# Cost-Efficient Soft Error Protection for Embedded Microprocessors

Jason A. Blome, Shantanu Gupta, Shuguang Feng, Scott Mahlke
Advanced Computer Architecture Lab
University of Michigan - Ann Arbor, MI
{jblome, shangupt, shoe, mahlke}@umich.edu

Daryl Bradley
ARM Ltd.
Cambridge, United Kingdom
daryl.bradley@arm.com

## ABSTRACT

Device scaling trends dramatically increase the susceptibility of microprocessors to soft errors. Further, mounting demand for embedded microprocessors in a wide array of safety critical applications, ranging from automobiles to pacemakers, compounds the importance of addressing the soft error problem. Historically, soft error tolerance techniques have been targeted mainly at high-end server markets, leading to solutions such as coarse-grained modular redundancy and redundant multithreading. However, these techniques tend to be prohibitively expensive to implement in the embedded design space. To address this problem, we first present a thorough analysis of the effects of soft errors on a production-grade, fully synthesized implementation of an ARM926EJ-S embedded microprocessor. We then leverage this analysis in the design of two orthogonal low-cost soft error protection techniques that can be tuned to achieve variable levels of fault coverage as a function of area and power constraints. The first technique uses a small cache of live register values in order to provide nearly twice the fault coverage of a register file protected using traditional error correcting codes at little or no additional area cost. The second technique is a statistical method used to significantly reduce the overhead of deploying time-delayed shadow latches for low-latency fault detection.

## Categories and Subject Descriptors

B.5.3 [**Reliability and Testing**]: [Built-in tests]; C.3 [**Special-Purpose and Application-Based Systems**]: [Real-time and Embedded Systems]

## General Terms

Reliability, Design, Experimentation

## Keywords

Reliability, Soft Errors, Embedded Processors

## 1. INTRODUCTION

A soft error, or single event upset (SEU), is defined as a transient piece of incorrect machine state. A soft error in logic occurs when the result of a transient fault in logic propagates to a storage element and is latched. A soft error in a memory element occurs when sufficient charge is generated to invert the value stored in the memory element. Transient faults can be the result of electrical noise, such as crosstalk, or high-energy particle strikes. Soft errors due to energetic particle strikes are typically caused by either alpha particles, which can be emitted by radioactive contaminants in microprocessor packaging materials, or high-energy neutrons from cosmic radiation. While dealing with alpha particles is largely a manufacturing issue, addressing neutron strikes poses a significant problem because adequate shielding is prohibitively expensive.

Current device scaling trends suggest that dramatic increases in microprocessor soft error rates (SER) are inevitable. Device scaling results in lower operating voltages, which in turn reduces the energy required to cause a voltage pulse at the output of a logic gate or invert the value stored within a sequential element. Thus, lower-energy particle strikes that did not pose a threat in past technology generations could induce transient errors in future technology generations. Further, the rate of particle strikes increases exponentially as the energy level of the particles decrease [19]. Therefore, with each new technology generation, the rate of particle strikes that may potentially affect the logical operation of the microprocessor increases significantly. These trends, coupled with an explosive growth of embedded microprocessor distribution for a number of safety-critical applications suggests a strong need for understanding reliability as it applies to the embedded design space.

Traditionally, reliability research has focused largely on the high-performance server market. High availability systems, such as the IBM G5 server [17] and the HP NonStop architecture [4], rely on large scale modular redundancy to provide fault tolerance. Other research has focused on providing fault protection using redundant multithreading [13, 14]. In general, these techniques are expensive in terms of both the area and power required for redundant computation and are not generally appropriate for embedded designs.

The design constraints of the embedded domain differ substantially from those in the high-performance arena. In the embedded design space, area and power are primary constraints which are balanced with processor performance. This typically leads to longer clock cycle times, larger logic depths between sequential state elements, and a higher degree of signal fan-out in embedded designs. Further, high performance microprocessors typically employ a large amount of out-of-order execution hardware and speculative state. These mechanisms can decrease the overall utilization of the chip, lowering the probability that a particle strike will affect a sensitive piece of state within the design, and increasing the amount of fault masking. In general, high performance microprocessor cores tend to have much more area devoted to sequential state than to combinational logic. This invariably affects the be-

*CASES'06,* October 23–25, 2006, Seoul, Korea.
Copyright 2006 ACM 1-59593-543-6/06/0010 ...$5.00.

havior of soft errors on the design. Since the design constraints for the embedded domain are considerably different from those in the high-performance domain, it stands to reason that methods for fault tolerance will also vary dramatically.

In order to fully understand the way in which soft errors affect embedded microprocessors, we conduct a thorough analysis of the behavior of faults on an ARM926EJ-S embedded core. In these experiments, we measure the amount of fault masking that occurs when faults are injected into both state elements and combinational logic and also analyze the propagation behavior of the errors throughout the design. We then use this study to motivate two soft error mitigation techniques appropriate for the embedded design space. First, we propose the register value cache, a small and efficient mechanism that protects the register file against faults occurring in both sequential state elements and combinational read/write logic within the register file. This design provides higher fault coverage at a lower area overhead than traditional error correcting codes (ECC). Second, we propose a method for the strategic deployment of transient pulse detectors using time-delayed shadow latches, which provides a high degree of fault coverage for the rest of the design, while only requiring a small number of detectors.

The contributions of this work are as follows:

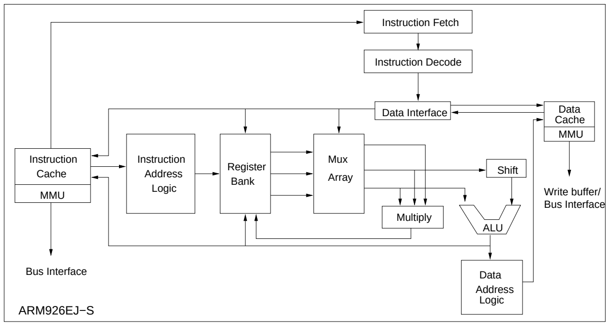
- An empirical derivation of the logical and temporal soft error masking rates for a commercial embedded microprocessor.
- An analysis of the error fan-out and propagation behavior of soft errors in a commercial embedded microprocessor.
- A lightweight architectural technique for protecting register files from faults in both combinational and sequential logic.
- A statistical technique for deploying time-delayed shadow latches for tolerating soft errors occurring in arbitrary logic elements.

# 2. FAULT ANALYSIS OF THE ARM926EJ-S PROCESSOR CORE

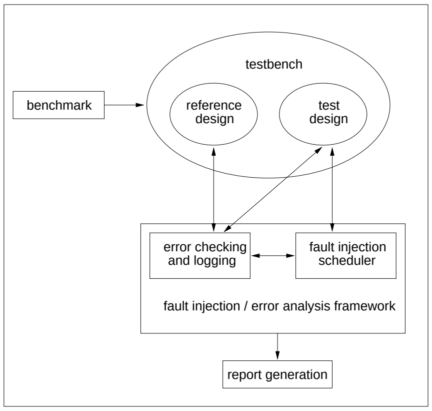
Though the effects of transient faults on high-performance microarchitectures have been studied in the past [11] [18], relatively little published data exists regarding their effects on embedded designs. Previous work that does involve embedded-style microprocessors typically focuses on the effects of transient faults at the macrocell and software levels [8] [15]. In order to motivate architectural and microarchitectural solutions to the soft error problem, the goal of the following analysis is to understand how transient faults affect an embedded microprocessor at the circuit level and how these faults propagate throughout system state. In this section, we provide a brief overview of the ARM926EJ-S microprocessor, describe our fault analysis framework, and present detailed results describing fault masking and propagation throughout the microprocessor core.

## 2.1 Fault Analysis Framework

The soft error analysis conducted in this work uses a Verilog model of an ARM926EJ-S microprocessor [1]. The ARM926EJ-S is a 32-bit embedded architecture with a standard five stage pipeline consisting of fetch, decode, execute, memory and write-back stages. The datapath of the core is depicted in Figure 1. The implementation used in this work has 37 architecturally defined registers (31 32-bit general purpose and six status registers), 4 KB of instruction cache, and 4 KB of data cache. The Verilog model was synthesized using Synopsys Physical Compiler with scan-chain insertion and design-for-test methodologies using an Artisan library characterized for a 130 nm process. The synthesized netlist and a hand-designed floorplan were processed in Avanti Astro for clock tree synthesis and physical placement. Once fully synthesized with all design rule constraints satisfied, timing information was extracted in Standard Delay Format so that it could be annotated onto the netlist and simulated using Synopsys VCS.



**1:** A block diagram of the ARM926EJ-S five-stage pipeline


testbench
benchmark
reference design
test design
error checking and logging
fault injection scheduler
fault injection / error analysis framework
report generation

**2:** Overview of the soft error injection and analysis framework

The testbench used for simulation instantiates a pair of the synthesized netlists: a reference design and the unit under test. Both netlists are annotated with the timing information gathered from the synthesis and layout tools. The testbench also includes a behavioral memory model that is used to load benchmarks for the simulations. An overview of the soft error test harness is shown in Figure 2.

The soft error injection and analysis framework used in our experiments is composed of a set of Verilog Programming Interface (VPI) libraries that are invoked at the start of each simulation. Upon invocation, the framework probes the design to derive the set of all sequential state elements and combinational logic gates within the unit under test. Depending on the simulation parameters, the framework may schedule fault injection experiments at arbitrary points in time for arbitrary durations, selecting a random design element (register or logic gate) as a fault injection target and inverting the value at the node's output.

Experiments are conducted targeting both sequential state elements and combinational logic gates. Workload-specific analysis is carried out by running benchmark code for an image processing algorithm which maps an input image from the RGB to the YUV color space. Upon initialization, the framework will select a random point in time between 2,500 and 5,000 cycles after the start of simulation to conduct its first fault injection. If the fault is to be injected into a combinational logic element, the fault injection time is randomly selected in picoseconds, and the fault duration is randomly selected on the interval $[0.25 * CLK, CLK]$, where $CLK$ is the cycle time of 5 ns. If the experiment is for a sequential state element, the fault injection time is scheduled at a random future rising edge of the clock signal and will be held for the duration of one cycle.

At fault injection time, depending on the type of injection experiment being simulated (faults in combinational logic or sequential state), a random design element is selected for fault injection from the unit under test. If the fault is to be injected into a logic element, a random logic gate in the design is selected and the value present on it's output is inverted, simulating a transient fault caused by a particle strike. Similarly, when faults in registers are being simulated, a random register is selected and its output is inverted. When a fault is injected into the design, the framework logs the fault site, the time of injection, and the pulse duration.

After a fault has been injected into the system, *every* microarchitectural register in the unit under test is compared against its dual in the reference design at each subsequent rising clock edge. Further, all top-level output ports on the design (I/O buses, coprocessor interface, test equipment) and inputs into the caches are checked to ensure that no corrupt values have escaped from the core datapath. If, during the first cycle after fault injection, no register, cache, or top-level port mismatches occur, the injected fault did not affect the system, and so a new random time in the future is selected for another fault injection experiment. If any register, cache, or port mismatches does occur, the fault analysis framework logs the relative cycle and site of the error for later analysis.

The fault analysis framework continues to track the progress of errors throughout the system for a given number of cycles after the fault injection. If during this period, no errors are present, and no errors have propagated out to the caches or top-level ports, the system is clean, and the fault was successfully masked, allowing a new time for fault injection to be scheduled. If top-level port or cache errors did occur, or a latent error still lingers in the design that has not yet affected architectural state, then simulation halts, and error logs are written for post-processing to analyze propagation and architectural state effects. Though latent errors in the design may not have caused errors in software-visible state, they still pose a threat and may potentially cause data corruption given a more diverse workload.

## 2.2 Fault Analysis Results

In this section, we provide an empirical derivation of the soft error masking rate for the ARM926EJ-S processor core, as well as a detailed examination of the soft error propagation behavior. The soft error rate is directly related to a set of derating factors that mask faults from being latched at the output of a circuit. The three circuit-level derating factors that affect the SER are as follows:

- *Logical masking*: Logical masking occurs when a transient pulse is effectively gated from all possible target sequential state elements; for example, a transient pulse at the output of a circuit that is ANDed with 0 will be logically masked.

- *Temporal masking*: Temporal masking (or latching-window masking) occurs when a transient pulse propagates to a state element, but does not arrive within the capture window of the state element.

- *Electrical masking*: Electrical masking occurs when a transient pulse is attenuated by subsequent logic gates such that the pulse does not affect the output of the circuit.

The experiments presented in this work examine the effects of both logical and temporal masking on the overall soft error rate, but leave electrical masking for future work.

| Error Location | Logical Masking Rate |
|---|---|
| Microarchitectural state | 6.47% |
| Architectural state | 88.35% |
| Top-level port | 89.32% |

**1:** Average logical masking rate for soft errors occurring in sequential state elements.

### 2.2.1 Logical and Temporal Masking

In this set of experiments, we derive the rates at which injected faults are masked from affecting different classes of processor state: microarchitectural state, architectural state, and the top-level output ports of the design (I/O buses, coprocessor interface, test equipment) while the image processing benchmark, rgb2yuv, is executed on the processor model. We define the architectural state as the set of 37 software-visible physical registers defined in the ARM instruction set architecture [16] and the microarchitectural state as the set of all state elements within the design, excluding the architectural registers. In the first experiment, we restrict our fault injections to only sequential state elements and observe the rate at which the different classes of errors appear over a period of 200 cycles subsequent to fault injection. The results of this experiment is presented in Table 1. These results demonstrate the average logical masking rate for faults occurring in sequential state elements.

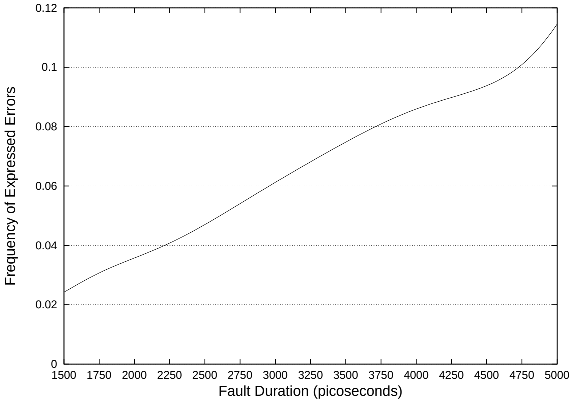
Table 1 shows that when a fault occurs within a state element, it is very common for microarchitectural state to remain corrupted in the cycles subsequent to fault injection, with only about 6% of faults being masked from affecting the microarchitectural state. Though the microarchitectural state masking rate is quite low (and thus the error rate is quite high), these faults rarely propagate into the architectural register file or to the top-level ports of the design where they have the potential to corrupt software state. The second row in Table 1 shows that only about 11% of the injected faults actually affect the ARM926EJ-S architectural registers, and the third row shows that even fewer faults are propagated outside of the core from toplevel ports within the 200 cycle experiment window.

Since combinational logic nodes consume nearly 58% of the cell area of the ARM926EJ-S design, we expand upon the previous study and next examine the effects of faults injected at arbitrary combinational logic nodes. Here we introduce two sets of data, the first presents the masking rates for the different types of errors, based on a pulse being injected at clock cycle boundaries and lasting for the duration of an entire clock cycle. This experiment yields the average logical masking rate for faults occurring at arbitrary logic nodes. Then, in the second experiment, we uniformly select a random point in time at which to inject a fault, irrespective of clock cycle boundaries. We hold this fault for a random duration on the interval $[0.25 * CLK, CLK]$. This experiment results in the average combined temporal and logical masking rate of the microprocessor core. The results of these two experiments are shown in columns two and three respectively of Table 2.

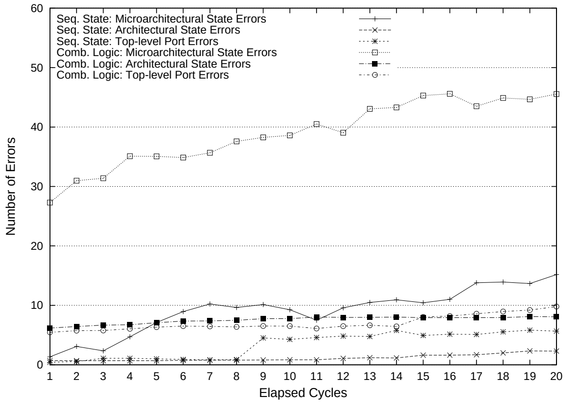
The results presented in Table 2 demonstrate that the microarchitectural masking rate for faults in combinational logic is substantially higher than the microarchitectural masking rate observed for faults occurring in sequential state. Nevertheless, it would be a mistake to take this to mean that faults in combinational logic are less significant than faults in sequential state. Though there is a large disparity between these microarchitectural masking rates, the difference is not nearly so pronounced at the software interface. Table 2 shows that the architectural masking rate for faults occurring in logic is only about 8% greater than for faults occurring in

| Error Location | Logical Masking Rate | Logical + Timing Masking Rate |
|---|---|---|
| Microarchitectural state | 78.44% | 83.76% |
| Architectural state | 94.74% | 96.59% |
| Top-level port | 95.12% | 96.33% |

**2:** Average logical and logical and temporal masking rates for soft errors occurring in combinational logic.



**3:** Relative frequency for which incorrect state is observed within the processor as a function of transient pulse duration



**4:** Average number of incorrect bits for various error types over the span of twenty cycles following the fault injection. Faults are injected in sequential state and combinational logic elements

sequential state. Further, even though the architectural masking rate for faults occurring in combinational logic tends to be slightly higher, we will demonstrate in Section 2.2.2 that faults in combinational logic tend to be much more insidious.

In order to illustrate the effects of latching-window masking and pulse duration on the overall soft error rate, we provide a further analysis of faults occurring at worst-case nodes in the design. In this experiment, we restrict fault injection to the output of sequential state elements and vary the pulse duration randomly across the clock cycle time, thus ensuring the worst case delay for each propagation path between the fault injection site and the target state elements. Histogram data representing the frequency with which a given fault duration causes an error is shown in Figure 3.

Figure 3 shows that there is a definite correlation between the fault duration and the likelihood that errors are expressed in the microprocessor. However, this figure demonstrates that even relatively small pulses may be responsible for a significant percentage of the total soft error rate. In addition, it is important to note that the results presented in Figure 3 are conservative, since particle strikes are likely to be random throughout the depth of the circuit. Further, as technology scales, the latching window will become a more significant portion of the clock cycle time, further marginalizing the effects of latching window masking.

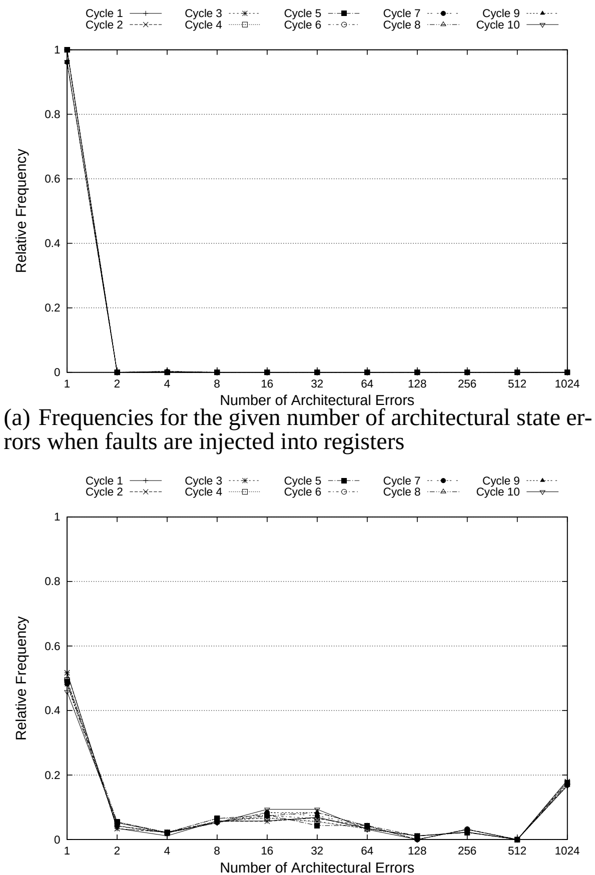
### *2.2.2 Soft Error Propagation Behavior*

In this section, we analyze how soft errors propagate throughout the microprocessor core over time. The effects of transient faults on both architectural and microarchitectural state as well as the toplevel ports of the design are analyzed in the cycles subsequent to fault injection. In these experiments, only fault injection data for those faults that have caused at least a single bit error are used, and the number and type of errors present in the core over the cycles following fault injection are analyzed. Figure 4 demonstrates the average number of state bit errors for each class of error discussed in the previous section. In Figure 4, each data point represents the average number of bit errors that are present in the design for a particular error class, given that at least a single error of that class was expressed over the course of the experiment.

Figure 4 demonstrates a large disparity between how faults occurring in sequential state affect the processor core as opposed to faults in combinational logic. In the first cycle after fault injection, faults occurring in combinational logic typically cause more than 27 bits of incorrect microarchitectural state and can cause six bits of corrupt architectural state on average, whereas faults in sequential state typically only cause only single bit errors of any class. From Figure 4, it is clear that faults occurring in combinational logic cannot be accurately modeled as single bit state errors. It is important to note that although Section 2.2.1 shows the masking rate for faults in combinational logic is about 8% higher, more than 58% of the design is consumed by combinational logic, meaning that more faults are likely to occur in the combinational logic than in the sequential state elements, potentially offsetting the descripency in masking. Further, these faults are likely to corrupt significantly more system state. It is also interesting to note from Figure 4, on average, when faults in logic cause an error in architectural state, they often cause multi-bit errors within the register file, whereas faults in state elements rarely demonstrate such effects.

In order to better understand the propagation of errors within the system and how they may potentially affect software state, the next experiment is focussed solely on the propagation behavior of soft errors into architectural state. Figures 5(a) and 5(b) demonstrate the relative frequency of architectural state bit errors manifested during the ten cycles after fault injection.

Figure 5(a) shows that multi-bit architectural state errors tend to occur very rarely when faults are injected into sequential logic. This corroborates the position that simply applying error correcting codes (ECC) to the words stored within the register file is a potentially valuable tool in protecting against the effects of transient faults occurring in state elements. However, Figure 5(b), demonstrates that when transient faults occur in combinational logic, and they manifest as errors in architectural state, multi-bit errors of four bits or more account for more than 45% of the occurrences. Further, as shown by the spike at the tail end of Figure 5(b), more than 15% of the faults occurring in combinational logic cause more than 90% of the architectural state bits ($\sim 1000$ state elements) to hold



(a) Frequencies for the given number of architectural state errors when faults are injected into registers

(b) Frequencies for the given number of architectural state errors when faults are injected into logic

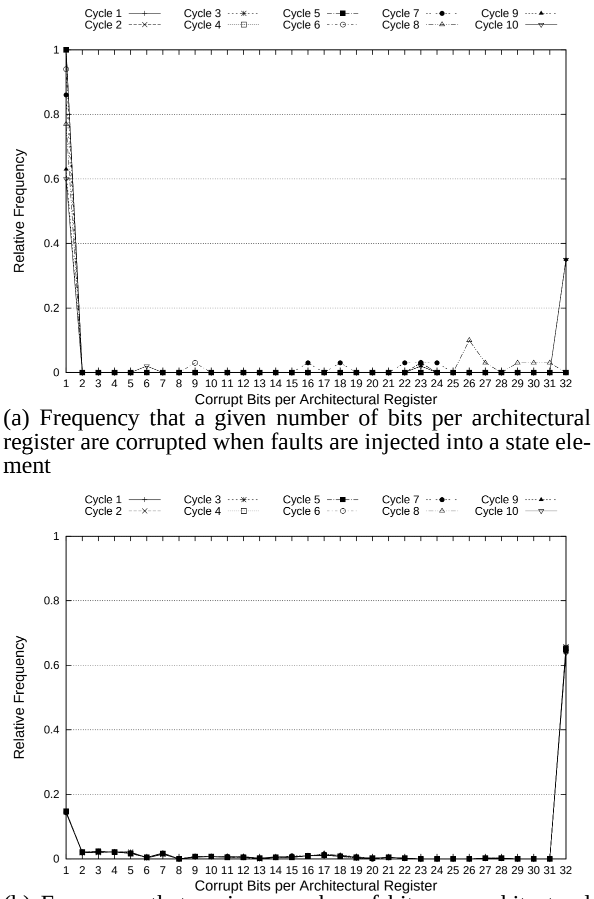
**5:** Architectural state errors

incorrect values. This is typically the result of faults occurring in extremely sensitive design for test logic, such as scan-enable nodes.

To highlight the nuances of the architectural effects presented in Figure 5(b), we also study the number of incorrect bits per architectural register that are corrupted over time. Figure 6(a) the majority of faults injected into state elements cause only single bit errors in architectural state.In contrast to this, Figure 6(b) demonstrates that multi-bit errors, where the entire 32 bits of the architectural register are corrupted, tend to be the norm for faults occurring in combinational logic. Very often, this is the result of faults occurring in the read/write logic of the register file, causing an incorrect register to be read or written. In the case of an incorrect write address, the result is actually a pair of incorrect 32-bit architectural registers, one for the incorrectly written register and one for the register that should have been written, neither of which could have been recovered using ECC.

### 2.2.3 *Soft Error Analysis Discussion*

The experiments conducted in Section 2.2.2 lead to two important insights about how to efficiently provide soft error protection. First, the majority of the faults (both in combinational and sequential logic) that affect the architectural state of the processor actually occur within the register file. The standard practice of applying ECC can only handle faults occurring directly in the register state array and does not provide coverage for faults ocurring in the read/write logic. Since the majority of the cell area within the register file is consumed by combinational logic, it is clear that ECC is unlikely to supply adequate protection. Therefore, a low-cost technique that defends the register file against faults in both the state array *and* the combinational logic is necessary to provide significant protection for the microprocessor core.



(a) Frequency that a given number of bits per architectural register are corrupted when faults are injected into a state element

(b) Frequency that a given number of bits per architectural register are corrupted when faults are injected into logic

**6:** Number of bits that are corrupted per architectural register

The second insight is derived from the observation that faults occurring in arbitrary combinational logic exhibit a high degree of fanout, causing errors in a large number of sequential state elements. This property can be exploited to minimize the number of fault detection sites necessary to provide adequate coverage. By using the set of most vulnerable state elements (nodes with high error fan-in) to guide the placement of fault detection units it is possible to achieve high fault coverage with minimal overhead.

In the following sections, we present two techniques which leverage these insights to provide significant soft error coverage with little area and power overhead.

## 3. FAULT MITIGATION TECHNIQUES

In this section, we leverage the data presented in Section 2.2 to motivate two complementary techniques for addressing soft errors within an embedded microprocessor. The purpose of the techniques presented in this section is to provide scalable mechanisms for addressing a statistically significant portion of faults with minimal area and power overhead. We show that by targeting only highly vulnerable portions of the design, we can achieve substantial fault coverage with little expense.

Our analyses demonstrate that the majority of faults affecting the operation of software occur within the architectural register file. The first technique that we present addresses these faults using a mechanism referred to as the *register value cache* (RVC) in Section 3.1. The RVC relies on locality of reference, maintaining duplicate copies of only the most recently used register data in order to provide high fault coverage. Unlike traditional mechanisms, such as ECC, the RVC protects against faults occurring in both the combinational logic and the state array, yielding more than twice the fault coverage. Further, the coverage provided by the RVC may be increased by simply adding more cache entries, thus balancing constraints for area and power against fault tolerance.

To address faults occurring outside of the register file, we leverage the significant amount of observed fault fan-out within the core, demonstrated in Section 2.2, to strategically deploy transient pulse detectors at high fan-in state elements. We then use these detectors to proactively flush processor state and correct transient errors occurring in microarchitectural state. The process of determining the most effective location for these pulse detectors and inserting them into the design is fully automatable and the subject of ongoing research. We demonstrate how this technique can be used to translate available area and power into fault coverage in Section 3.2.

## 3.1 Register Value Cache

The register file in the ARM926EJ-S microprocessor consumes only 8.7% of the total core area, yet 57.4% of the faults that result in errors at the software interface (architectural registers, memory, instruction cache, or data cache) occur within the register file. This implies that any efficient strategy for tolerating soft errors must implement a mechanism for handling faults occurring within the register file.

The traditional technique for dealing with faults occurring in large state arrays is to employ error correcting codes (ECC). However, ECC in the context of the register file is problematic for several reasons, especially for processors implementing the ARM instruction set. First, the ARM instruction set allows for up to three register read operations and up to two write operations per cycle. This significantly complicates the logic within the register file and requires that an ECC protected register file include three ECC encode units and two ECC decode and correct units, each of which is expensive to implement, both in power and area. In addition, ECC is limited in terms of the amount of fault coverage it can provide for the register file, since ECC can only protect against faults which occur directly in the register state array.

Sequential state consumes only 44.1% of the ARM926EJ-S register file cell area. If particle strikes are assumed to be uniformly distributed over the area of the processor core, then more faults are likely to occur in the read/write logic of the register file than in the state array. This has serious implications for the efficacy of ECC as a soft error tolerance mechanism. Since ECC is only capable of correcting single bit errors occurring in the actual state array, more than 55% of the faults occurring in the register file will go potentially undetected.

In order to address the shortcomings of ECC in protecting the register file, we propose a new mechanism called the *register value cache* (RVC). The RVC is capable of detecting and correcting faults occurring in both the combinational and sequential logic of the register file. It maintains duplicate copies of the most recently accessed values within the register file, allowing for a high degree of fault coverage without duplicating the entire register file. Also, since the inputs to the register file are split off and fed directly to the RVC, the read/write control logic for the register file is essentially duplicated. A detailed schematic of the RVC implementation is shown in Figure 7.

The RVC is implemented as a separate module alongside the register file, and all read/write address and enable inputs to the register file are duplicated and fed to the RVC. On a read operation, if the RVC contains the requested read value, it provides the duplicate data and asserts the appropriate valid signal. The read results from the RVC and the register file are then compared and the result of the comparison is ANDed with the valid signal to determine if an error is present. If an error is present, the processor stall signal is asserted and the processor pipeline must stall for one cycle while the RVC determines whether the value it supplied was correct. This is done by conducting a cyclic redundancy check (CRC) on the register value contained in the cache. If the CRC check fails, then the value contained in the register file was correct, while if the CRC check passes, then the value in the RVC was correct. It is assumed here that the probability of multiple particle strikes where both the RVC *and* the register file are corrupted *and* the corrupted values correspond to the same register value is negligible.

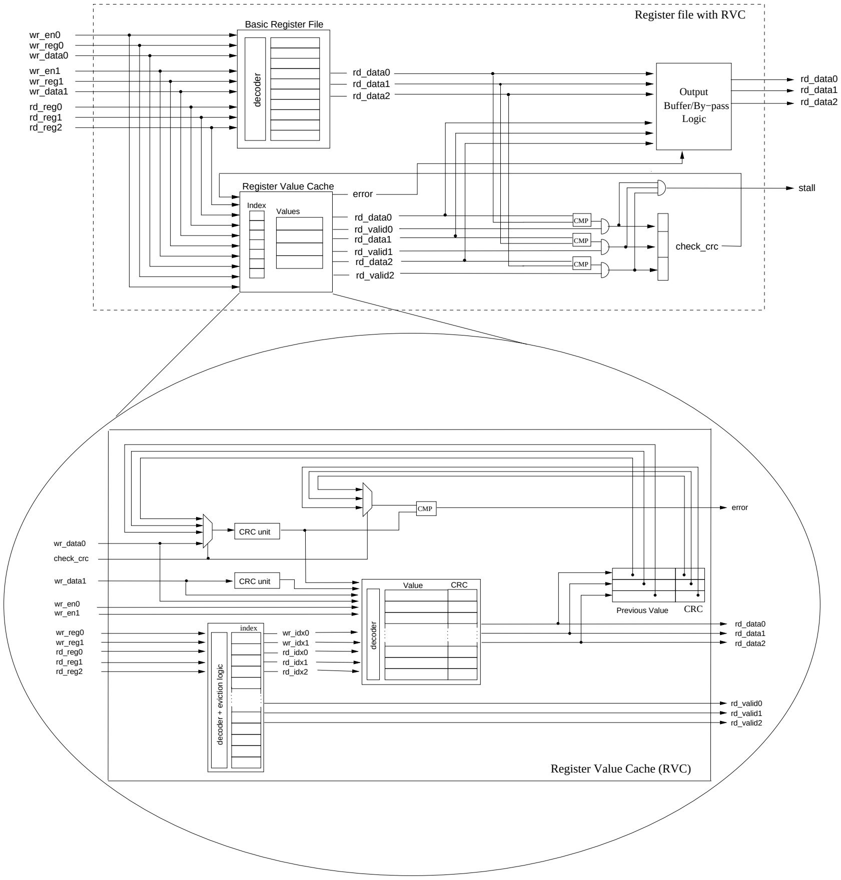
There are two main reasons for using CRC in the RVC rather than ECC. First, the former is much cheaper to implement in terms of both area and power. Second, since there are already two copies of the register read data, it is only necessary to determine which of the two values is correct, not to actually find and correct the error, removing the need for expensive error correction logic.

The RVC employs two CRC units for the purpose of encoding and decoding read/write values for the cache. The CRC units use a five bit CRC polynomial for encoding up to two write values per cycle. Since the CRC value is only checked when the output from the register file does not match the output of the RVC, at which point the processor pipeline is already stalled, one of the CRC units may be reused for checking the CRC on a mismatched read value. The operation of the RVC on read and write requests is described as follows:

**Write request:** When the write enable signal is asserted, the RVC checks to see if the register to be written already has an entry assigned to it in the value array. This is done by checking the index array value corresponding to the register number. If the index array value is valid, it is used as the write address for the value array. In the case that the index array value is not valid, the least recently used value is evicted from the cache. While the write value address calculation is taking place, a five-bit CRC value is computed and forwarded to the value array.

**Read request:** On a register read request, the index array entry corresponding to the register read number is checked for a valid entry. If the entry is valid, the associated output valid signal is asserted and the address from the index array entry is forwarded to the value array. The value array sets the read output data to the value stored in the cache, and forwards the read value as well as its CRC value to the *previous value* buffer. The read value and the CRC must be copied to this temporary buffer in order to handle the case where a read value mismatch has occurred for a register that was both read from and written to in the previous cycle. If the index array entry is not valid, the appropriate valid line is deasserted and the output data value is set to {`32'bx`}.

The output from the register file and RVC are compared, and if the comparison fails AND the valid line for the read address is asserted, the two read results are temporarily buffered and the pipeline is stalled for one cycle. During this cycle, the RVC conducts a CRC check on the previously read RVC value. If the CRC check fails, then the error signal is asserted and the buffered value from the register file is identified as the correct value, otherwise the buffered value from the cache is assumed to be the correct value.



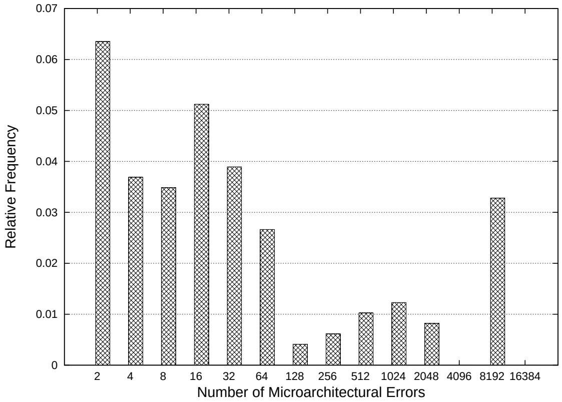
**7:** Incorporation of the register value cache into a processor core

## 3.2 Time-delayed Shadow Latches

In order to provide fault tolerance for faults occurring outside of the register file, we first determine which of these faults are most critical. Our results demonstrate that less than 30% of the faults injected into sequential logic which are observable at the software interface occur outside of the register file, as opposed to more than 50% for faults injected into combinational logic. In order to achieve a high degree of fault coverage while still maintaining low overhead, we specifically target faults occurring in combinational logic using transient pulse detectors. Though these pulse detectors are ideal for detecting faults in combinational logic, they also provide fault detection for faults occurring in sequential state elements as well.

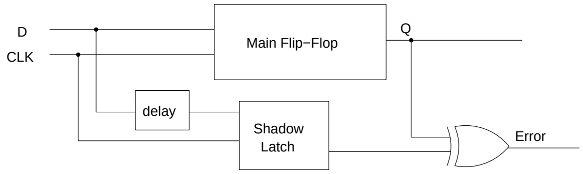
To maintain low power and area overhead, we exploit the fault fanout observed in Section 2.2 to strategically place transient pulse detection units at high fan-in sequential state elements. Conveniently, these high fan-in elements, also tend to be the nodes responsible for generating errors at the software interface when faults are injected into sequential state, and so these detectors also provide fault coverage for faults occurring in state elements. Since these detectors are being used specifically to protect against soft errors occurring in the microarchitectural state outside of the register file, once an error is detected, it can be corrected by simply flushing the processor pipeline.

To help motivate the strategic placement of detectors at only a small subset of state elements within the design, we demonstrate the frequency of multi-bit errors that are observed in the cycle directly after a fault has been injected into combinational logic. These results are shown in Figure 8.



**8:** Frequencies for the given number of microarchitectural state errors for faults that are injected into combinational logic.

Figure 8 shows that more than 30% of faults in combinational logic, that actually result in errors, result in multi-bit errors. Further, more than 20% of these errors are of ten bits or more. It is also interesting to note the outlying data which shows that almost 5% of the faults result in more than 8,000 incorrect bits of state. This sort of occurrence is especially problematic and typically the result of faults occurring at nodes used in design for test, such as scan chains, and other general test logic.



**9:** Time-delayed shadow latch used for transition detection.

In Figure 9, we demonstrate the design for the transition detection circuit used in this work, which is presented in the Razor [2] dynamic voltage scaling system. This figure depicts a standard flip-flop, which is augmented with a time-delayed shadow latch. For this detector, the input signal to the latch is split off and subjected to a delay proportional to the width of the pulses that are to be detected. The delayed input is then passed to a shadow latch and an error signal is generated if the output of the shadow latch and the flip-flop do not match. This particular detection mechanism is capable of detecting faults occurring both in the logic cone feeding the flip-flop, and the flip-flop itself. There are a number engineering difficulties involved in implementing such a system, which are beyond the scope of this paper, that are discussed in [5].

Since an automated tool for placing these detectors and integrating them into the microprocessor core is the subject of ongoing research and future work, we present a manual technique for placing fault detectors in order to determine the achievable fault coverage as a function of the number of detection units used in the design. To determine the most valuable detection points, we first conduct statistical analysis using Monte Carlo fault injection simulations on the ARM926EJ-S core. We then rank order the set of flip-flops which should be protected based on the number of unique fault sites that would be covered by protecting the flip-flop. Once this rank-order has been generated, the amount of fault coverage for the chip can be incrementally improved by replacing traditional flip-flops with the enhanced flip-flops within the design. This allows chip designers to systematically tweak fault coverage as a function of the area and power budgets available for fault tolerance. In Section 4.2, we explore the cost versus fault coverage achieved by augmenting the ARM926EJ-S with these transition detection circuits.

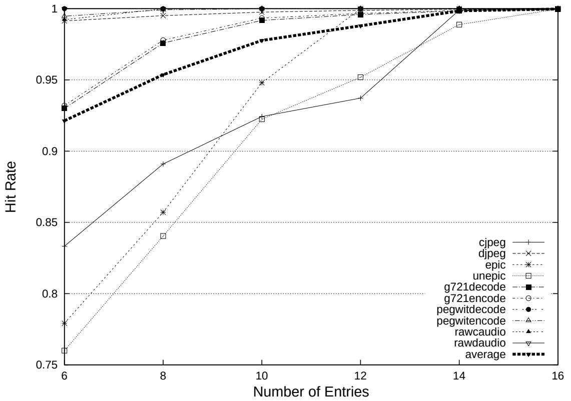
# 4. EVALUATION

In this section, we conduct a number of experiments to demonstrate the efficacy of the two proposed soft error protection mechanisms. We evaluate the fault coverage provided when each technique is applied in isolation as well as when they are employed cooperatively. For each experiment, we detail the amount of fault coverage gained with respect to its cost, in terms of both area and power.

## 4.1 Register Value Cache Analysis

To evaluate the RVC, we first derive a bound for the maximum fault coverage achievable. This bound is calculated by analyzing the hit rate of RVCs of various sizes and multiplying the hit rate by the percentage of faults that occur within the register file that lead to errors at the software interface. We determine the hit rate by simulating the RVC operating on a set of traces from the MediaBench benchmarks [9]. These benchmarks were compiled using the arm-linux-gcc cross compiler version 2.95 and were simulated on the SimpleScalar ARM926EJ simulator [3]. The hit rates for various size RVCs are shown in Figure 10.

As shown in Figure 10, an RVC with 6 entries demonstrates an average hit rate of more than 90%, and an RVC with 8 entries has an average hit rate of better than 95%. Some benchmarks exhibited hit rates of 99% with a cache of only six entries, but on average, a 99% hit rate required at least a 12 entry cache. Hit rates of this magnitude demonstrate that high fault coverage can be gained with relatively few cache entries, allowing for small and efficient RVC designs.



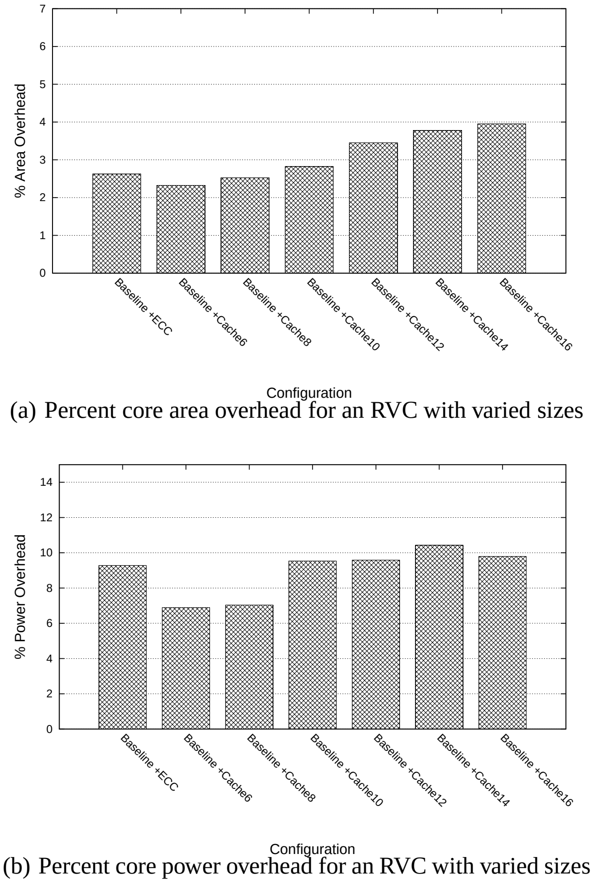
**10:** Register value cache hit rates as a function of the number of entries in the cache

In order to evaluate the cost of implementing various RVC designs in comparison to traditional fault tolerance techniques, we augmented the ARM926EJ-S register file with ECC protection. We implemented ECC protection circuits using minimum odd-weight-column SEC-DED codes as described in [7]. The ECC protected register file required two ECC encode units, one for each of the potential write operations, and three ECC decode and error correction units, one for each of the potential read operations. We also implemented the RVC with a variety of sizes ranging from 6 to 16 entries at intervals of two. Area results for the baseline ARM926EJ-S ECC protected register file and each of the RVC designs were generated by the Synopsys Physical Compiler, and power numbers were generated by the Synopsys Power Compiler. The area results for each of the configurations are shown in Figure 11(a) and the power results are shown in Figure 11(b).

The data presented in Figure 11 demonstrates the percent area and power overhead for the ARM926EJ-S core for each fault tolerant register file configuration. These results indicate that an RVC of eight entries or less will consume both less on-chip area and power than a register file augmented with ECC. In general, the area requirements for the RVC tend to scale regularly, while the power requirements tended to be more erratic. This was largely due to synthesis optimizations which could utilize more efficient logic for cache sizes which were a power of two.

The fault coverage provided by protecting the register file is bounded by the fact that only 57.4% of the faults that were visible at the software interface occurred within the register file. This coverage bound is further reduced for ECC implementations because ECC can only detect and correct faults occurring in the actual register state array. Only 44.1% of the register file is sequential state, and so we can conservatively estimate the fault coverage of the ECC protection to be (57.4% * 44.1%) = 25.31%. This estimate is conservative because a modest fraction of the sequential state elements within the register file is microarchitectural state rather than architected registers. They would not be protected by ECC and could further degrade the fault coverage provided by ECC.

Since the RVC is capable of protecting against both faults occurring in combinational logic and sequential state elements, the fault coverage for the RVC design is bounded only by the cache hit rate multiplied by the coverage gained by protecting the register file. We assume here that the probability of multiple concurrent



(a) Percent core area overhead for an RVC with varied sizes

(b) Percent core power overhead for an RVC with varied sizes

**11:** Area and power analysis of register value cache implementations

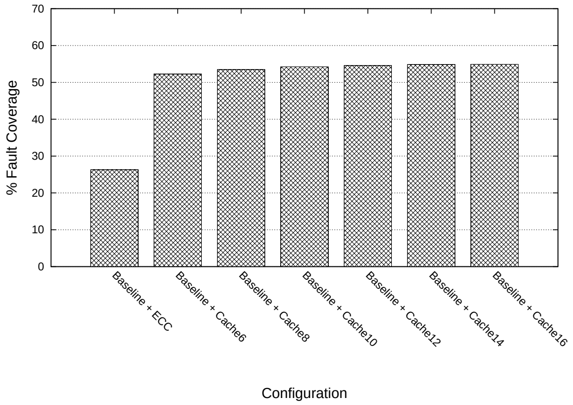
particle strikes is negligible. Figure 12 presents the achievable fault coverage for various fault tolerant register file configurations.

Figure 12 demonstrates that an RVC, with as few as eight entries, is capable of delivering more than twice the fault coverage of an ECC protected register file. Even more importantly, the RVC can be scaled to 14 entries, increasing the fault coverage to 99% of the faults affecting the register file, while increasing the area and power overhead by only about 1% over the cost of implementing ECC.

## 4.2 Time-delayed Shadow Latch Analysis

In this section, we analyze the area and power requirements necessary for adding sufficient detectors to achieve a given amount of fault coverage. A large number of Monte Carlo fault simulations were conducted to identify state elements for potential augmentation with time-delayed shadow latches. In these experiments, a counter is maintained for each state element within the design. The counter is incremented every time the state element is corrupted within the first cycle following a fault injection. As a result, the counter associated with each state element reflects the number of distinct faults that would be detected by augmenting that state element with a pulse detection unit. The set of state elements is then rank-ordered using these counter values. Two experiments, provided with this prioritized list of candidate state elements, are performed to determine the amount of attainable fault coverage.

The first experiment presents a limit study that demonstrates an upper bound on the number of faults covered as each state element is protected with a shadow latch. In the second experiment, the fault injection data is divided into training data and test data. The training set is used to generate the candidate list described previously and the test set is used to evaluate the fault coverage achieved when deploying the detectors accordingly. For the second experiment we present the average achieved coverage over several thousand trials. We refer to the results from these two experiments as the coverage limit study and the observed coverage respectively.



**12:** Fault coverage for a variety of register file configurations

In Figure 13, we present a study of the amount of area and power overhead incurred by augmenting flip-flops with the time-delayed shadow latches described in Section 3.2. As mentioned in Section 3.2, an automated technique for inserting these detectors into a microprocessor core is the subject of future work, and here we present only a manual technique. The overhead for inserting these detectors is presented in terms of the additional logic cell overhead alone, however, preliminary experiments show that interconnect costs will not significantly impact the results presented.

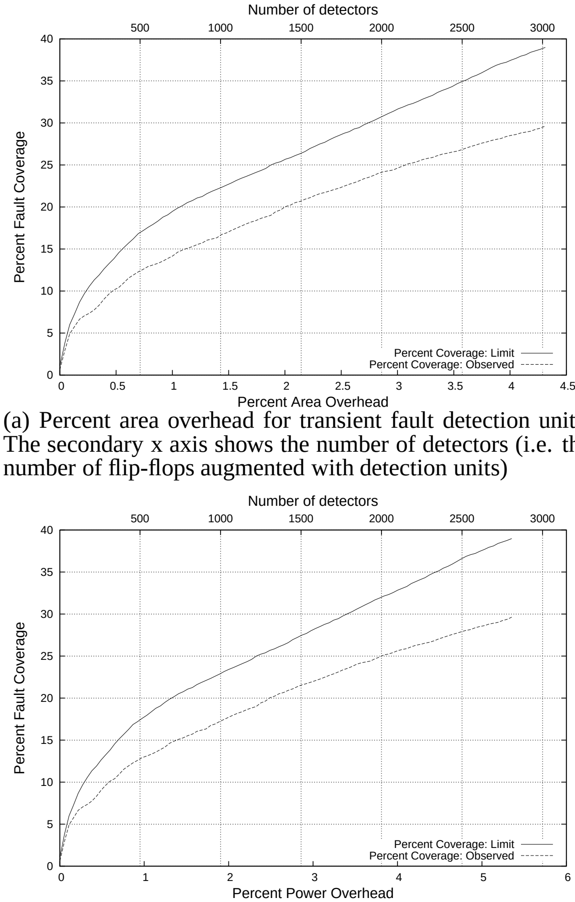
Figure 13 demonstrates two sets of data, one for the coverage limit study and one for the observed coverage as described above. The limit study demonstrates that in the best case, less than 30% of the registers in the design would need to be augmented in order to achieve 99.9% fault coverage for faults occurring outside of the register file. For the observed coverage metric in Figure 13, our results show that on average, 90% fault coverage could be acheived by augmenting approximately 25% of the state elements within the design.

## 4.3 Combined Approach Analysis

The techniques proposed in this paper work together in a cooperative fashion to address disjoint sets of faults. Here, we present an analysis of the combined fault coverage that can be achieved by employing both techniques together to tolerate soft errors. The area and power numbers shown here represent the combined cost of implementing the two techniques, in terms of the logic cell overhead alone.

Figure 14(a) demonstrates the combined cell area overhead and Figure 14(b) demonstrates the combined total power overhead for each technique discussed in this paper. Each line in Figures 14(a) and 14(b) represents a different register file configuration and its effectiveness when used in conjunction with the amount of area and power budgeted for the time-delayed shadow latches described in 3.2. For each graph, fault coverage increases with the percent overhead as more transient pulse detectors are added to the design.

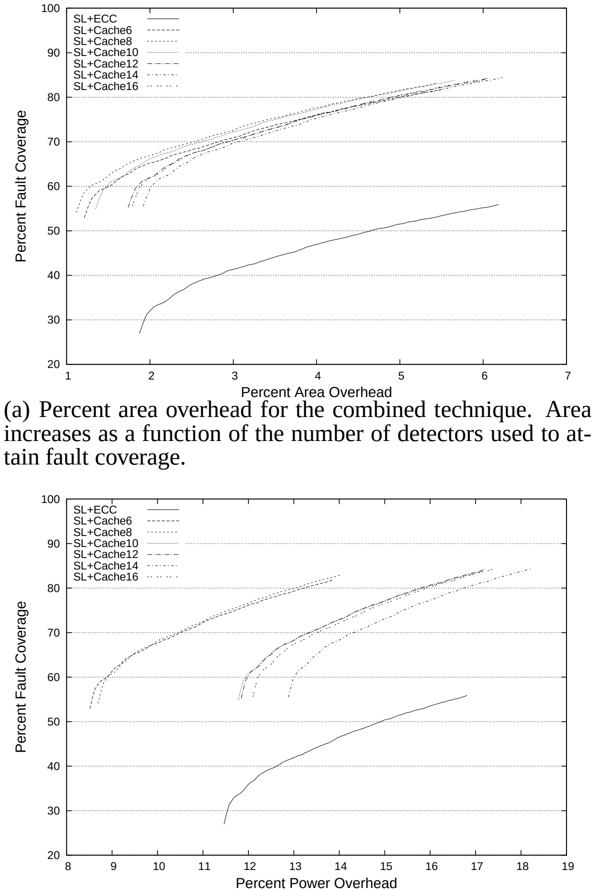
Several observations can be made from this data. First, it is clear that ECC is far less effective than the RVC in protecting the register file from corruption. Second, the increase in area overhead for different configurations of RVC is almost negligible, while the power overhead increase is slightly more dramatic. However, the majority of the power overhead comes from the addition of more



(a) Percent area overhead for transient fault detection units. The secondary x axis shows the number of detectors (i.e. the number of flip-flops augmented with detection units)

(b) Percent power overhead for transient fault detection units. The secondary x axis shows the number of detectors (i.e. the number of flip-flops augmented with detection units)

**13:** Area and power analysis for transient fault detection units



(a) Percent area overhead for the combined technique. Area increases as a function of the number of detectors used to attain fault coverage.

(b) Percent power overhead for the combined technique. Power increases as a function of the number of detectors used to attain fault coverage.

**14:** Area and power analysis for the combined technique. SL represents the *time delayed redundant latch* technique, ECC represents the *error correcting codes* technique for register file protection and CacheX represents the *register value cache* technique with X number of cache entries.

time-delayed shadow latches, which tend to yield diminishing returns. Lastly, more than 80% fault coverage can be achieved at a cost of less than 14% power overhead and less than 6% in area overhead.

# 5. RELATED WORK

Kim and Somani [8] conducted software-simulated fault injection campaigns on an RTL model of the PicoJava-II microprocessor to determine the *soft error sensitivity* of logic blocks within the design. The soft error sensitivity (SES) metric used in this work is defined as the probability that a soft error within a given logic block will cause the processor to enter an incorrect architectural state. The fault model used in this work is similar to our own, though the authors of this paper conduct error analysis strictly at the architectural level.

In [11], Mukherjee et al. define the term *architectural vulnerability factor* (AVF) to be the probability that a fault in a microarchitectural structure will cause an error in program output. The authors use a performance simulator of the Itanium II microarchitecture to determine the AVFs for structures within their simulated microarchitecture. Our work presents similar results at the architectural level for faults injected into sequential state , but focuses on the microarchitectural effects of soft errors on a less aggressive processor core.

Wang, et al. [18] characterize the effects of soft errors on an out-of-order, superscalar Alpha-like processor core. The fault model used in this work simulates single bit flips in sequential state elements within the design, and an analysis of the failure modes exhibited in simulation is described. In their work, the authors explore the effects of soft errors on a substantially different microarchitectural model and propose techniques for detecting soft errors based on *symptoms* observed at the software interface.

Saggese, et al. [15] present a similar analysis of the effects of soft errors occurring in both sequential state elements and combinational logic on a DLX microprocessor model. The error manifestation rates demonstrated in their work are corroborated by our own, however, in our work we have chosen to focus on the error propagation behavior exhibited at the microarchitectural level rather than a sensitivity analysis of different blocks within the design.

In [10], the authors present a technique for protecting register files in high-performance architectures from faults that may occur when the clock frequency is aggressively scaled. The authors specifically target systems with large physical register files typically found in superscalar pipelines and store redundant live register values in unused physical registers. Though the idea proposed here is similar in nature to our own, it only feasible for systems with large, underutilized register files, and provides no coverage when the register file is fully utilized.

In this work, we leverage a large body of research [5] [6] [12] focused on circuits for detecting delay faults caused by electrical noise, particle strikes and inadequate voltage levels. This work provides the basis for the proposed strategic placement of transient fault detectors. We exploit the circuit-level characteristics of embedded microprocessors in order to efficiently utilize this technology.

## 6. CONCLUSION

This work presents a thorough analysis of soft errors on an ARM926EJ-S core. This analysis was done in order to motivate low-overhead soft error tolerance mechanisms appropriate for the embedded design space. We demonstrate how soft errors in combinational logic affect the behavior of soft errors at the microarchitectural level, and why this is important in the embedded domain.

For mitigating soft errors, we present two low-overhead complementary techniques that provide scalable fault coverage as a function of the available area and power budgets. In the first technique, we introduce the register value cache, an architectural mechanism, that provides twice the fault coverage of ECC when applied to the register file and costs less to implement in terms of both area and power. The second technique that we present makes use of time-delayed shadow latches for fault detection. It identifies high fan-in nodes in the microprocessor core for placing these detectors and achieves up to 40% fault coverage. In conjunction, the two proposed fault tolerance techniques can provide approximately 84% fault coverage while incurring less than 5.5% area overhead and about 14% power overhead.

## 7. REFERENCES

[1] ARM Ltd. *ARM926EJ-S Technical Reference Manual*, Jan. 2004. http://www.arm.com/pdfs/DDI0198D_926_TRM.pdf.

[2] T. Austin, D. Blaauw, T. Mudge, and K. Flautner. Making typical silicon matter with razor. *IEEE Computer*, 37(3):57–65, Mar. 2004.

[3] T. Austin, E. Larson, and D. Ernst. Simplescalar: An infrastructure for computer system modeling. *IEEE Transactions on Computers*, 35(2):59–67, Feb. 2002.

[4] D. Bernick, B. Bruckert, P. D. Vigna, D. Garcia, R. Jardine, J. Klecka, and J. Smullen. Nonstop Advanced Architecture. In *International Conference on Dependable Systems and Networks*, pages 12–21, June 2005.

[5] S. Das, D. Roberts, S. Lee, S. Pant, D. Blaauw, T. Austin, K. Flautner, and T. Mudge. A self-tuning dvs processor using delay-error detection and correction. *IEEE Journal of Solid-State Circuits*, 41(4):792–804, 2006.

[6] P. Franco and E. J. McCluskey. On-line delay testing of digital circuits. In *Proc. of the 1994 IEEE VLSI Test Symposium*, pages 167–173, 1994.

[7] M. Y. Hsiao. A class of optimal minimum odd-weight-column sec-ded codes. *IBM Journal of Research and Development*, 14(4):395–401, 1970.

[8] S. Kim and A. Somani. Soft error sensitivity characterization for microprocessor dependability enhancement strategy. In *International Conference on Dependable Systems and Networks*, pages 416–428, June 2002.

[9] C. Lee, M. Potkonjak, and W. Mangione-Smith. MediaBench: A tool for evaluating and synthesizing multimedia and communications systems. In *Proc. of the 30th Annual International Symposium on Microarchitecture*, pages 330–335, 1997.

[10] G. Memik, M. Chowdhury, A. Mallik, and Y. Ismail. Engineering over-clocking: Reliability-performance trade-offs for high-performance register files. In *International Conference on Dependable Systems and Networks*, pages 770–779, June 2005.

[11] S. S. Mukherjee, C. Weaver, J. Emer, S. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high performance microprocessor. In *International Symposium on Microarchitecture*, pages 29–42, Dec. 2003.

[12] M. Nicolaidis. Time redundancy based soft-error tolerance to rescue nanometer technologies. In *Proc. of the 1999 IEEE VLSI Test Symposium*, pages 86–94, 1999.

[13] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simulataneous multithreading. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, June 2000.

[14] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *International Symposium on Fault Tolerant Computing*, pages 84–91, 1999.

[15] G. P. Saggese, A. Vetteth, Z. Kalbarczyk, and R. Iyer. Microprocessor sensitivity to failures: Control vs. execution and combinatorial vs. sequential logic. In *International Conference on Dependable Systems and Networks*, pages 760–769, June 2005.

[16] D. Seal. *ARM Architecture Reference Manual*. Addison-Wesley, London, UK, 2000.

[17] L. Spainhower and T. Gregg. IBM S/390 Parallel Enterprise Server G5 Fault Tolerance: A Historical Perspective. *IBM Journal of Research and Development*, 43(6):863–873, 1999.

[18] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline. In *International Conference on Dependable Systems and Networks*, page 61, June 2004.

[19] J. Zeigler. Terrestrial cosmic ray intensities. *IBM Journal of Research and Development*, 42(1):117–139, 1998.