Characterizing the Effects of Transient Faults on a High-Performance Processor Pipeline

Nicholas J. Wang Justin Quek Todd M. Rafacz Sanjay J. Patel
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

Abstract

The progression of implementation technologies into the sub-100 nanometer lithographies renew the importance of understanding and protecting against single-event upsets in digital systems. In this work, the effects of transient faults on high performance microprocessors is explored. To perform a thorough exploration, a highly detailed register transfer level model of a deeply pipelined, out-of-order microprocessor was created. Using fault injection, we determined that fewer than 15% of single bit corruptions in processor state result in software visible errors. These failures were analyzed to identify the most vulnerable portions of the processor, which were then protected using simple low-overhead techniques. This resulted in a 75% reduction in failures. Building upon the failure modes seen in the microarchitecture, fault injections into software were performed to investigate the level of masking that the software layer provides. Together, the baseline microarchitectural substrate and software mask more than 9 out of 10 transient faults from affecting correct program execution.

1. Introduction

Among the various issues facing the scaling of implementation technologies into the deep submicron regime, the issue of transient faults remains largely an unknown entity. Transient faults can arise from multiple sources: external sources such as high-energy particles that cause voltage pulses in digital circuits, as well as internal sources that include coupling, leakage, power supply noise, and temporal circuit variations.

While transient faults have always to some extent plagued semiconductor-based digital systems, the scaling of devices, operating voltages, and design margins for purposes of performance and functionality raises concerns about the susceptibility of future-generation systems to such transient effects. Historically, transient faults were of concern for those designing high-availability systems or systems used in electronics-hostile environments such as outer space. Because of the confluence of device and voltage scaling, and the increasing complexity of digital systems, the problem of transient faults is forecast to be a problem for all future digital systems. From high-energy neutrons alone, experts estimate that Failures in Time (FITs) for a chip will increase with the number of devices (i.e., with Moore's Law).

One major question is what should be done to protect the unstructured control logic that exists within a modern processor pipeline? The relative amount of chip area devoted to such general logic is increasing with chip complexity, and therefore the effects of transient faults through combinational logic networks and pipeline latches is of particular concern. Relatively straightforward techniques exist to protect large RAM structures from infrequent, localized transient events while few, and mostly ad-hoc, techniques exist for protecting the instruction processing pipeline of a modern high-performance microprocessor.

In this paper, this question is approached by examining the effect of transient faults on a modern microprocessor similar to the Alpha 21264 or AMD Athlon through fault injection on a detailed Verilog model. The degree of *fault masking*, or the rates at which transient faults are masked from appearing as software visible errors, is estimated, and vulnerable portions of the processor are identified. Based on this assessment, we derive some lightweight mechanisms to harden these structures, significantly improving the resilience of the pipeline to soft errors.

In this work, we make three basic contributions:

- **Microarchitectural Effects of Transient Faults:** We study the effects of transient faults that propagate into pipeline state (such as a latch or RAM cell) and thus become an error at the microarchitectural level. The purpose of this component of our work is to examine the level and types of fault masking that occur when a transient fault manifests as a latched error in the pipeline logic of a modern processor. This study is conducted on a *latch-accurate* Verilog model of a modern wide-issue Alpha processor that uses speculative execution. This particular contribution is similar to previous work [6, 12]; here a more intensive fault injection campaign is performed on a substantially more complex and speculative processor. This component is also a continuation of work that examined the fault propagation into a latch [3, 11, 16, 17, 20].

Here the fault propagation out from the latch is examined.

• **Lightweight Microarchitectural Protection Mechanisms:** Using the data gathered from our first set of fault injection campaigns, we identify vulnerable components within the processor pipeline and devise low-overhead protection mechanisms to increase the microarchitectural masking level. These mechanisms result in a sizable reduction of failures without resorting to the use of wholesale redundancy or an architectural checker [23].

• **Architectural Effects of Microarchitectural Errors:** The effects of latch-level errors that have propagated into architectural processor state (i.e., register file and instruction words) are studied. Using simplistic fault models derived from our study of microarchitectural faults, fault masking in software is observed and characterized.

2. Experimental Methodology

In this section, we describe our experimental methodology. First, we introduce the processor microarchitecture and Verilog model used in our experimentation. Next, we describe our fault model and fault injection framework. Finally, we discuss the statistical significance of the results presented in the remainder of the paper.

2.1. Processor Model

Given that our objective is to examine the effects of transient faults on a modern high-performance processor pipeline, we needed to develop a sufficiently detailed model of a representative microprocessor architecture (microarchitecture). In this subsection, we describe the microarchitecture and the Verilog model used in our experimentation.

Our microarchitecture is a superscalar, dynamically-scheduled pipeline similar in complexity to the Alpha 21264 [1] and the AMD Athlon [14]. The processor executes a subset of the Alpha instruction set—due to time considerations, floating point instructions, synchronizing memory operations, and some miscellaneous instructions were not implemented. The processor includes such features as speculative instruction scheduling, memory dependence prediction, and sophisticated branch prediction, which are necessary ingredients for high-performance processing. The processor can have up to 132 instructions in-flight in the 12-stage pipeline. Every cycle, up to 6 instructions are selected for execution using a dynamic scheduler of 32 entries. A diagram of the processor is shown in Figure 1 and more details are listed in Figure 2. The important point to note is that our microarchitecture is representative of current-generation high-performance microprocessors; it contains a similar rich set of performance enhancing features (e.g., speculation) that can affect the ways in which the processor reacts to transient faults.

For us, understanding the ways in which transient faults affect a microarchitecture of this complexity requires building a model of the processor that is representative down to the latch-level of a real chip implementation. That is, all state elements (latches, bits of RAM, etc) present in a real implementation are also present in the model and vice-versa. We selected an edge-triggered clocking methodology, so all of our pipeline latches are edge-triggered devices.

We argue that without such a *latch-accurate* model, it is not possible to model all fault situations, making it difficult to evaluate fault masking or to assess coverage of a protection scheme. For this reason, great care was taken to create a detailed and accurate Verilog model upon which to perform these fault injection studies.

Note that in our model, an L1 miss takes a constant eight cycles to service. This has the effect of removing longer periods of processor idleness that would result from L2 cache miss delays. As a result, our pipeline is more sensitive to transient errors, causing us to underestimate the level of masking in the pipeline.

2.2. Fault Model

Our fault model is a single bit flip of a state element. This fault model captures the state-inverting phenomenon of a neutron-strike to a state-keeping transistor of a latch or RAM cell. This model does not accurately represent faults that occur within combinational networks. However, since combinational networks have much lower sensitivities due to pulse attenuation, logical masking, latching-window masking, and capacitive loading, they are not as problematic as state elements.

Our experimentation consists of a set of trials, each consisting of a fault injection and determination of outcome. In each trial, the time at which to inject a transient fault is first selected. Then the bit to corrupt is selected randomly across all of the eligible state of the processor, where eligible state is defined by the particular experiment being run. The processor model (including caches and predictor tables) was allowed to "warm-up" prior to each fault injection.

In our experiments, we divided our fault injection campaigns into two varieties: those targeting both latches and pipeline RAM arrays and those targeting only latches. Isolating latches from all of pipeline state has significance on several fronts: First, latches may have different fault rates and fault models from RAM structures due to implementation differences [17]. By distinguishing between these types of state in our experiments, we can derive separate results for these different structures. Second, data stored in latches might have different characteristics compared to data stored in RAM type structures. For example, latches might store data that are more transient in nature or perhaps are less vulnerable to transient faults. Third, data stored in RAMs may be easier and more efficient to protect using parity or error correcting codes. Pipeline structures that are implemented using RAM arrays include the register file, RAT files, register free lists, scheduler and ROB payloads, and various queues. There are about 14,000 bits of storage in latches and 31,000

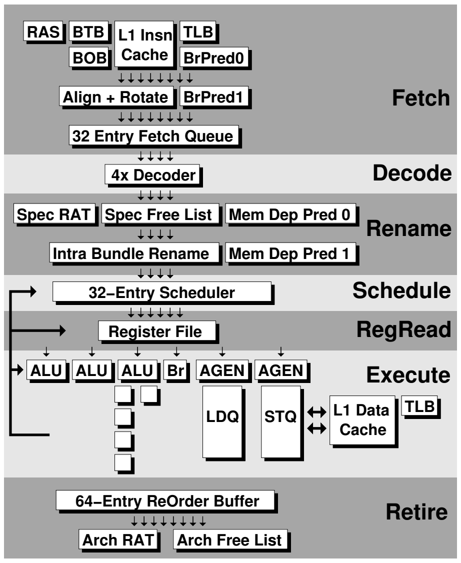

Figure 1. Processor model diagram.

Stage	Features
Fetch	1024 entry 4-way set-associative with bimodal branch predictor Hybrid branch predictor: bimodal, local, and global predictors [13] 8-entry return address stack with pointer recovery 8-wide split-line fetch from a 2-way set-associative 8kB L1 cache 32-entry fetch queue
Decode	4-wide decode
Rename	4-wide rename from 80 physical registers Speculative and architectural rename maps maintained
Issue	32-entry scheduler w/ speculative wakeup and instruction replay [8]
Reg Read	80 65-bit physical register file with 11 read ports and 7 write ports
Execute	2 simple ALUs 1 complex ALU (2-5 cycles) with buffer for register file port conflicts 1 branch ALU 2 address generation units for memory instructions
Memory	16-entry load and store queues 2-cycle, dual-ported 2-way set-associative 32kB L1 dcache Dual porting achieved with eight interleaved banks 16 non-coalescing miss handling registers for lockup free accesses Memory dependence prediction using store sets [5]
Retire	64-entry reorder buffer with 8-wide retire

Figure 2. Processor model details.

bits of storage in RAM arrays in our pipeline across which we perform injection.

After the fault injection occurs, the trial is continually monitored for up to 10,000 cycles and compared against a non-injected golden execution of the latch-level Verilog model. Each trial results in one of four outcomes: (1) *μArch Match* - microarchitectural state match, (2) *Termination* - premature termination of the workload, (3) *SDC* - silent data corruption, or (4) *Gray Area* - none of the above. These outcomes are described in the following paragraphs.

Microarchitectural state match occurs when the ENTIRE microarchitectural state of the processor model (i.e., every bit of state in the machine) is equivalent to that of a non-fault-injected simulation. If a trial results in a microarchitectural state match with no previous architectural state inconsistencies, we can conclusively declare that the injected transient fault's effects have been masked by the microarchitectural layer. These trials are placed in the *μArch Match* category.

Architectural state (i.e., program-visible state such as memory, registers, and program counter) is verified every cycle. If the architectural state comparison fails, then the transient fault has corrupted architectural state, and the trial is considered a failure (*Terminated* or *SDC*). Trials that result in register and memory corruptions are placed into the *SDC* category, along with those that result in TLB misses. Trials in the *Terminated* category are those trials that resulted in pipeline deadlock or resulted in an instruction generating an exception, such as memory alignment errors and arithmetic overflow[1].

If a trial does not result in failure or *μArch Match* within our 10,000 cycle simulation limit, the trial is placed into the *Gray Area* category. Either the fault is latent within the pipeline, or it was successfully masked, but the timing of the simulation was thrown off such that a complete microarchitectural state match was never detected. Of those that are latent, some will eventually affect architectural state while others have propagated to portions of the processor where they will never affect correct execution.

2.3. Statistical Significance

In this study, statistical sampling was used to identify trends in the effects of transient faults, so enough samples must be taken such that the experimental results have statistical significance. Ideally, both the cycle in which the fault injection occurs and the state bit that is affected would be selected uniformly. While uniform sampling was implemented for selecting the bit to corrupt, the fault injections were performed on a set of about 250–300 start points for each experiment. This methodology skews our results toward those of the individual start points. However, with a relatively large number of start points, the skewing effect is minimal.

Each experiment's results are the compilation of 25,000–30,000 trials. If the faults could be injected at any randomly selected clock cycle, the overall results would have a confidence interval of less than 0.7% at a 95% confidence level. Note that for many of the experiments, the aggregate results are subdivided for analysis, yielding larger confidence intervals. As an extreme example, the *qctrl* results in Figure 9 consisted of only approximately 100 trials. This yields a confidence interval of about 10%, the largest of the data presented in this work.

3. Injection Experiment Results

In this section, we present the results of our fault injection campaigns. Our results are partitioned into three sub-

[1]Technically, some fraction of TLB misses would result in Termination, specifically if the errant execution accesses an invalid or inaccessible page of memory. We conservatively categorize all TLB misses as SDC.

sections. In the first subsection, we analyze the effects of injecting faults randomly throughout the pipeline logic. In the second subsection, we target groups of elements within the pipeline logic with similar logical function, for example, the latches and RAMs that constitute the physical register file. Finally, we examine the relationship between microarchitectural masking and pipeline utilization.

3.1. Transient Faults in Pipeline State

Using the fault injection methodology described in Section 2, we performed two fault injection campaigns: one where we injected all bits of state (latches and RAM cells) within the processor pipeline and one where we injected only latches. The objective of these experiments is to gain insight into the native level of microarchitectural masking present in a modern processor.

Before presenting the results, we must point out that we concentrate the fault injections on the irregular portions of the pipeline by excluding data cache, instruction cache, and predictor RAM arrays from the fault injection campaigns. Fault injection into cache arrays is not interesting because these structures are easily protected with parity and error correcting codes (We do, however, inject errors into the various structures that support the caches, such as miss handling registers and memory data path latches). We also exclude any prediction structures determined to have no effect on correctness (typically, prediction structures such as branch predictors only affect timing).

Figure 3 contains the results of both fault injection campaigns. Each bar in the graph represents a different benchmark application from the SPEC2000 integer benchmark suite. Furthermore, the data from fault injection into latches and RAMs are labeled with an *l+r* suffix, while data from injection into only latches are labeled with an *l* suffix.

The different benchmarks represent different workloads on the processor, which affect the masking rate of the microarchitecture. The aggregate results are presented in the rightmost bars in each graph. The benchmark *gzip* has the highest rate of instructions committed per cycle (IPC) and *bzip2* has relatively high IPC and branch prediction rates as well as the highest data cache hit rate. These factors contribute to higher failure rates, since on average, more meaningful work is in progress resulting in more vulnerable state. We quantitatively measure this effect in Section 3.3.

Examining the aggregate bars of both graphs, one can observe that approximately 85% of latch+RAM faults and about 88% of latch-based faults are successfully masked. The fraction of trials in the *Gray Area* accounts for another 3% for both experiments; these faults are likely to have been masked also, but we were not able to determine conclusively in our framework. The remaining 12% of latch+RAM trials and 9% of latch trials were known failures that were either *SDC* or *Terminated*.

To understand the intrinsic level of microarchitectural masking for our microarchitecture (between 80-90%) one must consider that for a high-performance processor, there are many instances of idle logic, dead program state, and incorrect speculation that mask the effects of a transient fault. The effect of incorrect speculation is of significance for a current processor and increases the masking rates over the 60-70% estimated for a processor from the late 1980s by Czech and Siewiorek [6].

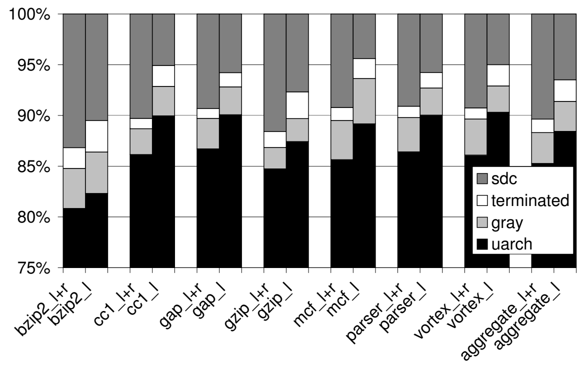

Figure 3. Fault injection results by benchmark.

3.2. Transient Faults in Logic Blocks

The next question we ask is how various logic blocks in the pipeline contribute to the failure rate of the microarchitecture. To accomplish this, each latch or RAM cell in the processor was categorized based on the general function provided by that bit of state. For example, latches and RAM cells that hold instruction input and output operands are placed into a *data* category. Table 1 lists the various categories of logic blocks and provides a brief description for each, as well as the number of bits of latches and RAM cells within that category.

The results of the fault injection campaigns (latches and latches+RAMs) were then categorized by the logic block of the bit of state that was injected and the resulting outcome of the trial. The results are presented in Figures 4 and 5.

Examining Figure 4, which presents the results for each functional block when errors are injected into latches+RAMs, one can observe that the architectural register alias table (*archrat*) and the physical register file (*regfile*) are especially vulnerable to soft errors. This is not surprising since these structures contain the software visible register file. The speculative register alias table (*specrat*) and the speculative free list (*specfreelist*) also appear to be particularly vulnerable. In order to bolster the overall reliability of our microarchitecture, it would be sensible to harden these and other structures, and we discuss some ways to do so in Section 4.

Both the latch+RAM injections and the latch-only injections show high vulnerability for the bits categorized as *qctrl* and *valid*. Their impact on the overall fail rate is small, however, since they constitute only a small fraction of the total state of the machine. Also, it is interesting to note that the fail rate of the *data* category is the lowest, due to a combination of low utilization rate, speculation, and logical masking.

Category	Description	Bits of Latches	Bits of RAMs
addr	64-bit address field for memory operations.	384	3584
archfreelist	Architectural register free list.	0	336
archrat	Architectural register alias table.	0	224
ctrl	Miscellaneous control state such as decoded instruction bundle control words and state machines.	2502	1916
data	Instruction input and output operands.	5899	2820
insn	Parts of the instruction word passed along with each instruction.	1525	2016
pc	62-bit program counter fields.	1984	12480
qctrl	Control state associated with queues.	176	0
regfile	65-bit register file entries and scoreboard bits.	80	5200
regptr	7-bit physical register file pointers.	978	1852
robptr	6-bit ROB tags.	352	444
specfreelist	Speculative register free list.	0	336
specrat	Speculative register alias table.	0	224
valid	Valid bits throughout the pipeline.	263	124

Table 1. Description of different categories of state.

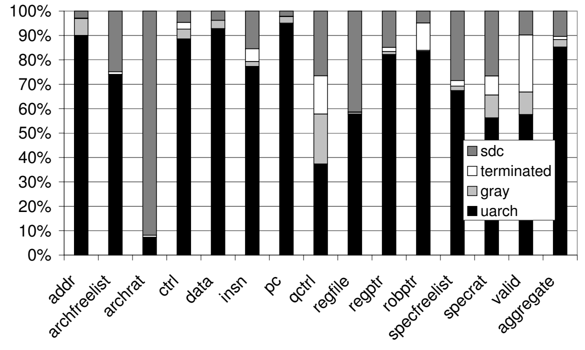

Figure 4. Results of fault injection into latches+RAMs by type.

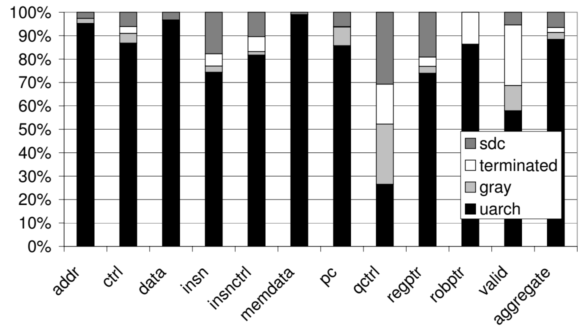

Figure 5. Results of fault injection into latches by type.

3.3. Correlation Between Utilization and Masking

We were able to extract an interesting phenomenon from the data collected from our fault injection campaigns—there is a correlation between the number of valid instructions in the pipeline and the level of microarchitectural masking. In Figure 6, a scatter-plot shows the percentage of non-failures (*Gray Area* and *μArch Match*) versus the number of valid instructions in the pipeline at the time of injection. Here, valid instructions are defined as instructions that will eventually commit their results to architected state, i.e. those that are not a result of a mis-speculation. This plot was generated for injections into latches+RAMs, and a linear least mean squared trendline is also displayed. This data is in the same vein as work done by Mukherjee et al. [21], which estimated architectural vulnerability factors for various structures based on their level of utilization.

Each data point in the scatter plot represents 100 trials from a starting checkpoint. The relatively small number of trials per data point results in a large confidence interval, contributing to noise in the graph. Nonetheless, a strong trend is present, indicating that a microprocessor is more vulnerable to transient faults when it is full of valid instructions. Interestingly, even when the pipeline is nearly full (we can theoretically have at most 132 instructions in the pipeline at any point in time), approximately 70% of all transient faults still do not propagate to architectural state, better reflecting masking levels quoted by past researchers.

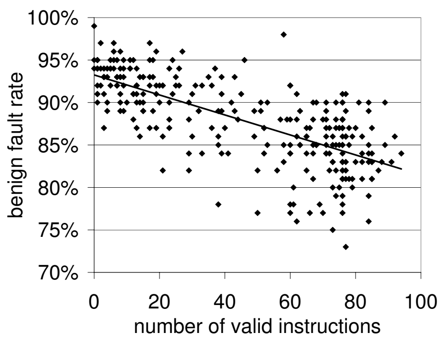

Figure 6. Scatter plot of benign fault rate versus valid instructions.

There are three explanations for this result. First of all, even when a processor is nearly filled to capacity with instructions, there is often a large portion of *dead* state not directly associated with any instruction. Examples of such state in our processor can include data path latches, the register file, and various queues that facilitate deep pipelining. Second, even some portion of processor state directly associated with valid instructions is also commonly dead. Reasons for this include structures retaining information for longer than necessary in order to support speculation (for example, our scheduler does not free an instruction's entry until it is known that the instruction will complete) and state that is not always utilized (for example, state in the memory unit that records store to load forwarding, which does not always occur). Finally, software level masking can also have a factor in this result, since we verify architectural state at cycle boundaries instead of instruction boundaries.

In summary, we observe that 85% of trials in the latch+RAMs campaign and 88% of trials in the latch-only campaign are masked. This is a significant result, particularly if one notes the fact that we are injecting approximately 50%-55% of the surface area of a modern processor die (as estimated from die photos of the Alpha 21264 and the Pentium 4). The non-injected portions include the cache RAM arrays and predictor structures, which either can be easily hardened from soft errors through redundant coding or do not contribute to failures. We also observed that the masking levels for latches is higher than that of RAM arrays, indicating that latches are generally less utilized.

4. Lightweight Protection

In this section, we develop several lightweight protection mechanisms to cover the vulnerable portions of the pipeline identified by our analysis from the previous section. We discuss the overheads of these mechanisms and evaluate their coverage with new fault injection campaigns.

4.1. Failure Modes

We begin by more deeply evaluating the 12% failure rate of the Latch+RAMs experiments from the previous section. Recall that a failure is a trial that results in a *SDC* or *Terminated* outcome. We further subdivide these failed trials by examining the manner in which the failure occurred. For example, a trial might have ended as *SDC* because the architectural register file was inconsistent with that of the golden reference model.

Table 2 lists and describes the seven failure modes. *Regfile* and *mem* failures respectively indicate that a corruption in the software visible register file or memory image was detected. A *ctrl* failure describes trials where the injected fault causes the processor to fetch, execute, and commit an incorrect (but valid) instruction. An *except* failure occurs when the processor raises an exception (e.g. memory alignment error or divide by zero). A trial that ends with a *locked* failure exhibits deadlock or livelock symptoms. In our experimentation, this is detected when 100 cycles pass without any instructions exiting the pipeline. Finally, *itlb* and *dtlb* describe transient faults that result in instruction and data translation lookaside buffer (TLB) misses. We preload both TLB's with all the pages accessed by the workload in the absence of faults, so a TLB miss in our experimentation indicates a potentially illegal memory access.

Figure 7 presents our assessment of the failure mode of each of these cases, subdivided by functional block. Figure 8 presents the relative contributions of each type of state element to the total number of failures. From these figures, we observe that the failure modes are dominated by register file inconsistencies and that a large portion of these corruptions are due to injections into the register file, register alias tables, and register free lists. Various register pointer fields throughout the pipeline also contribute to the register file corruption total. If these fields could be protected from transient faults, a large fraction of the failures would be removed.

Failure	Type	Description
ctrl	SDC	Control flow violation - incorrect insn executed
dtlb	SDC	Non-speculative access to an invalid virtual page
except	Term.	An exception was generated
itlb	SDC	Processor redirected to an invalid virtual page
locked	Term.	Deadlock or livelock detected
mem	SDC	Memory inconsistent
regfile	SDC	Register file inconsistent

Table 2. Description of failure modes.

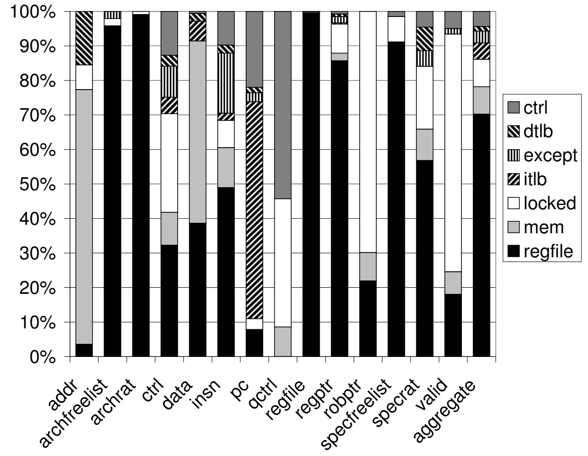

Figure 7. Breakdown of failure modes for injections into latches+RAMs.

The second leading source of failures is pipeline deadlock. Many of these failures can be attributed to corrupted *ctrl*, *qctrl*, *robptr*, and *valid* fields. In many of these cases, simply forcing a pipeline flush would reset these corrupted fields and allow the pipeline to continue executing instructions correctly. An example of a deadlock that would *not* be resolved by a pipeline flush is a corruption of a queue control field in the store buffer. Since the store buffer maintains its state across pipe flushes, another mechanism is required to resolve its deadlocks.

4.2. Protection Mechanisms

In this section, we outline four lightweight protection mechanisms that guard against the most common pipeline failures. Their implementations and overheads in terms of extra state, logic, and cycle time are discussed.

- **Timeout Counter:** The first protection mechanism is a timeout counter, which targets the *locked* pipeline failures described previously. It detects when the pipeline has not retired an instruction for a certain number of cycles (for

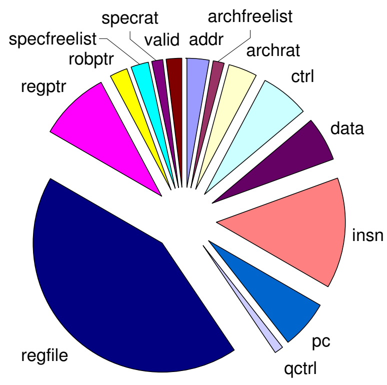

Figure 8. Relative contributions of each state type to *SDC* and *Terminated*.

our model, 100 cycles) and forces a pipeline flush in an attempt to clear any potential deadlocks. The overhead of this mechanism is estimated to be minimal in terms of both state storage and combinational logic, requiring on the order of 10 latches and an incrementer in the processor's retirement stage. Care must be taken to ensure that the counter's implementation does not cause the processor to enter livelock.

- **Register File ECC:** The next protection mechanism we chose to implement involves protecting register file contents with error correcting codes in a similar fashion to [9]. Because each register file entry potentially holds non-speculative software visible state, it is not sufficient to simply detect that an error exists if we wish to mask the transient fault. The hardware must be able to recover the data once it detects a corruption. Thus, we decided to use ECC, which added an overhead of eight bits for each of the 80 register file entries.

- **Register File Pointer ECC:** In Section 4.1, we saw that a number of structures that hold physical register file pointers contributed greatly to the number of register file corruption failures. These structures include the *archfreelist*, *archrat*, *regptr*, *specfreelist*, and *specrat* categories. For this protection mechanism, all of these structures are protected by accompanying each register file pointer with ECC. This added 4 bits of overhead to each 7 bit register file pointer. Since these pointers are simply passed from structure to structure, the generation of the ECC data only needs to occur once, at the initialization of the pipeline. Error detection and repair modules, however, are strategically placed throughout the pipeline for maximum coverage and minimum overhead.

- **Instruction Word Parity:** In our model, the instruction word (along with various decoded information) is passed along with each instruction through the pipeline to provide control information in various stages. To protect instruction words, parity bits for each 32-bit instruction word are generated as they enter the pipeline from the L1 instruction cache. As instructions flow through the pipeline and portions of their instruction words are dropped, the parity bit is updated using information from the dropped portions. When the remainder of the instruction word ceases to be propagated through the pipeline, the parity bit is checked for consistency. In the case of a parity error, a pipeline flush is initiated before the offending instruction has an opportunity to write the register file or data cache.

4.3. Overheads

In each of the implementations presented previously, the overheads in terms of extra state and logic were discussed. Another possible overhead is the impact on the clock rate of the machine. To avoid aggravating the critical path, complete fault coverage was sacrificed for ease of implementation. For example, the ECC data for the register file entries are generated a cycle after the data is written. This allows ample time for ECC generation, but leaves the data vulnerable for that first cycle. Other overheads may include higher power requirements and capacitive loads on various transistors. With the implementation of the above protection mechanisms, an extra 3061 bits of storage out of about 45K were required. Roughly two-thirds of this state storage overhead was in the form of RAM type storage, while the remainder was in the form of latches.

Depending on the nature of the various sources of transient faults, the overheads from these mechanisms likely result in a higher fault rate, due to a larger amount of vulnerable hardware. For example, a larger number of storage elements might increase the rate of faults caused by neutron strikes. Fortunately, nearly all of the introduced overheads are naturally redundant. For example, if a transient fault were to affect a parity bit protecting an instruction word, a forced pipeline flush would result with no ultimate effect on correct program behavior. Nonetheless, it is important to consider the effect of any introduced overheads.

4.4. Results

In this section, we estimate the effectiveness of the protection mechanisms described above by another fault injection campaign. For brevity, only results from injecting transient faults into latches+RAM are presented. State introduced by the protection mechanisms are also subject to fault injection. Figure 9 breaks down the results of this experiment by type of state injected. Note the addition of two new categories: *ecc* and *parity*, which respectively represent state used to store ECC and parity information.

Compared against Figure 4, the number of failed trials drops significantly. The failure rates for the *archfreelist*, *archrat*, *insn*, *regfile*, *specfreelist*, and *specrat* categories all exhibit large decreases as a result of the protection mechanisms. The set of *insn* bits, however, sees a large number of trials move from *μArch Match* to *Gray Area*. This is a result of the parity protection mechanism initiating a recovery via pipeline flush when the bit corruption would not have resulted in failure. The *Gray Area* category does not cover all

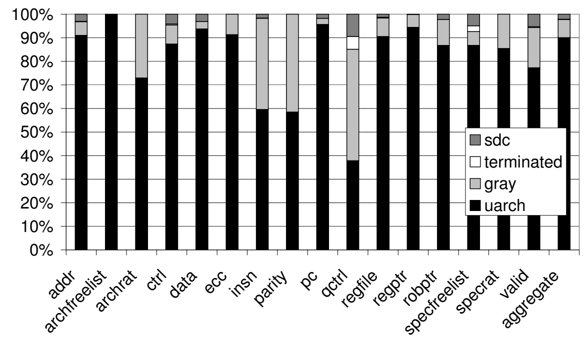

Figure 9. Results of fault injection into latches+RAMs broken down by type.

the *insn* trials, however, since only valid instruction words with incorrect parity information will trigger a recovery.

Somewhat interesting to note is the large *Gray Area* category of the *archrat* state elements. In the protection mechanism implementation, a corrupted architectural register alias table entry is never repaired, only overwritten with new, hopefully non-corrupted data. This only occurs when an instruction that writes its result to the corresponding register file entry commits. Many of the trials in the *archrat*'s *Gray Area* category are due to an injection into a register alias table entry whose corresponding architectural register is not written to within the simulation limit.

The *Gray Area* categories of the *ctrl*, *qctrl*, *robptr* and *valid* state classifications also increase in size, displacing *locked* failures. This is evidence that the timeout counter mechanism worked to flush and restart the pipeline, resulting in subsequent correct execution. Unfortunately, the change in timing due to the pipeline flush makes a complete state match unlikely, pushing many trials into *Gray Area*.

In Figure 10, a pie chart depicting the relative contributions of each state type to failures is presented. This figure is in contrast to Figure 8, from the unprotected experiment. The failures are now dominated by transient faults affecting the *pc*, *ctrl*, and *data* categories. Note that failures from the protected elements were not completely eliminated. These failures were the result of transient faults affecting areas that were left unprotected for minimal cycle time impact.

Worth noting is that directly comparing the aggregate total in Figure 9 to its counterpart in Figure 4 is not fair. This is due to the 6-7% extra (mostly non-vulnerable) state introduced by the various protection mechanisms. After accounting for a 7% higher transient fault rate, the implemented mechanisms reduce the known failure rate (represented by the *SDC* and *Terminated* categories) by approximately 75%.

5. Architectural Implications

Soft errors that do not get masked in the microarchitectural level propagate to the architectural level and become visible to the running application. However, masking continues to occur, and some fraction of these errors are masked at the

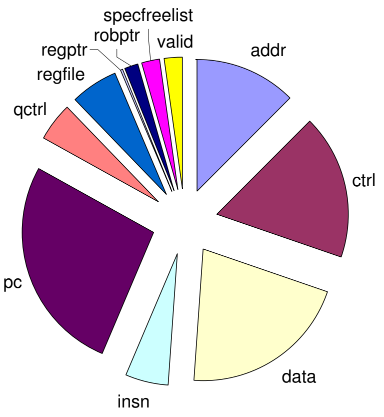

Figure 10. Relative contributions of each state type to *SDC* and *Terminated*.

architectural (or application) level. In this section, we model errors that have propagated to the architectural level and observe their effects.

For this set of fault injections campaigns, we use a modified version of SimpleScalar's functional simulator [4]. An instruction from the dynamic instruction stream is selected at random and forced to execute incorrectly. The program is then allowed to proceed, and the simulation is monitored for one of four outcomes: (1) Exception, (2) State OK, (3) Output OK, and (4) Output Bad. If the error-injected program generates an exception, it is placed in the *Exception* category. This is a "noisy" failure. Otherwise, if the architectural state (memory, registers, program counter) completely matches that of a non-error-injected execution of the program prior to a system call (the form of external communication for our applications), the trial is placed in the *State OK* category. This category represents trials that resulted in software masking of faults. If the trial does not fit in either of the first two categories, the user visible output of the application may still be correct. To identify when this occurs, the output of the application is compared against that of a reference simulation. If the program outputs were identical, the trial is placed in *Output OK*. Note that the *Output OK* category is weaker than the *State OK* category. Finally, a trial that generates incorrect user visible output is added the *Output Bad* category.

We use six different fault models in this experiment: (1) a single bit flip targeting the lower 32 bits of the result of a register write, (2) a single bit flip targeting all 64 bits, (3) replacing the result of a register write with 64 random bits of data, (4) a single bit flip into an instruction word, (5) changing an instruction into a no operation (*nop*), and (6) forcing conditional branches to flip direction. Fault models (1)-(4) in particular reflect the failure modes seen from the microarchitectural fault injection experiments from Section 3, while fault models (5) and (6) provide an additional sense of the transient fault masking levels of software. Results of these experiments are presented in Figure 11 as averages across 10 SPEC2000 integer benchmarks. They represent approx-

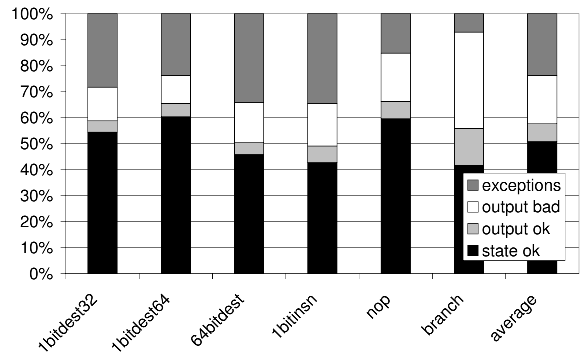

Figure 11. Results of various fault models on software.

imately 10,000-15,000 trials each, yielding a confidence interval of less than 1% at a 95% confidence level for each fault injection campaign.

From the results, we see that across all the injection campaigns, approximately half of the trials result in complete architectural state convergence (*State OK*). This indicates that the masking levels of software are significant, and roughly indicates that half the errors that escape the hardware layer are eventually masked by the application. This masking effect is largely due to dead and transitively dead values in the instruction stream.

We also note that in the first five fault models, a moderately sized portion (10–20%) of the trials from *State OK* had divergent control flow when compared against a reference execution. This means that the induced fault temporarily impacted the control flow of the application before the fault was completely masked. A fault model that only affected control flow was used in the last experiment, and we further investigated this phenomenon in [22].

6. Limitations of Results

The presented experimental results are heavily based on our choice of fault models, microarchitecture, simulation models, and workloads. For example, much of this work is geared towards characterizing the effects of single bit corruptions. If this fault model fails to accurately model physical transient faults, an underlying assumption of this work is broken. The same is true of our microarchitectural model: we only characterize the failure rates of our particular pipeline in this paper; but we believe that there are lessons to be learned that are more broadly applicable. For example, the general methodology of identifying vulnerable portions of a microprocessor and devising low overhead protection mechanisms for those portions is a generally applicable technique.

Furthermore, implementation choices we made in the microarchitectural and logic design process may affect the measured masking levels. There were occasions where we chose a simpler implementation over a more complex and compact implementation. For example, some Program Counter (PC) fields within each Reorder Buffer entry could have been stored more efficiently within a smaller separate structure, potentially reducing the number of bits in the Reorder Buffer and potentially reducing the masking rate. The extent to which this has an affect on our results is unclear, but we suspect it to be fairly small. These sorts of tradeoffs are also made on real implementations, and some real decisions might also increase masking rates.

While care was taken to create a detailed microarchitectural experimental infrastructure, not all of the intricacies of a modern dynamically scheduled processor were fully modeled. Nonetheless, we believe that our model was created with sufficient detail to provide error manifestation results accurate to within 10s of percent when compared with those of a real implementation.

7. Related Work

Czeck and Siewiorek [6] performed a similar analysis through fault injection into selected bits of state in their simulation model. Here, we use a more modern simulation model and do a more thorough classification of the failure modes of various types of state in a microprocessor.

Mukherjee et al. [21] introduced a method to compute Architectural Vulnerability Factors for various processor components and IA-64 software through analysis. The general experimental results presented in this work corroborate their analytic findings.

Kim and Somani [12] injected faults into picoJava-II, a microprocessor core developed by Sun Microsystems. Their microarchitectural model is more accurate than the one used in this work; however, it is less complex in terms of high-performance microarchitectural features. Also, they only verify the architectural state of the machine. Here, trials that result in a complete microarchitectural state match are identified along with architectural state failures.

Ando et al. [10] protected the data and address paths of their SPARC64 design with parity. Gaisler [9] protected the register file in his SPARC V8 implementation using a technique similar to the one used in this work. Furthermore, he protected various flip-flops by using triple modular redundancy and providing three separate clock trees. Franklin [7] noted different modes of failure throughout the pipeline, and proposed mechanisms to guard against them. Here, vulnerable state was identified through fault injection, and protection mechanisms to defend against a majority of transient faults were proposed, implemented, and tested.

Other work related to the microarchitectural work presented here include higher overhead mechanisms to protect microprocessors with various forms of redundancy in microarchitecture [15, 18, 23]. Here, arguably lower overhead approaches are proposed, albeit with lower fault coverage.

Previous work [19, 21, 2] has also explored the composition of dynamic instruction streams for dead and silent instructions. This work explores the same subject through fault injection and identifies a larger set of dynamically dead instructions. Namely, a significant portion of control instructions are dead, and thus, instructions that produce values for these control instructions are also possibly dead.

8. Conclusion

In this work, an analysis of the effects of transient faults on high performance processors was characterized. To accomplish this, a detailed microarchitectural model was created, and a fault model was selected. The results of the ensuing fault injection experiment were not particularly surprising: the most vulnerable parts of a processor are those that often hold architectural state. This information was taken into account when devising lightweight protection mechanisms to cover the majority of the failures.

To summarize our experimental findings, we found that at least 85% of injected single event upsets in our baseline microarchitecture are masked from software. We also found significant masking levels present in software for various fault models. Together, the microarchitectural and architectural levels of masking hide more than 9 out of every 10 latched transient faults from affecting correct program execution. With precisely placed low overhead protection mechanisms, the level of masking is even higher. This gives an idea of the underutilization of modern microprocessors and dynamic inefficiencies of software.

9. Acknowledgments

We thank the other members of the Advanced Computing Systems group as well as Zbigniew Kalbarczyk, Claudio Basile, and the anonymous referees for providing feedback during various stages of this work. We also thank Joel Biskie and Wojciech Magda for their respective contributions. This work was supported by the C2S2 Marco center, NSF grant EIA-0224453, and equipment donation from AMD.

References

[1] B. A. Gieseke et al. A 600MHz superscalar RISC microprocessor with out-of-order execution. In *1997 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, pages 176–178, Feb. 1997.

[2] B. Fahs et al. Performance characterization of a hardware framework for dynamic optimization. In *Proceedings of the 34th Annual International Symposium on Microarchitecture*, pages 16–27, 2001.

[3] M. Baze and S. Buchner. Attenuation of single event induced pulses in CMOS combinational logic. *IEEE Transactions on Nuclear Science*, 44(6):2217–2223, Dec. 1997.

[4] D. Burger, T. Austin, and S. Bennett. Evaluating future microprocessors: the simplescalar tool set. Technical Report 1308, University of Wisconsin - Madison Technical Report, July 1996.

[5] G. Z. Chrysos and J. S. Emer. Memory dependence prediction using store sets. In *Proceedings of the 25th Annual International Symposium on Computer Architecture*, pages 142 – 153, 1998.

[6] E. W. Czeck and D. Siewiorek. Effects of transient gate-level faults on program behavior. In *Proceedings of the 1990 International Symposium on Fault-Tolerant Computing*, pages 236–243, June 1990.

[7] M. Franklin. Incorporating fault tolerance in superscalar processors. In *Proceedings of High Performance Computing*, pages 301–306, Dec. 1996.

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

[9] J. Gaisler. A portable and fault-tolerant microprocessor based on the SPARC V8 architecture. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 409–415, Sept. 2002.

[10] H. Ando et al. A 1.3 GHz fifth generation SPARC64 microprocessor. In *Design Automation Conference*, pages 702–705, June 2003.

[11] H. Cha et al. A gate-level simulation environment for alpha-particle-induced transient faults. *IEEE Transactions on Computers*, 45(11):1248–1256, Nov. 1996.

[12] S. Kim and A. K. Somani. Soft error sensitivity characterization for microprocessor dependability enhancement strategy. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 416–425, Sept. 2002.

[13] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.

[14] D. Meyer. *AMD-K7$^{(TM)}$ Technology Presentation*. Advanced Micro Devices, Inc., Sunnyvale, CA, Oct. 1998. Microprocessor Forum presentation.

[15] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 99–110, May 2002.

[16] P. Lidén et al. On latching probability of particle induced transients in combinational networks. In *Proceedings of the 1994 International Symposium on Fault-Tolerant Computing*, pages 340–349, June 1994.

[17] P. Shivakumar et al. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–398, June 2002.

[18] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of Fault-Tolerant Computing Systems*, pages 84–91, June 1999.

[19] E. Rotenberg. Exploiting large ineffectual instruction sequences. Technical report, North Carolina State University, Nov. 1999.

[20] S. Buchner et al. Comparison of error rates in combinational and sequential logic. *IEEE Transactions on Nuclear Science*, 44(6):2209–2216, Dec. 1997.

[21] S. S. Mukherjee et al. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th International Symposium on Microarchitecture*, pages 29–40, Dec. 2003.

[22] N. Wang, M. Fertig, and S. Patel. Y-branches: When you come to a fork in the road, take it. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, pages 56–66, 2003.

[23] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 87–98, May 2002.