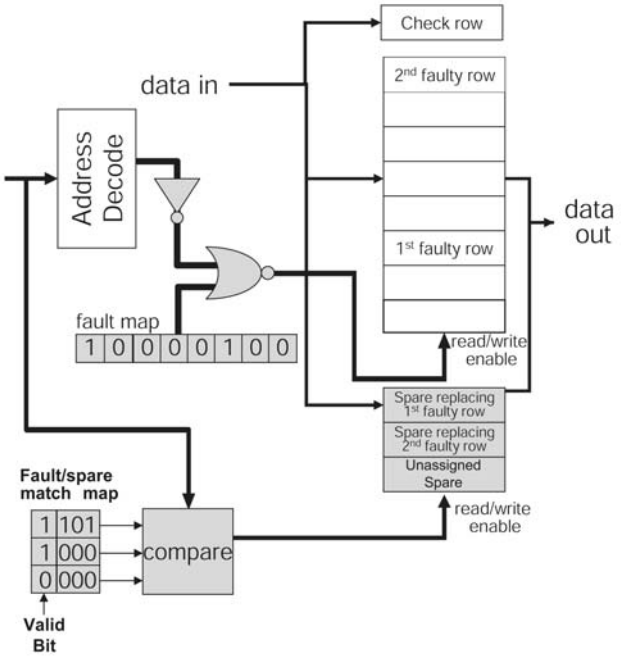

Fig. 2. Self-repair for tables.

In order to avoid a reduction in the effective buffer capacity due to hard faults, spare rows can be used. Since there is no need to replace the faulty row with any particular spare row, the detection of the faulty row prompts incrementing the total buffer capacity by one entry (by adding the spare) while maintaining the same effective capacity. SRAS-CR can tolerate as many hard faults as there are spares without any degradation of buffer performance. If the number of faulty rows exceeds the number of spare rows, then the effective buffer capacity is allowed to shrink, resulting in graceful degradation of the buffer performance. Assuming that adding one or two to the pointers does not dramatically change timing or power consumption, the only overhead of this repair mechanism is the small additional area taken by the fault map and the additional power consumed for pointer preprocessing, updating fault map entries, and updating the buffer size. Section 4.3.4 discusses the overall overhead of the complete SRAS-CR architecture in more detail.

BHT Remapper. In tables, the logical address of the data is determined by the program execution prior to accessing the data. Since rows do not have equal functionality, a faulty row needs to be replaced by a specific spare row. In this case, we need a logical indirection to map out the faulty rows. This problem is quite similar to the memory repair problem, and many online repair mechanisms have been proposed [11], [21]. However, in microprocessor array structures, logic inserted into the critical path directly impacts performance, so we must implement a timing-efficient repair mechanism. In SRAS-CR, we distribute spare rows over subarrays of the table, and a spare can only replace a row within its own subarray. This choice may make the use of spares inefficient for highly localized faults, but it enables timing efficient implementation of the repair logic, as shown in Fig. 2. Once again, hardware for SRAS-CR is shown in gray.

Similar to the buffer case, we keep the fault map information in a table. However, we also use a fault/spare match map which contains information on which functional row each spare row is replacing. If a row is identified faulty and an unused spare is found to replace it, the corresponding bit in the fault map is set to 1. In addition, the physical address of the faulty row is written into the corresponding entry of the fault/spare match map. In order to prevent an illegal write or read of the spare rows, the fault/spare match map also needs to contain one bit to indicate whether the address in the map is valid. In the example shown in Fig. 2, we can see that the first spare is allocated to row 5 and the second spare is allocated to row 0, hence the 1 in the fault map at the zero and the sixth positions. To indicate that the third spare is not allocated, the first bit of the fault/spare match map is set to zero. The address decode logic, which is present in all tables, enables a row of the table to be read or written by generating the individual read/write enable signals for the table rows. During a read or write access, these signals are modified by the remap logic to generate the updated read/write enable signals for the table entries as well as the read/write enable signals for the spare entries. The remap logic consists of n inverters and n 2-input NOR gates, where n is the size of the subarray. To generate the read/write enable signals for the spare rows, $klog(n)$ 2-input XOR gates and the equivalent of $k(log(n)+1)$-input NOR gates (denoted by the compare block in Fig. 2) are needed, where k is the number of spares assigned to the subarray.

Assuming the compare logic can execute faster than the address decode logic, SRAS-CR will add two gate delays (one inverter and one NOR gate delay) to the table access time. Since the additional level of indirection for accessing the physical table entries is on the critical path, this additional time cannot be ignored. In order to avoid set-up or hold time violations, we very conservatively use a second pipeline stage to access the table entries. This additional pipeline stage will impose a penalty in the normal mode of operation. While we expect that the actual performance penalty would be far less than a pipeline stage (e.g., if BHT access latency is not the determining factor in pipeline stage latency), we choose this pessimistic design point as a lower bound on SRAS's benefit. In Section 5, we run experiments to

assess the impact of this additional pipeline stage on the execution time in the absence of hard faults.

4.3.4 SRAS-CR Costs

The cost of a fault tolerance scheme has three aspects: hardware (area) overhead, performance (timing) overhead, and power consumption overhead. For aggressive microprocessor architectures, the performance overhead during fault-free execution is often the most critical parameter.

In order to keep the performance overhead at a minimum, buffers and tables are handled differently in SRAS. The distinct nature of buffers that makes all of their rows have equal functionality enables a no-timing-overhead implementation. Tables, however, require a definitive logical address for the data, which results in a need for an additional level of indirection. This indirection results in two gate delays in access times (e.g., for the Pentium4, an inverter delay is about 1-2 percent of the clock period [40]). Since gate delay will be larger than inverter delay, and since we cannot know how much margin exists in an existing design, we very conservatively add a pipeline stage for access to tables. The additional pipeline stage results in increased latency and an increased number of stalls, and we quantitatively evaluate its performance overhead in Section 5.

The increase in power consumption in SRAS-CR stems mostly from increased data read/write activity due to the check rows. Since the write/read activity is doubled, the dynamic power consumption in the array structures will roughly be doubled as well. If power consumption is still a concern, accesses to check rows can be reduced at the expense of increasing the fault detection latency.

Finally, the hardware overhead of SRAS-CR includes the need for:

1. DIVA,
2. spare rows (including spare check rows),
3. one logic circuit for repair and check per array structure,
4. the per-row counters for diagnosing hard faults, and
5. two additional read ports and one additional write port on the protected array structures to support simultaneous writing of the check row and reading of the result and check rows.

Among these hardware costs, the two most significant are DIVA and the extra ports on the array structures. According to Weaver and Austin [44], a DIVA checker's size is less than 5 percent of an Alpha 21264 core. The extra ports on each array structure are more problematic since they can significantly increase the structure's size and degrade its access latency. Particularly, for large array structures, the cost of the extra ports may prohibit the use of SRAS-CR; fortunately, SRAS-EDC does not have this cost.

4.3.5 Limitations of SRAS-CR

The implementation of SRAS-CR in this paper does not tolerate all microprocessor faults. We divide these untolerated faults into three categories. First, SRAS-CR does not tolerate faults in its own logic, e.g., the pointer remapping logic or the fault map. These structures are far smaller than the structures they are protecting, which makes them less prone to hard faults, but they could still fail. Second, SRAS-CR does not tolerate a fault in a table subarray if no more spare rows are available in that subarray. This limitation does not apply to buffers except in the extreme case in which every row of the buffer, including spares, is faulty. Third, SRAS-CR does not tolerate a fault in a subarray (for a buffer or table) if all of the check rows for that subarray are faulty.

All of these untolerated faults present the designer with a classic engineering trade-off: fault tolerance versus hardware cost. Future SRAS-CR implementations could develop hardened logic if the first fault model is considered important. The probabilities of the latter two categories can be decreased by designing the SRAS-CR protection to use more spare rows and more check rows.

4.4 SRAS-EDC: Self-Repair Design Without DIVA Backstop

In this section, we present a design for array self-repair that is independent of DIVA and that is fully integrated into the microprocessor datapath. The design attempts to minimize the amount of logic, particularly on critical paths. An illustration of our design (simplified for purposes of illustration) is shown in Fig. 3. As we mentioned previously in Section 4.1, the microarchitecture is similar to that of the Intel PentiumPro in that the reorder buffer holds the results of completed but not yet committed instructions (rather than keeping them in the physical register file). The array structures we protect are the instruction buffer, instruction scheduling window, reorder buffer, load-store queue, and BHT. In the figure, unprotected instructions are fetched into the datapath, and protected data is eventually written back to the register file or data cache. The register file and data cache are highlighted to emphasize that they hold architectural state and that they cannot be recovered using the core's misprediction recovery mechanism. Note that, with minor modifications, our scheme could be adapted for use in microarchitectures with register update units (RUUs) or microarchitectures that keep the results of completed but uncommitted instructions in the physical register file and use explicit register renaming with a map table. Our design treats the combinational logic that manipulates the data that flows through the microprocessor (e.g., instruction decoders, functional units) as black boxes. Protecting this logic from hard faults is an orthogonal issue.

As an instruction progresses through the pipeline, every time it is modified, an EDC write must occur. As Fig. 3 shows, this activity occurs after instruction fetch, instruction decode, ALU operation, memory reads, and before updating the BHT, assuming it is optionally protected. With the exception of the BHT update, these EDC write operations must be on the critical path of the pipeline, and, thus, add additional latency to the instruction's processing time. The use of EDC, as opposed to ECC, is advantageous in that it provides for a lower-latency operation that takes less logic to implement in the timing and space-constrained pipeline. EDC must be checked after any access to a datum contained in a protected structure. However, the only time that this EDC check activity is on the critical path of an instruction's execution is when the instruction's result is to be committed to architectural state

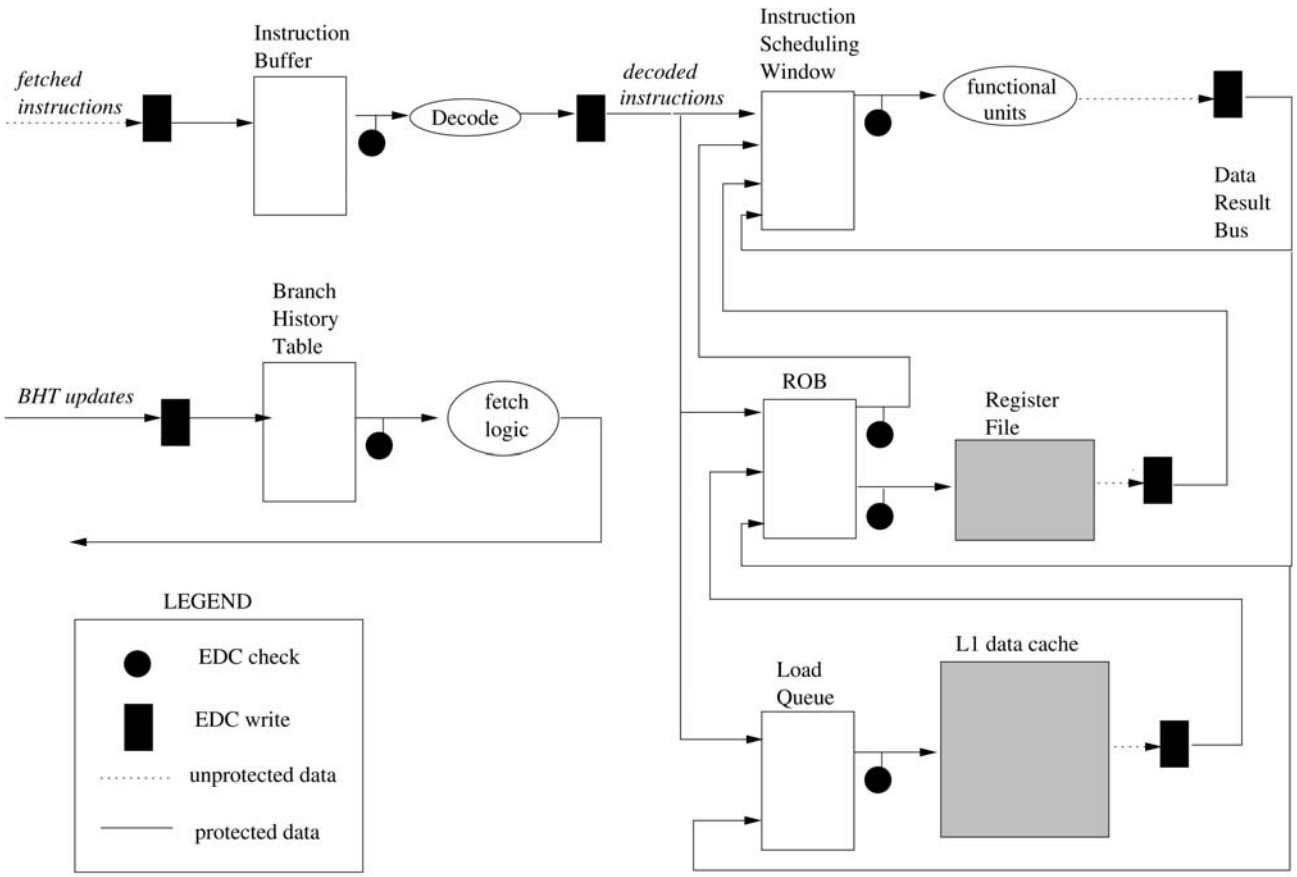

Fig. 3. **Datapath design with SRAS-EDC**. The register file and L1 data cache (L1 D$) are highlighted to emphasize that they hold architectural state. This simplified figure ignores the store queue since stores are handled just like nonload instructions, except that they write their results to the L1 D$ instead of the register file.

at the end of the pipeline. At all other times, the datum can be used by a subsequent pipeline stage without knowing the EDC result, since the later discovery of an error in the EDC check can be contained by flushing the pipeline.

4.4.1 Detection and Diagnosis

SRAS-EDC uses error detecting codes (EDC) to detect and diagnose errors in array rows. There are numerous kinds of EDCs, including parity and cyclic redundancy check (CRC) codes. EDCs add some number of check bits, k, to the original d data bits, and the tradeoff is between the cost due to the number of check bits added and the added error detection capabilities of having more check bits. For example, a single parity bit adds a $1/d$ cost and can detect all single-bit errors. For implementation purposes, we prefer a *separable* EDC, i.e., the check bits are not interleaved with the data bits. Thus, each array row consists of d data bits followed by k check bits. We also want an EDC that can detect all single-bit errors and many types of multiple-bit errors, particularly unidirectional errors (i.e., all 0->1 or 1->0). Many EDC options exist—a designer can choose the EDC that best suits the system, based on the tradeoff between error detection capability and implementation cost. Because of our fault model, we choose Berger codes [5] to protect all arrays except the BHT, since Berger codes can detect all single-bit errors and all unidirectional multiple-bit errors. A Berger code will detect all single stuck-at faults and coupling faults (from one bit to any number of neighboring bits). In a Berger code, the k check bits are the binary representation of the number of zeros in the original data, and, thus, $k = \lceil \log_2(d+1) \rceil$. For the BHT, which has only 2-bit entries, we simply use a parity bit for EDC.

As in SRAS-CR, to distinguish hard faults from soft faults, we add a small counter to each row that is incremented for every error detected in it, and all counters are periodically cleared. If an error increments a counter such that it exceeds a specified threshold, then this row is considered to have a permanent fault; otherwise, the error is considered transient. All data written into arrays is protected with EDC, and all data read from arrays has its EDC checked. We also maintain EDC bits in the register file, in order to not have to recompute EDC for data that is read from the register file to be written into the instruction window. Nevertheless, we are not protecting the register file from hard faults—a hard fault would be detectable but unrecoverable. Future work will explore how to extend SRAS-EDC to protect architectural state.

The only six instances in which EDC logic (writing EDC bits to the end of a datum or checking EDC bits) can potentially impact performance are when:

1. Fetched instructions go through logic that adds EDC bits to them before inserting them into the instruction buffer.

2. Decoded instructions go through logic that adds EDC bits to them before inserting them into the instruction scheduling window. EDC needs to be recomputed here, since the process of decoding the instructions modifies their data payload.
3. Data produced by functional units goes through logic that adds EDC bits before being written into the instruction window (as operands) and the ROB (as results). EDC needs to be recomputed here, since the functional units produce new data. This EDC logic could be associated with the functional units or with the data result bus. An optimization is to compute the EDC (for the outputs of the functional units) in parallel with the outputs. This requires more hardware but hides the latency of the EDC logic, and we will explore the potential of this approach in our evaluation in Section 5. Future work will explore *self-checking circuits* [43], in which EDC codewords (using *arithmetic codes*) are produced by the combinational logic (e.g., functional units). This would enable us to check the functional units themselves (but not to map out hard faults in them), and it would also remove the need for this EDC recomputation logic.
4. Data loaded from the L1 data cache goes through logic that adds EDC bits before being written into the ROB. EDC needs to be recomputed here, since we do not assume that the caches implement the same EDC. If we were to relax this assumption, then this logic for recomputing EDC would no longer be necessary.
5. Data from the ROB goes through logic that checks the EDC before being committed into the register file (or into the L1 data cache, for stores). In Fig. 3 (which omits stores, for clarity), this data is shown as unprotected (before it is checked) despite coming from the protected ROB. This is because, unlike for other structures, the EDC check on this data cannot be done later, and, thus, undo the effects of writing this potentially erroneous data into the register file or data cache.
6. Updates to the branch history table go through logic that adds a parity bit. However, checking the parity bit of data that is read from BHT is off the critical path.

In the first five of these situations, EDC logic is on the critical path, and we pessimistically assume that we must add an extra pipeline stage to accommodate this latency. The exception is adding the parity bit to the BHT—we assume that this simple operation will not force the addition of a pipeline stage. In all other instances, EDC logic is *off the critical path*. For example, when instructions pass from the instruction buffer to the instruction window (after being decoded and renamed), their EDCs are checked off the critical path. That is, erroneous data could be written into the instruction window before the EDC check is complete; however, the EDC check will fail soon thereafter and trigger a system recovery which will eliminate the effects of the error before they can be committed to architectural state. Other EDC checks are between the instruction window and the functional units, between the load queue and the data cache, and between the ROB and the instruction window.

One potential challenge for fast implementation of EDC (or ECC, for that matter) is that partial writes to a structure (i.e., writes that do not modify the entire data) turn into RMW operations. Recall that this limitation is also present for SRAS-CR for any write that will have a check performed. The read is necessary to help compute the EDC over the entire data before writing it. Since RMWs are slower and require extra array bandwidth, we would like to avoid them if possible. Our solution is to compute EDCs over independently written fields of array rows, instead of over the entire row, in order to avoid any possible partial writes. For example, in the ROB, we compute separate EDCs for the result data and for the rest of the entry. Thus, when the entry is allocated, we must compute both, but this is no more complex than computing it over the whole entry. The key savings is when the result is written during instruction completion, since we no longer need to do a RMW.

4.4.2 Recovery

Recovery is implemented with the microprocessor's normal misprediction recovery mechanism. Thus, unlike SRAS-CR, SRAS-EDC does not need DIVA. This recovery mechanism effectively deletes all speculative, uncommitted microprocessor state, but it cannot undo changes made to the architectural state such as the register file. This is why the EDC check between the ROB and register file is on the critical path.

4.4.3 Remapping

We use the same techniques as SRAS-CR for mapping out faulty rows of arrays.

4.4.4 SRAS-EDC Costs

The costs for SRAS-EDC are less than those of SRAS-CR in two important ways: First, SRAS-EDC does not require DIVA. Second, SRAS-EDC does not require all of the extra reads and writes that were necessary for the check rows. However, SRAS-EDC does add some hardware for performing EDC computations. It also adds some performance overhead because of those instances in which EDC logic is on the critical path.

4.4.5 Limitations of SRAS-EDC

There are a few limitations of SRAS-EDC. First, fault coverage is limited by the strength of the chosen EDC. This is a tuneable parameter, in which a designer can tradeoff error detection capability against implementation cost. Second, we can only protect structures that do not hold committed architectural state. Thus, we can protect the ROB, LSQ, IQ, IW, etc., but we cannot protect the register file, processor status word, condition codes, etc. Future work will extend our approach to encompass architected state (e.g., by periodically checkpointing it to memory).

4.5 Applicability of SRAS to Specific Structures

While we have thus far generalized structures as buffer-like or table-like, each structure must be considered in detail to understand how SRAS can be made to work on it. We now present the detailed assumptions about the different structures we studied to provide better intuition in

applying SRAS techniques to a specific design that we have not specifically addressed with this study.

4.5.1 Instruction Buffer

The instruction buffer is just a simple buffer. As the holding place for fetched instructions awaiting decoding, this buffer is a FIFO queue, with each entry written once. There is no requirement to modify its basic structure to accommodate SRAS.

4.5.2 Instruction Scheduling Window

After instructions are decoded, they are stored in this structure until their operands are ready and functional units are available to execute them. Allocation of entries is buffer-like, but the following aspects of the instruction scheduling window are not. First, the structure is also a content-addressable memory (CAM), in order to enable wake-up and select logic to find ready instructions as well as to allow operand readiness to be properly updated each cycle. Second, this structure is the beginning of the out-of-order execution of the microprocessor core. Instructions are removed as they become ready, not in FIFO order. Typical implementations perform a compaction of the structure at the end of each cycle to keep the oldest instructions near the head of the queue and to simplify allocation of entries in the next decode cycle. Finally, each entry will typically be updated between allocation and its eventual use and retirement.

For SRAS-CR, these factors are mitigated by performing a full write of the entry at its point of allocation and performing the check at that time. As mentioned in the discussion of SRAS-CR, pseudorandom data from the tick counter is used to populate the uninitialized fields of this structure to allow for the check to be calculated properly without requiring partial updates to be converted to RMWs and to have a fixed upper bound on the number of check circuits required to perform checks (for n-wide decode circuitry, we need n check circuits). Subsequent overwriting of partial data and movement during compaction is effectively ignored by SRAS-CR. This is tolerable since DIVA will correct any errors introduced by moving a good datum to an array entry that has a hard fault present that has yet to be deconfigured.

For SRAS-EDC, EDC must be recalculated for every update to the structure. We can avoid RMWs by dividing the EDC into separate EDC fields for each of the written subpieces of the entry. This also provides the advantage of only requiring $2n$ EDC calculation circuits, since at most n operands will become ready in a given cycle in an n-wide processor. The calculated EDC for a particular operand becoming ready is independent of the rest of the instruction window data. This allows a single EDC calculation to be written multiple times at all applicable locations in the instruction window. So, in a given cycle, n EDC calculations will be required for the incoming n decoded instructions and the newly computed n ready operands (making for a total of $2n$). Compaction is not a problem, since the EDC travels with the entry and remains valid during the compaction. As with SRAS-CR, the number of EDC checking circuits required for the structure is equivalent to the issue width of the processor.

4.5.3 Load-Store Queue

Like the instruction window, the LSQ is a buffer that also can be accessed as a CAM. This feature does not adversely impact either of the SRAS schemes. SRAS-CR again uses pseudorandom data if necessary to perform the check at the time of entry allocation. SRAS-EDC must maintain $2n$ EDC calculation circuitry sets (one for the initial write of the entry and one for address calculation arrival from an ALU) in order to allow for a peak sustained memory bandwidth of n instructions per cycle on an n-wide processor. Only n copies of the EDC check circuit are required. As with the instruction window, the subfields of an entry may have their EDC calculated separately to simplify the EDC calculation circuitry's implementation.

4.5.4 Branch History Table

The BHT is a table with addressable content on a very small granularity. The descriptions of SRAS operation for table structures were crafted with the BHT as a motivating example. For other tabular structures, the aforementioned techniques of writing pseudorandom data (for SRAS-CR) or splitting the table entry into separate EDC fields (for SRAS-EDC) may be applicable.

4.5.5 Reorder Buffer

The ROB is a FIFO queue with the potential of multiple partial writes during the lifetime of an instruction. Issues here are similar to those found in the instruction scheduling window. The same techniques would apply here for the two SRAS methods.

5 Evaluation

In this section, we evaluate the benefits and costs of adding self-repair to microprocessor arrays. Our goal is to determine whether self-repair is viable, primarily in terms of performance, as performance is of critical importance in the commodity processor design space. We will compare both SRAS-CR and SRAS-EDC to systems protected with DIVA as well as to each other. Comparing SRAS to DIVA is somewhat unfair, since DIVA was not designed to handle hard faults, but it is the best alternative currently available. An important question we seek to answer is whether SRAS-EDC can achieve comparable performance to SRAS-CR despite not requiring DIVA support or the other drawbacks of SRAS-CR.

5.1 Methodology and System Model

We use the SimpleScalar toolset [2] to evaluate our design. We model a dynamically scheduled microprocessor that is similar to currently available microprocessors, such as the Intel Pentium4 [14] and Alpha 21364 [13]. The details of the target system are shown in Table 2. We protect the instruction buffer, instruction scheduling window, reorder buffer, and load-store queue. The percentage of the microprocessor core that this protects depends on implementation. Specific details are proprietary, but estimation can be done with annotated die photos of a representative chip. Such analysis of the Alpha 21264 [34] shows that these array structures comprise roughly 33 percent of the noncache microprocessor core die area.

TABLE 2
Target System Parameters

pipeline depth	22
pipeline width	3
instruction fetch buffer	64
scheduling window	32
load-store queue	48
reorder buffer	126
functional units	4 integer adders and multiplier, 1 FP adder, 1 FP multiplier
branch predictor	gshare: BHT is 4096 entries, BHT entry is 2-bit counter, BHR is 8 bits
registers	192
L1 D-cache	8K total size, 4-way, 2-cycle
L1 I-cache	8K total size, 4-way, 2-cycle
L2 cache	256K size, 8-way, 7-cycle

We simulate the SPEC2000 CPU benchmarks, and we use the SimPoint toolset [33] to choose statistically representative samples of these long benchmarks for detailed simulation. We inject varying numbers of both stuck-at errors and coupling errors into the protected structures. Due to fault masking, injected hard faults do not always lead to errors when the faulty structures are accessed. For example, a stuck-at-one fault does not effect a bit that is dynamically set to one during execution.

5.2 Results

In this section, we present the results of our evaluation.

5.2.1 Fault-Free Performance

Our first experiment explores the performance impact of SRAS for a system with no faults injected. The goal of this experiment is to determine the fault-free performance overhead of our schemes relative to a system with DIVA. In Fig. 4, we plot the fault-free runtimes (taller bars correspond to worse performance) of several systems, normalized to the baseline case of a system with DIVA (or an unprotected system), for all of the SPEC integer and floating point benchmarks. For each benchmark, we plot:

1. the baseline,
2. SRAS-CR (protecting just the ROB[1]),
3. unoptimized SRAS-EDC,
4. SRAS-EDC with a partial optimization in which we compute functional unit EDC in parallel for addition and subtraction, since these are the most common and the easiest to perform in parallel (i.e., they require the least extra hardware for parallel EDC computation), and
5. SRAS-EDC with the full optimization described in Section 4.4 to compute all functional unit EDCs in parallel (or not compute them at all, for self-checking circuits).

The results show that SRAS-CR has the same performance as the baseline and that SRAS-EDC unsurprisingly incurs some penalty with respect to DIVA, due to adding some EDC logic on the critical path. The full optimization for SRAS-EDC helps quite a bit on most benchmarks, and the partial optimization does almost as well. One trend is that SRAS-EDC tends to suffer worse degradation in performance on the integer benchmarks, explained by the fact that the extra pipeline stages in SRAS-EDC exacerbate the branch misprediction penalty which is incurred more frequently by the integer benchmarks.

1. Results discussed later will show that protecting the BHT is not worthwhile, and, thus, we do not wish to incur its fault-free performance penalty in this experiment.

5.2.2 Performance in Presence of Faults

In this experiment, we study the performance benefit of self-repair for a system in which hard faults have been injected. Our goal is to determine whether self-repair provides enough benefit in the presence of hard faults to be worth its costs (in terms of implementation and fault-free performance) In Fig. 5, we plot the runtime of SRAS-EDC versus that of a system protected by DIVA, in the presence of hard faults injected into the reorder buffer. For clarity, we do not plot SRAS-CR results since they are very similar to SRAS-EDC. We inject one, four, and eight stuck-at-1 hard faults in order to evaluate the relative impact of varying numbers of hard faults. We normalize the results to the case of DIVA with no faults injected. Here, we see that, in general, the presence of hard faults leads to SRAS-EDC outperforming DIVA. For the few integer benchmarks for which SRAS-EDC incurs the greatest fault-free performance degradation, however, DIVA may still have a slight advantage in the case of only one hard fault, but SRAS-EDC always outperforms DIVA for four and eight faults. Considering that defect and fault rates are increasing, and we cannot eliminate all of them with burn-in testing [28], [4], these results demonstrate that SRAS is worthwhile. We observe that the floating point benchmarks derive relatively more benefit from self-repair. This effect is due to these benchmarks tending to better utilize the pipeline and thus incur more of a loss when an error causes DIVA to have to flush the pipeline.

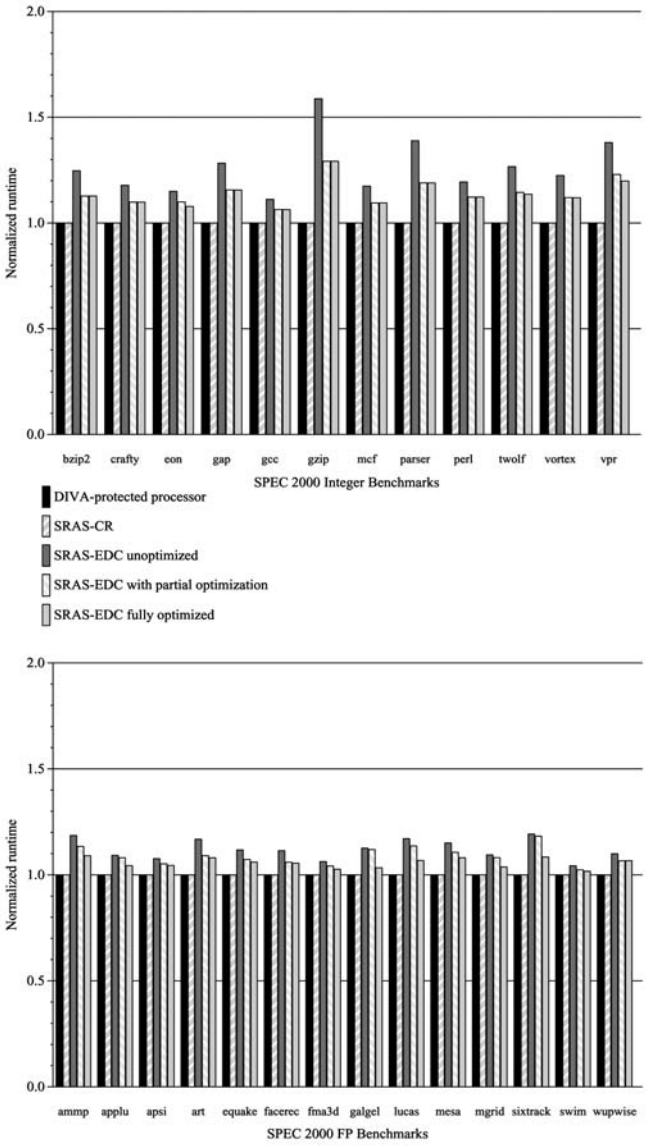

Fig. 4. Fault-free runtime.

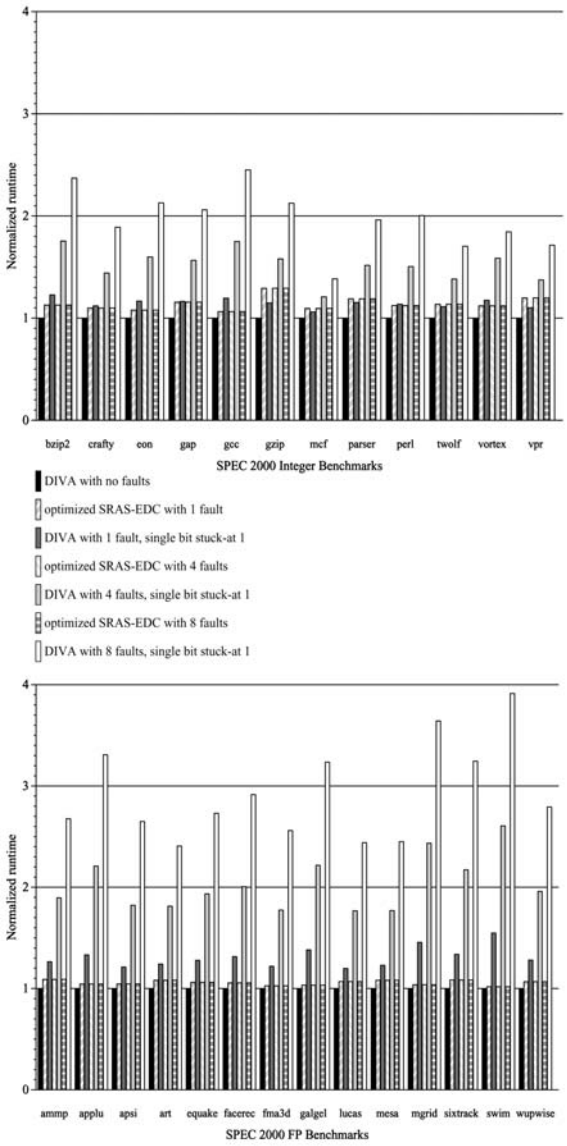

Fig. 5. Runtime with hard faults injected into the reorder buffer.

5.2.3 Relative Performance Impact of Protecting Different Arrays

In this experiment, we explore the impact of hard faults on the other array structures that we are protecting with self-repair. Having shown in the previous experiment that ROB self-repair is beneficial in the presence of hard faults, we now compare the relative benefits of self-repair for other arrays. For each of the five structures we are protecting with self-repair—ROB, load-store queue, instruction window (scheduling window), instruction buffer (fetch buffer), and branch history table (BHT)—we injected a single stuck-at fault in that structure (i.e., we created five systems, each with a single fault in a different array). We then simulated each system's performance on a system with DIVA (i.e., without self-repair), to gauge the performance degradation that DIVA would incur (and that a system with self-repair would not incur). In Fig. 6, we plot the runtimes for these five systems, normalized to a fault-free system. Thus, a taller bar in the graph indicates that self-repair is more important for this structure. We observe that there is no one particular structure that is always the most important to protect with self-repair, although there are some patterns. For example, the instruction window benefits more from self-repair than the ROB, as does the instruction buffer for most benchmarks. A significant result is that faults in the BHT have virtually no impact on performance. This is because the BHT is a large structure that is accessed sparsely, and faults in the BHT are likely to be masked. Moreover, faults in the BHT can only lead to incorrect branch predictions, not incorrect execution, so the corresponding pipeline squashes can be initiated earlier (after the execution stage, instead of at the commit stage) and, thus, incur less performance penalty.

6 Conclusions

In this paper, we have presented a framework for designs for self-repair of microprocessor array structures, and we have developed two particular designs based on that

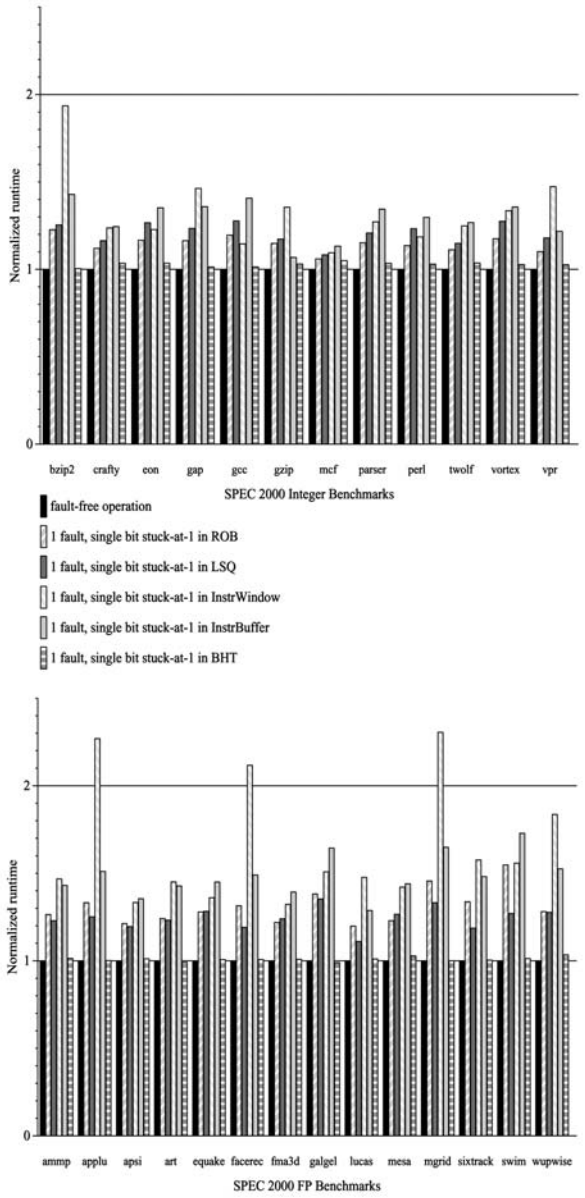

Fig. 6. Impact on runtime of hard faults on other array structures.

framework: SRAS-CR and SRAS-EDC. These designs are motivated by the belief that per-part hard fault rates will increase as we scale CMOS to smaller and smaller device geometries and pack ever more devices into a single microprocessor. This motivation is grounded in cautionary statements from the ITRS [16] and detailed studies of lifetime reliability by Srinivasan et al. [36].

A survey of methods for achieving hard-fault tolerance in the microprocessor core shows that we have a gap in capability for protecting the noncache area. This gap stems from the following two facts:

1. Traditional hard-fault tolerance design points could afford large-scale redundancy, for example replication of the entire core, so they used techniques like TMR to achieve fault tolerance.
2. Newly developed low-cost fault-tolerance techniques are not designed to tolerate hard faults—even though they sometimes can, this tolerance comes at a high performance cost.

In the commodity microprocessor market, performance and cost are the key motivating constraints. As CMOS trends begin to impact this design space, we believe that fault tolerance will gain in importance. Low-cost methods to achieve hard fault tolerance will become necessary as a result. The two SRAS methods presented are two such designs.

In the case of fault-free execution, both SRAS methods may add some performance overhead compared to an unprotected system, due to the few instances in which self-repair logic is on the critical path. However, if hard faults exist in arrays, then SRAS outperforms the existing light-weight approaches for tolerating faults while avoiding large-scale replication of microprocessor cores. As hard fault rates continue to increase, we believe that SRAS will become an increasingly attractive design point.

Acknowledgments

This material is based on work supported by the US National Science Foundation under grants CCR-0309164, CCF-0444516, and EIA-9972879, the National Aeronautics and Space Administration under grant NNG04GQ06G, Intel Corporation, IBM, and a Duke Warren Faculty Scholarship. The authors thank Alvy Lebeck, Tong Li, Pete Marinos, and Ismet Bayraktaroglu for their insightful comments and criticisms of this work. This paper partially includes research that appeared in the Proceedings of the 2004 International Conference on Dependable Systems and Networks [8].

References

[1] M. Abramovici, M.A. Breuer, and A.D. Friedman, *Digital Systems Testing and Testable Design.* IEEE Press, 1990.
[2] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An Infrastructure for Computer System Modeling," *Computer,* vol. 35, no. 2, pp. 59-67, Feb. 2002.
[3] T.M. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," *Proc. 32nd Ann. IEEE/ACM Int'l Symp. Microarchitecture,* pp. 196-207, Nov. 1999.
[4] T.S. Barnett, A.D. Singh, and V.P. Nelson, "Extending Integrated-Circuit Yield-Models to Estimate Early-Life Reliability," *IEEE Trans. Reliability,* vol. 52, no. 3, pp. 296-300, Sept. 2003.
[5] J.M. Berger, "A Note on Error Detecting Codes for Asymmetric Channels," *Information and Control,* vol. 4, pp. 68-73, Mar. 1961.
[6] D.T. Blaauw, C. Oh, V. Zolotov, and A. Dasgupta, "Static Electromigration Analysis for On-Chip Signal Interconnects," *IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems,* vol. 22, no. 1, pp. 39-48, Jan. 2003.
[7] R. Blish et al. "Critical Reliability Challenges for the International Technology Roadmap for Semiconductors (ITRS)," Technical Report 03024377A-TR, Int'l SEMATECH, Mar. 2003.
[8] F.A. Bower, P.G. Shealy, S. Ozev, and D.J. Sorin, "Tolerating Hard Faults in Microprocessor Array Structures," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 51-60, June 2004.
[9] T. Chen and G. Sunada, "A Self-Testing and Self-Repairing Structure for Ultra-Large Capacity Memories," *Proc. Int'l Test Conf.,* pp. 623-631, Oct. 1992.
[10] T. Chen and G. Sunada, "An Ultra-Large Capacity Single-Chip Memory Architecture with Self-Testing and Self-Repairing," *Proc. Int'l Conf. Computer Design (ICCD),* pp. 576-581, Oct. 1992.
[11] T.J. Dell, "A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory," IBM Microelectronics Division Whitepaper, Nov. 1997.
[12] D.J. Dumin, *Oxide Reliability: A Summary of Silicon Oxide Wearout, Breakdown, and Reliability.* World Scientific Publications, 2002.
[13] L. Gwennap, "Alpha 21364 to Ease Memory Bottleneck," *Microprocessor Report,* Oct. 1998.

[14] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of the Pentium 4 Processor," *Intel Technology J.,* Feb. 2001.
[15] IBM, Enhancing IBM Netfinity Server Reliability: IBM Chipkill Memory, IBM Whitepaper, Feb. 1999.
[16] Int'l Technology Roadmap for Semiconductors, 2003.
[17] JEDEC Solid, State Technology Assoc., "Failure Mechanisms and Models for Semiconductor Devices,""JEDEC Publication JEP122-B, Aug. 2003.
[18] D. Jewett, "Integrity S2: A Fault-Tolerant UNIX Platform," *Proc. 21st Int'l Symp. Fault-Tolerant Computing Systems,* pp. 512-519, June 1991.
[19] S. Krumbein, "Metallic Electromigration Phenomena," *IEEE Trans. Components, Hybrids, and Manufacturing Technology*, vol. 11, no. 1, pp. 5-15, Mar. 1988.
[20] B.P. Linder, J.H. Stathis, D.J. Frank, S. Lombardo, and A. Vayshenker, "Growth and Scaling of Oxide Conduction After Breakdown," *Proc. 41st Ann. IEEE Int'l Reliability Physics Symp.,* pp. 402-405, Mar. 2003.
[21] P. Mazumder and J.S. Yih, "A Novel Built-In Self-Repair Approach to VLSI Memory Yield Enhancement," *Proc. Int'l Test Conf.,* pp. 833-841, 1990.
[22] S. McFarling, "Combining Branch Predictors," Technical Report TN-36, Digital Western Research Laboratory, June 1993.
[23] S.S. Mukherjee, C. Weaver, J. Emer, S.K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," *Proc. 36th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* Dec. 2003.
[24] M. Nicolaidis, N. Achouri, and S. Boutobza, "Dynamic Data-Bit Memory Built-In Self-Repair," *Proc. Int'l Conf. Computer Aided Design,* pp. 588-594, Nov. 2003.
[25] K. Nikolic, A. Sadek, and M. Forshaw, "Fault-Tolerant Techniques for Nanocomputers," *Nanotechnology,* vol. 13, pp. 357-362, 2002.
[26] I. Pomeranz and S.M. Reddy, "On n-Detection Test Sets and Variable n-Detection Test Sets for Transition Faults," *Proc. 17th IEEE VLSI Test Symp.,* pp. 173-180, Apr. 1999.
[27] S.K. Reinhardt and S.S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," *Proc. 27th Ann. Int'l Symp. Computer Architecture,* pp. 25-36, June 2000.
[28] W.C. Riordan, R. Miller, J.M. Sherman, and J. Hicks, "Microprocessor Reliability Performance as a Function of Die Location for a 0. 25um, Five Layer Metal CMOS Logic Process," *Proc. 37th Ann. IEEE Int'l Reliability Physics Symp.,* pp. 1-11, Mar. 1999.
[29] R. Rodriguez, R.V. Joshi, J.H. Stathis, and C.T. Chuang, "Oxide Breakdown Model and Its Impact on SRAM Cell Functionality," *Simulation of Semiconductor Processes and Devices (SISPAD),* pp. 283-286, Sept. 2003.
[30] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," *Proc. 29th Int'l Symp. Fault-Tolerant Computing Systems,* pp. 84-91, June 1999.
[31] K. Sawada, T. Sakurai, Y. Uchino, and K. Yamada, "Built-in Self Repair Circuit for High Density ASMIC," *Proc. IEEE Custom Integrated Circuits Conf.,* 1989.
[32] J. Saxena et al., "Scan-Based Transition Fault Testing—Implementation and Low Cost Test Challenges," *Proc. Int'l Test Conf.,* pp. 1120-1129, Oct. 2002.
[33] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," *Proc. 10th Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* Oct. 2002.
[34] P. Shivakumar, S.W. Keckler, C.R. Moore, and D. Burger, "Exploiting Microarchitectural Redundancy For Defect Tolerance," *Proc. 21st Int'l Conf. Computer Design,* Oct. 2003.
[35] L. Spainhower and T.A. Gregg, "IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective," *IBM J. Research and Development,* vol. 43, nos. 5/6, Sept./Nov. 1999.
[36] J. Srinivasan, S.V. Adve, P. Bose, and J.A. Rivers, "The Impact of Technology Scaling on Lifetime Reliability," *Proc. Int'l Conf. Dependable Systems and Networks,* June 2004.
[37] J.R. Srour, D. Long, D. Millward, R.L. Fitzwilson, and W.L. Chadsey, *Radiation Effects on and Dose Enhancement of Electronic Materials.* Noyes Publications, 1984.
[38] K. Sundaramoorthy, Z. Purser, and E. Rotenberg, "Slipstream Processors: Improving Both Performance and Fault Tolerance," *Proc. Ninth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* pp. 257-268, Nov. 2000.
[39] J. Tao, J.F. Chen, N.W. Cheung, and C. Hu, "Modeling and Characterization of Electromigration Failures Under Bidirectional Current Stress," *IEEE Trans. Electron Devices,* vol. 43, no. 5, pp. 800-808, May 1996.
[40] S. Thompson et al., "An Enhanced 130nm Generation Logic Technology Featuring 60nm Transistors for High Performance and Low Power at 0.7-1.4V," *Proc. Int'l Electron Devices Meeting,* pp. 257-260, Dec. 2001.
[41] C.-W. Tseng and E.J. McCluskey, "Multiple-Output Propagation Transition Fault Test," *Proc. Int'l Test Conf.,* pp. 358-366, Nov. 2001.
[42] T.N. Vijaykumar, I. Pomeranz, and K.K. Chung, "Transient Fault Recovery Using Simultaneous Multithreading," *Proc. 29th Ann. Int'l Symp. Computer Architecture,* pp. 87-98, May 2002.
[43] J.F. Wakerly, *Error Detecting Codes, Self-Checking Circuits and Applications.* North-Holland, 1978.
[44] C. Weaver and T. Austin, "A Fault Tolerant Approach to Microprocessor Design," *Proc. Int'l Conf. Dependable Systems and Networks,* pp. 411-420, July 2001.
[45] D. Weiss, J.J. Wuu, and V. Chin, "The On-Chip 3MB Subarray Based 3rd Level Cache on an Itanium Microprocessor," *Proc. Int'l Solid-State Circuits Conf.,* pp. 112-113, Feb. 2002.
[46] D. Wilson, "The Stratus Computer System," *Resilient Computer Systems,* pp. 208-231, 1985.
[47] T.-Y. Yeh and Y. Patt, "Two-Level Adaptive Training Branch Prediction," *Proc. 24th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* pp. 51-61, Nov. 1991.
[48] L. Youngs and S. Paramanandam, "Mapping and Repairing Embedded-Memory Defects," *IEEE Design & Test of Computers,* pp. 18-24, Jan.-Mar. 1997.

Fred A. Bower received the BS degree in mechanical engineering and computer science from Oregon State University in 1996 and the MS degree in computer science and engineering from The Oregon Graduate Institute of Science and Technology in 1999. He is a PhD candidate in the Department of Computer Science at Duke University and is also the technical lead for high volume Intel servers in IBM's Systems and Technology Group. His present research focus is on developing low-cost hard-fault tolerance solutions for the microprocessor core. His other research interests include fault tolerance from a holistic system-wide perspective, including hardware, firmware, operating systems, and application software. He is a student member of the IEEE.

Sule Ozev received the BS degree in electrical engineering from Bogazici University, Turkey, and the MS and PhD degrees in computer science and engineering from the University of California, San Diego, in 1995, 1998, and 2002, respectively. Dr. Ozev is now an assistant professor in the Electrical and Computer Engineering Department at Duke University. Dr. Ozev is the program chair for IEEE North Atlantic Test Workshop (2005-2006), and serves on several program committees. Her research interests include testing of mixed-signal and RF circuits, process variability analysis for analog circuits, and fault-tolerant processor designs. She is a member of the IEEE and the IEEE Computer Society

Daniel J. Sorin received the PhD degree in electrical and computer engineering from the University of Wisconsin-Madison. He is an assistant professor of electrical and computer engineering and of computer science at Duke University. His research interests are in highly available computer architectures and architectures for emerging nanotechnologies. He is a member of the IEEE and the IEEE Computer Society.