

Tolerating Hard Faults in Microprocessor Array Structures

Fred A. Bower, Paul G. Shealy, Sule Ozev, and Daniel J. Sorin
Department of Electrical and Computer Engineering
Duke University

Abstract

In this paper, we present a hardware technique, called Self-Repairing Array Structures (SRAS), for masking hard faults in microprocessor array structures, such as the reorder buffer and branch history table. SRAS masks errors that could otherwise lead to slow system recoveries. To detect row errors, every write to a row is mirrored to a dedicated “check row.” We then read out both the written row and check row and compare their results. To correct errors, SRAS maps out faulty array rows with a level of indirection.

1 Introduction

As microprocessor fabrication technology continues to shrink devices and wires and increase clock frequencies, hard fault rates are consequently increasing. One reason is the increased probability of short and open circuits. As circuit dimensions continue to shrink, hard fault rates will increase [4, 17], as effects such as electromigration and gate dielectric breakdown become more likely. Moreover, with increasing numbers of transistors being used, the probabilities of microprocessor hard faults are correspondingly increasing.

Existing solutions for hard faults, which we discuss in more detail in Section 2, are either very expensive or suffer performance penalties for many classes of hard faults. One class of approaches uses redundant parallel processors (e.g. pair and spare) to provide forward error recovery (FER). These systems provide high availability, but they use a large amount of hardware and are thus expensive. At the other end of the high availability design spectrum, a recently developed scheme, called DIVA [3], uses only a small on-chip checker to achieve almost as much availability as redundant processor schemes. DIVA uses much less hardware than redundant processors, but it incurs a significant performance penalty for recovery every time a fault is exercised. This recovery penalty can be particularly problematic for hard faults in heavily used circuits, such as the reorder buffer or instruction queue. Ideally, we would like to

enhance DIVA to mask hard faults so that they do not lead to frequent, slow recoveries.

In this paper, we develop a lightweight hardware technique, called *Self-Repairing Array Structures (SRAS)*, that enables a microprocessor with DIVA to tolerate a broad class of hard faults without incurring frequent performance-degrading recoveries. SRAS masks hard faults in array structures within the microprocessor, so that DIVA recovery does not have to be invoked. Microprocessors contain many large array structures—including both buffers and tables, such as the reorder buffer (ROB) and the branch history table (BHT)—and our goal is to tolerate hard faults in rows of these structures. To detect and diagnose row errors, every write to a row is mirrored to a dedicated “check row.” We can then read out both the written row and the check row and compare their results to detect errors. To dynamically repair hard faults in rows, we extend a technique used in the context of disks and memories, in which faulty portions of arrays can be mapped out by using a level of indirection. While our high-level design logically uses a level of indirection, which could degrade processor performance, we present a detailed implementation that can optimize certain critical paths. In particular, for buffers that are not randomly addressable, we present an implementation that incurs no performance penalty.

To evaluate our idea, we simulate a microprocessor with our hard fault tolerance mechanisms, and we compare its performance to unmodified DIVA (i.e., without SRAS). We inject several representative types of faults into the simulated microprocessor. Results show that adding SRAS to DIVA enables a microprocessor to achieve performance close to the fault-free scenario despite the injection of hard faults into certain array structures.

2 Background

In this section, we provide a brief background in existing techniques for tolerating hard faults in microprocessors, before delving into the details of our fault model.

2.1 Existing Approaches

There are several existing techniques for comprehensively tolerating hard faults in microprocessor cores. The most obvious approach is forward error recovery (FER) via the use of redundant microprocessors in parallel, e.g., triple modular redundancy (TMR). For extreme reliability, this is an effective but not cost-efficient solution. IBM mainframes [22], Tandem S2 [11], and Stratus [24] are examples of systems that use redundant processors to mask hard faults. Mainframes also replicate certain structures within the processors to increase reliability [22]. The drawback of these schemes is the large added hardware expense and power usage of the redundant hardware. For non-mission-critical applications, this solution is not preferred.

Cost-effective approaches for comprehensively tolerating hard faults can be far less expensive, but they often sacrifice performance in the presence of hard faults. DIVA [3] protects a fast, aggressive processor—from both hard and soft faults—with a small, simple, on-chip checker processor. The checker processor is simple enough that the designers could formally prove that its design is correct. The checker processor sits at the commit stage of the aggressive processor and compares the results of its execution of each instruction to the result of execution on the aggressive processor. If the results differ, the checker assumes that it is correct and uses its result. This assumption is based on the provably correct design of the checker and its relatively small size with respect to the aggressive processor. To prevent the fault in the aggressive processor from propagating to later instructions, DIVA then flushes the aggressive processor’s pipeline. In the fault-free scenario, the performance of the system is virtually equal to that of the fast aggressive processor, since the simple checker can leverage the faster processor as a pre-fetch engine. DIVA’s small amount of redundancy is far less expensive and power hungry than TMR, but it has a performance penalty for each error it detects. Every time a hard fault manifests itself as an error, the performance of the system temporarily degenerates to that of the checker processor until the aggressive processor refills its pipeline, since the aggressive processor cannot help it. The checker processor is very slow—the DIVA paper reports that performance will degrade appreciably for error rates greater than one per thousand instructions. In the presence of hard faults that could get exercised frequently, performance will suffer.

Cost-effective approaches for tolerating only specific classes of hard faults also exist. One approach is the use of error correcting codes (ECC). ECC can tolerate up to a targeted number of faulty bits in a piece of

data, and it is a useful technique for protecting SRAM, DRAM, buses, etc., from this fault model. However, ECC cannot tolerate more than a certain number of faulty bits, nor can it be implemented quickly enough to be a viable solution for many performance-critical structures in a microprocessor. More general approaches for tolerating hard faults in memory storage are discussed in Section 3, since they are similar to SRAS.

2.2 Hard Fault Model

Several structural fault models have been developed for logic circuits and storage components over the past few decades [1]. The stuck-at fault model is the most commonly used model in VLSI testing and fault tolerance schemes. In the stuck-at fault model, a physical defect manifests itself as a signal consistently having a certain value (either zero or one) independent of the input. The coupling fault model has been recently defined for storage components [5]. For coupling faults, a write to a certain memory location always prompts a write to a neighboring location or locations.

In SRAS, we use check rows to determine whether data that is written into a row can be read out correctly. In this sense, the fault detection and repair scheme of SRAS is independent of the underlying physical fault model. A fault is detected as soon as it is excited. However, in order to study the impact of SRAS on overall operation, we need to inject physical faults. We inject single-bit and all-bits stuck-at faults within a given row of an array. The all-bits stuck-at- x fault is equal to the all-neighbors coupling fault when the write variable is x . While single-bit stuck-at faults could be tolerated with ECC, albeit with a likely performance penalty, all-bits stuck-at faults require a different approach.

3 High-Level View of SRAS

Technology and microprocessor architecture trends are leading towards 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 replicate these structures. Instead, we combine DIVA’s cost-efficient fault tolerance with a small amount of hardware that detects, diagnoses, and masks hard faults in these array structures. SRAS ensures that the performance of DIVA does not suffer in the presence of hard faults in frequently accessed array structures.

We seek to protect these array structures in a fashion similar to the way in which existing on-line

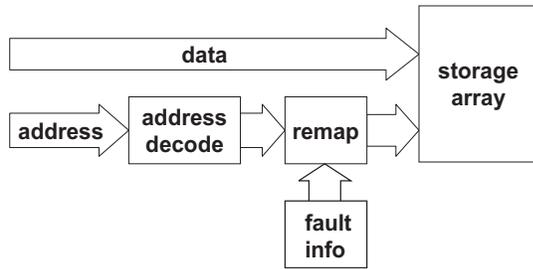


FIGURE 1. Array Remapping

(dynamic) 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. Disk sizes, for example, long ago reached the point at which hard faults were expected and had to be tolerated. Whole disk failures were addressed by RAID [18]. For disk faults that did not incapacitate the entire disk, the solution was to map out faulty portions of them at the sector granularity. Thus, a faulty disk could continue to operate correctly, albeit at a smaller effective size. Similar approaches have been developed for DRAM main memory. Whole chip failures are tolerated by chipkill memory and RAID-M [7, 10], and partial failures are tolerated with schemes that map out faulty locations [6, 13, 19]. For SRAM caches, techniques have been developed to map out defective locations during fabrication [26] and, more recently, during execution [16]. 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.

3.1 Mapping Out Faulty Rows

We logically add a level of indirection that can map out faulty rows in microprocessor array structures, as shown in Figure 1. The remapper serves as the interface between the array and the rest of the microprocessor. There are numerous implementation issues to address in this design, including how to add the remapper into the pipeline, and we discuss them in Section 4.

3.2 Detecting and Diagnosing Faulty Rows

While DIVA can detect errors in processor execution due to faulty rows, it cannot isolate the row or even the structure that is faulty. DIVA only checks end-to-end correctness, which is sufficient for detection but not diagnosis. Thus, in conjunction with remapping to tolerate detected faults, SRAS incorporates a simple scheme for detecting row errors and diagnosing which row is faulty. SRAS 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. Every time an entry is written to the array structure, the same data is also written into a check row. 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. 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 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.

3.3 SRAS Operation

If an error is detected, but the hard fault threshold has not yet been reached, then the fault is considered to be transient and it is tolerated by DIVA with its associated performance penalty. If the detected error raises the counter to the hard fault threshold, then DIVA also tolerates this fault, but the system then repairs itself so as to prevent this hard fault from being exercised again. The repair actions taken depend on whether the faulty row is a non-check row or a check row. If it is a non-check 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 maps in a spare check row.

4 SRAS Implementation

In Section 3, we described SRAS at a high level. In this section, we delve into the implementation issues. We develop several implementation variations of different aspects of SRAS, and we discuss the various pros and cons. Tolerating the faults and detecting/diagnosing them are mostly independent issues, from an implementation standpoint, so we split our discussion into these two topics (Section 4.1 and Section 4.2, respectively). In Section 4.3, we discuss the costs of SRAS, and in Section 4.4, we discuss the limitations of this implementation.

We can classify array structures within the microprocessor core into two categories: non-addressable 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 henceforth focus the discussion 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 in this paper apply broadly.

Reorder Buffer. The ROB is a circular buffer that is used in dynamically scheduled (a.k.a. “out-of-order”) processors to implement precise exceptions by ensuring that instructions are committed in program order. There is an entry in the ROB for each in-flight instruction, and there are pointers to the head and tail entries in the ROB. An entry is added to the tail of the ROB once it has been decoded and is ready to be scheduled. An entry is removed from the head of the ROB when it is ready to be committed. We focus on processors that perform explicit register renaming with a map table, such as the Pentium4 [9] and the Alpha 21364 [8], in which an ROB entry contains the physical register tags for the destination register and the register that can be freed when this instruction commits, plus some other status bits.

ROB sizes are on the order of 32-128 entries, which is large enough to have a non-negligible probability of a hard fault. The ROB is a buffer which cannot be randomly addressed, and we leverage this constraint in our remapper implementation. The ROB has a high architectural vulnerability factor [15], 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).

Branch History Table. The BHT is a table that is accessed during branch prediction. Common two-level branch predictor designs [25] 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). A typical BHT entry is a 2-bit saturating counter [21] that is incremented (decremented) when the corresponding branch is taken (not taken). A BHT value of 00 or 01 (10 or 11) is interpreted as a not-taken (taken) prediction.

BHRs and/or BHTs can be either local (one per branch PC), global (shared across all branch PCs), or shared (by sets of branch PCs). In this paper, we focus on the gshare two-level predictor [14], in which the BHT is indexed by the exclusive-OR of the branch PC

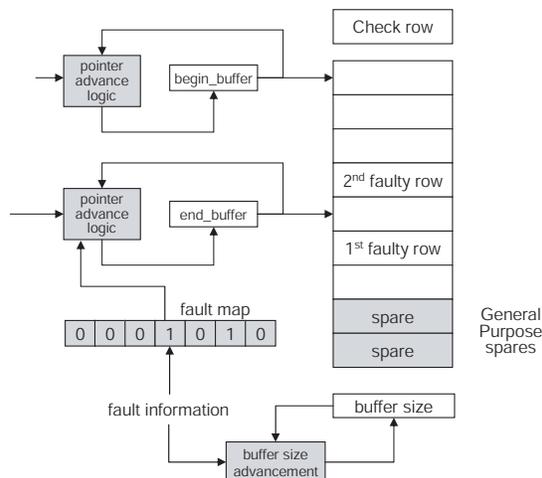


FIGURE 2. Self-Repair for Buffers

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. Thus, DIVA will never detect faults in it. However, a BHT fault can lead to incorrect branch predictions, which can degrade performance.

4.1 Tolerating Detected Faults

While remapping with a level of indirection is straightforward in the abstract, implementing it in a high performance microprocessor pipeline requires careful consideration. We now present remapper implementations for the ROB and BHT.

ROB Remapper. In buffer structures, as in the case of the ROB, 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 location of the active rows. When a new is added, 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. Thus, the faulty row can easily be mapped out by modifying the pointer advancement logic when a hard fault is detected. Figure 2 illustrates the implementation of the self-repair mechanism for buffers, with SRAS hardware highlighted in gray. SRAS uses a *fault map* bit-array to track faulty rows. If a row is determined to contain a hard fault, the corresponding bit in the fault map is modified. The fault map is used by the pointer advancement circuit to determine how far the pointer needs to be advanced. Once the pointer is updated accordingly, reads and

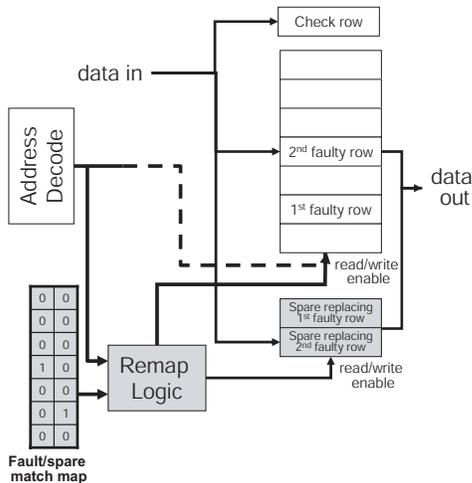


FIGURE 3. Self-Repair for Tables

writes of the buffer entries proceed unmodified. Since the pre-processing for pointer advancement can be done off the critical path, the proposed modification does not impact the read or write access time.

In order to avoid a reduction in the effective buffer size 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 size by one (by adding the spare) while maintaining the same *effective* size. SRAS can tolerate as many hard faults as there are spares without any degradation of buffer performance. If the number of faulty rows exceed the number of spare rows, then the effective buffer size 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 pre-processing, updating fault map entries, and updating the buffer size. Section 4.3 discusses the overall overhead of the complete SRAS 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 in tables, 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 on-line repair mechanisms have been proposed [6, 13]. 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, we distribute spare rows over sub-arrays

of the table, and a spare can only replace a row within its own sub-array. 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 Figure 3. Once again, hardware for SRAS is shown in gray.

Similar to the buffer case, we keep the fault map information in a table. However, we use an extended fault map which also stores the faulty-row/spare matching information. If a row is identified faulty and an unused spare is found to replace it, the corresponding entry of the fault/spare match map is set to 1. 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 (nxk) 2-input AND gates and k n -input OR gates, where n is the size of the subarray and k is the number of spares assigned to that subarray. Once a read/write signal is initiated by the address decode logic, this signal is “AND”ed with the corresponding entries of the fault/spare match map. If an entry is “1”, that spare replaces the row currently accessed. In this case, the spare replacing the faulty row will get activated for the access. To disable access to the faulty row, the bits in a row of the fault/spare match map are “NOR”ed and this signal is “AND”ed with the original read/write enable signal.

In all cases, SRAS will add two gate delays (one OR and one AND 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 setup 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.

4.2 Detecting and Diagnosing Faults

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 implementation may necessitate hav-

ing 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 sub-arrays, each of which has k check rows.

4.3 SRAS 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% of the clock period [23]). As discussed in Section 4.1, 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 evaluate its performance overhead in Section 5.

The increase in power consumption in SRAS 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 area overhead of SRAS mostly stems from the spare rows (including spare check rows), since there is only one logic circuit for repair and check for the entire structure. Thus, there is an engineering trade-off between availability and the area overhead incurred for spare rows.

4.4 Limitations of this Implementation

The implementation of SRAS in this paper does not tolerate all microprocessor faults. We divide these untolerated faults into three categories. First, SRAS 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

TABLE 1. Target System Parameters

pipeline depth	22
pipeline width	3
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

makes them less prone to hard faults, but they could still fail. Second, SRAS does not tolerate a fault in a table sub-array if no more spare rows are available in that sub-array. 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 does not tolerate a fault in a sub-array (for a buffer or table) if all of the check rows for that sub-array are faulty.

All of these untolerated faults present the designer with a classic engineering trade-off: fault tolerance versus hardware cost. Future SRAS 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 protection to use more spare rows and more check rows.

5 Evaluation

In this section, we quantitatively evaluate SRAS. We compare our approach to unmodified DIVA (i.e., DIVA without SRAS extensions for hard faults), in order to determine the relative performances in the fault-free and faulty scenarios. We begin by describing our methodology, and then we present our experimental results and the broader applicability of these results.

5.1 System Model and Methodology

We use the SimpleScalar toolset [2] to evaluate our design and compare it to unmodified DIVA. We model a dynamically scheduled microprocessor that is similar to currently available microprocessors, such as the Intel Pentium4 [9] and Alpha 21364 [8]. The details of the target system are shown in Table 1. We simulate DIVA fault tolerance by comparing each instruction’s result to the fault-free result and, if they do not match, triggering a pipeline squash in the aggressive processor core. We simulate the SPEC2000 CPU benchmarks, and we use

the SimPoint toolset [20] to choose statistically representative samples of these long benchmarks for detailed simulation. We inject single-bit stuck-at-1 and all-bits (in a single row) stuck-at-1 faults into the simulated systems. We vary the number of injected faults of these types and examine their impact on system performance.

5.2 Results

In this section, we present our results.

ROB. In Figure 4, we display the results of our experiments with injecting faults into a ROB with 126 entries. This large ROB size corresponds to that of the Pentium4. The two figures—which represent results for the SPEC integer and floating point benchmarks, respectively—plot the speedup of SRAS as compared to unmodified DIVA, as a function of the number of faults injected. The results show that, for most of the benchmarks, SRAS achieves a significant speedup over unmodified SRAS. Even for just a single all-bit fault, speedup results range up to 1.4.

The trends in the graphs reveal a few interesting phenomena. First, the SRAS speedup in the presence of a given number of single-bit faults is always less than the speedup in the presence of the corresponding number of all-bit faults. The insight for this result is that a single-bit fault is far more likely to be logically masked. Second, the results vary across the benchmarks more than one might expect. For unmodified DIVA, the number of recoveries per instruction should be fairly constant across benchmarks. However, as the graphs show, certain benchmarks (e.g., mcf) achieve smaller speedups than others. Also, in general, the integer benchmarks achieve smaller speedups than the floating point benchmarks. We examined the raw performances of every benchmark, in units of instructions per cycle (IPC), and we discovered a direct correlation between IPC and the magnitude of the SRAS speedup. As IPC decreases, the number of recoveries *per cycle* decreases, and thus the impact of SRAS (compared to unmodified DIVA) decreases.

BHT. In Figure 5, we compare the performance of SRAS and unmodified DIVA in the presence of hard faults in the BHT. We only show the results for the case of all-bit faults, since the results for 1-bit faults are almost identical. We observe in this figure that SRAS actually results in a slowdown with respect to unmodified DIVA. There are two reasons for this result. First, the penalty for faulty BHT rows is small, since each individual row is exercised more rarely and is more easily masked. Second, we force a very conservative performance penalty on SRAS by adding an extra pipeline stage for remapping this table. The delay for remapping

is only 2-4% of a pipeline stage, and this delay might not even be on the critical path (e.g., if the BHT access latency is not the critical path determinant of pipeline latency). However, even in the likely case that this added pipeline stage is overly conservative, we would still conclude that SRAS is probably not worth the effort for the BHT.

With faults injected, SRAS speedups are still small and often less than one. Moreover, speedup results are largely independent of the number of faults and the benchmarks. The performance penalty incurred by SRAS, with respect to unmodified DIVA, depends on the impact of adding the pipeline stage in the front-end of the pipeline (i.e., towards fetch). For workloads that have a bottleneck at the back-end (i.e., towards commit), the extra latency in the front-end gets hidden. Thus, even when a branch is mis-predicted and flushes subsequent instructions, the re-fetched instructions can still propagate back into the pipeline before it runs out of instructions to execute.

5.3 Broader Applicability of Results

Experimental results show that the addition of SRAS protection for the ROB is beneficial and that it is probably not a good idea for the BHT. These results also suggest which types of microarchitectural array structures are most likely to benefit from SRAS. The ROB is a heavily-used buffer with a high AVF, which is similar to the register file, reservation stations, and store buffer. We would expect SRAS to benefit these structures, and future work will explore adding SRAS protection to them. Conversely, the BHT is a sparsely-used table with an AVF of zero, which is similar to other tables that are used for prediction, such as for value prediction [12]. We would expect SRAS to have minimal impact on these structures, even if the remapping can be performed without degrading performance in the fault-free case. However, if a prediction table was small, and thus each entry was accessed more frequently, SRAS might help.

6 Conclusions

In this paper, we have developed Self-Repairing Array Structures (SRAS), a hardware technique for masking hard faults in microprocessor array structures. We combine SRAS with DIVA, a cost-effective error correction mechanism that incurs a performance penalty per error. SRAS masks faults by (a) detecting and diagnosing them with dedicated check rows, and (b) using a level of indirection to map out faulty rows. Experimental results show that the addition of SRAS to heavily-used buffers with a high AVF, such as the reorder buffer, improves performance (compared to unmodified DIVA)

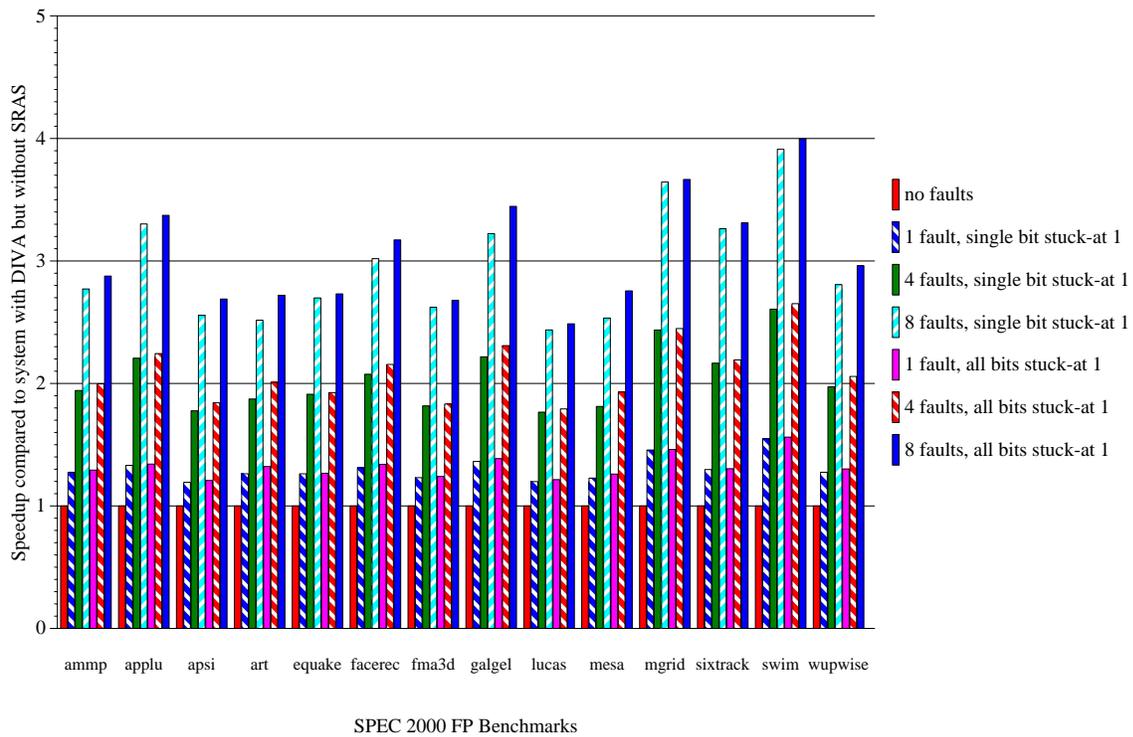
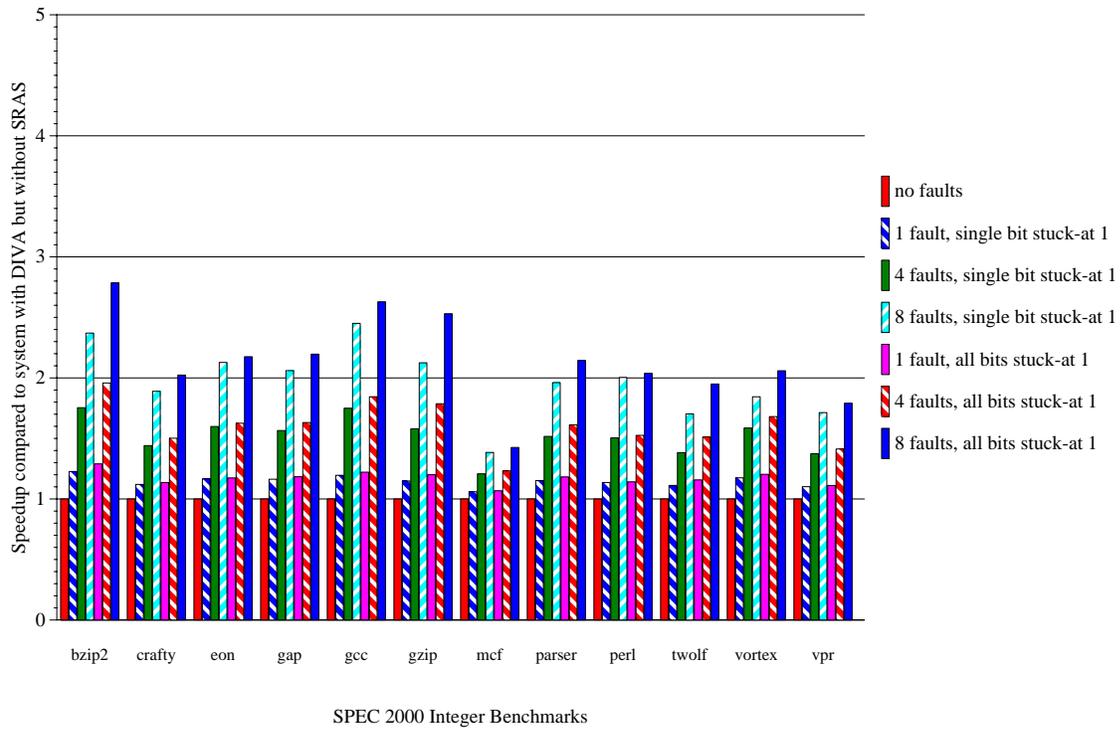
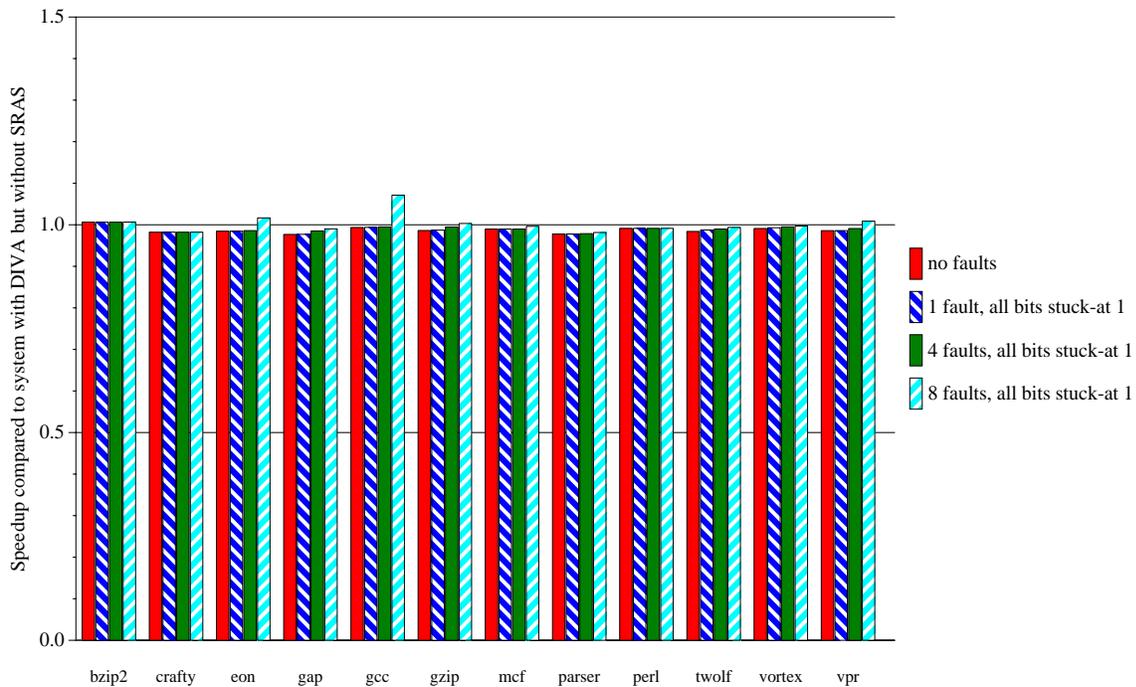
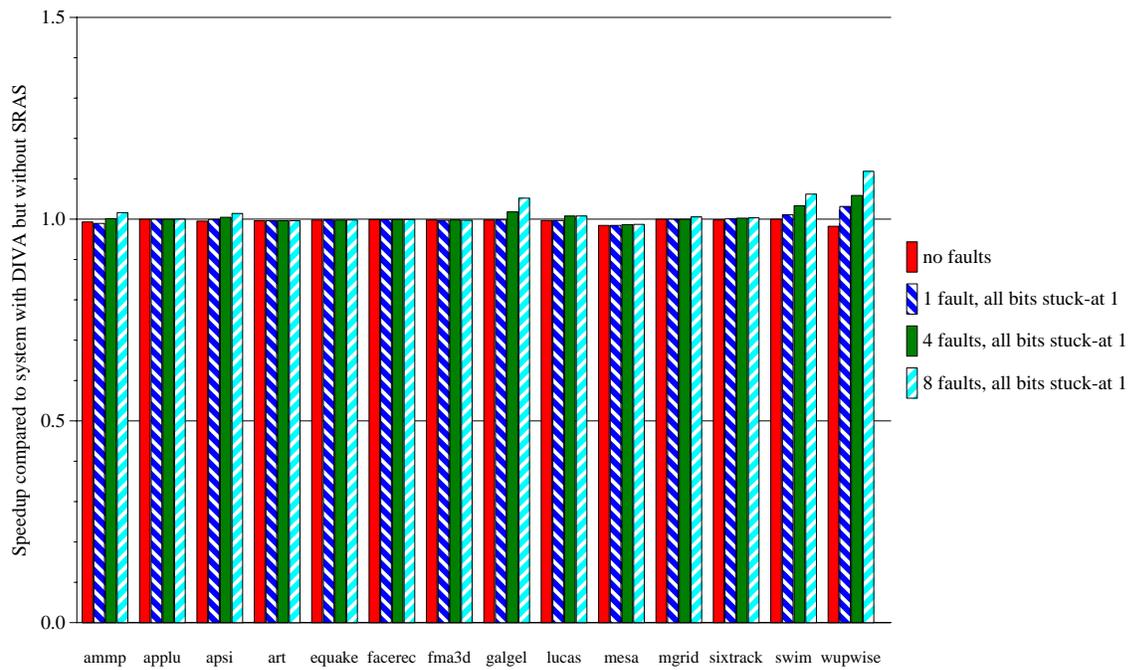


FIGURE 4. ROB Faults: Comparison of SRAS and Unmodified DIVA



SPEC 2000 Integer Benchmarks



SPEC 2000 FP Benchmarks

FIGURE 5. BHT Faults: Comparison of SRAS and Unmodified DIVA

when hard faults are injected into the simulated system. Results also show that SRAS is probably not beneficial for sparsely used tables with a low AVF, such as the branch history table.

Acknowledgments

This work is supported in part by the National Science Foundation, under grants CCR-0309164 and EIA-9972879, IBM, and a Duke Warren Faculty Scholarship. We thank the Duke Architecture group for their insightful comments and criticisms on a draft of this paper.

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. *IEEE Computer*, 35(2):59–67, Feb. 2002.
- [3] T. M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *Proc. of the 32nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.
- [4] R. Blish et al. Critical Reliability Challenges for The International Technology Roadmap for Semiconductors (ITRS). Technical Report 03024377A-TR, International SEMATECH, Mar. 2003.
- [5] T. Chen and G. Sunada. A Self-Testing and Self-Repairing Structure for Ultra-Large Capacity Memories. In *Proc. of the International Test Conference*, pages 623–631, Oct. 1992.
- [6] T. Chen and G. Sunada. An Ultra-Large Capacity Single-Chip Memory Architecture with Self-Testing and Self-Repairing. In *Proc. of the International Conference on Computer Design (ICCD)*, pages 576–581, Oct. 1992.
- [7] T. J. Dell. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory. IBM Microelectronics Division Whitepaper, Nov. 1997.
- [8] L. Gwennap. Alpha 21364 to Ease Memory Bottleneck. *Microprocessor Report*, Oct. 1998.
- [9] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Feb. 2001.
- [10] IBM. Enhancing IBM Netfinity Server Reliability: IBM Chipkill Memory. IBM Whitepaper, Feb. 1999.
- [11] D. Jewett. Integrity S2: A Fault-Tolerant UNIX Platform. In *Proc. of the 21st International Symposium on Fault-Tolerant Computing Systems*, pages 512–519, June 1991.
- [12] M. H. Lipasti and J. P. Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proc. of the 29th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 226–237, Dec. 1996.
- [13] P. Mazumder and J. S. Yih. A Novel Built-In Self-Repair Approach to VLSI Memory Yield Enhancement. In *Proc. of the International Test Conference*, pages 833–841, 1990.
- [14] S. McFarling. Combining Branch Predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.
- [15] S. S. Mukherjee et al. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *Proc. of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, Dec. 2003.
- [16] M. Nicolaidis, N. Achouri, and S. Boutobza. Dynamic Data-bit Memory Built-In Self-Repair. In *Proc. of the International Conference on Computer Aided Design*, pages 588–594, Nov. 2003.
- [17] K. Nikolic, A. Sadek, and M. Forshaw. Fault-Tolerant Techniques for Nanocomputers. *Nanotechnology*, 13:357–362, 2002.
- [18] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proc. of 1988 ACM SIGMOD Conference*, pages 109–116, June 1988.
- [19] K. Sawada, T. Sakurai, Y. Uchino, and K. Yamada. Built-in Self Repair Circuit for High Density ASMIC. In *Proc. of the IEEE Custom Integrated Circuits Conference*, 1989.
- [20] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically Characterizing Large Scale Program Behavior. In *Proc. of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.
- [21] J. E. Smith. A Study of Branch Prediction Strategies. In *Proc. of the 8th Annual Symposium on Computer Architecture*, pages 135–148, May 1981.
- [22] L. Spainhower and T. A. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(5/6), September/November 1999.
- [23] S. Thompson et al. An Enhanced 130nm Generation Logic Technology Featuring 60nm Transistors for High Performance and Low Power at 0.7-1.4V. In *Proc. of the International Electron Devices Meeting*, pages 257–260, Dec. 2001.
- [24] D. Wilson. The Stratus Computer System. In *Resilient Computer Systems*, pages 208–231, 1985.
- [25] T.-Y. Yeh and Y. Patt. Two-level Adaptive Training Branch Prediction. In *Proc. of the 24th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 51–61, Nov. 1991.
- [26] L. Youngs and S. Paramanandam. Mapping and Repairing Embedded-Memory Defects. *IEEE Design & Test of Computers*, pages 18–24, January-March 1997.