31st Annual International Symposium on Computer Architecture (ISCA), June 2004

Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor

Christopher Weaver[1], *Joel Emer*[1], *Shubhendu S. Mukherjee*[1], *and Steven K. Reinhardt*[1,2]

[1]Massachusetts Microprocessor Design Center
Intel Corporation
77 Reed Road, Hudson MA 01749
{christopher.t.weaver,joel.emer,shubu.mukherjee}@intel.com

[2]Advanced Computer Architecture Lab
EECS Department, University of Michigan
1301 Beal Avenue, Ann Arbor, MI 48109
stever@eecs.umich.edu

Abstract

Transient faults due to neutron and alpha particle strikes pose a significant obstacle to increasing processor transistor counts in future technologies. Although fault rates of individual transistors may not rise significantly, incorporating more transistors into a device makes that device more likely to encounter a fault. Hence, maintaining processor error rates at acceptable levels will require increasing design effort.

This paper proposes two simple approaches to reduce error rates and evaluates their application to a microprocessor instruction queue. The first technique reduces the time instructions sit in vulnerable storage structures by selectively squashing instructions when long delays are encountered. A fault is less likely to cause an error if the structure it affects does not contain valid instructions. We introduce a new metric, MITF (Mean Instructions To Failure), to capture the trade-off between performance and reliability introduced by this approach.

The second technique addresses false detected errors. In the absence of a fault detection mechanism, such errors would not have affected the final outcome of a program. For example, a fault affecting the result of a dynamically dead instruction would not change the final program output, but could still be flagged by the hardware as an error. To avoid signalling such false errors, we modify a pipeline's error detection logic to mark affected instructions and data as possibly incorrect rather than immediately signaling an error. Then, we signal an error only if we determine later that the possibly incorrect value could have affected the program's output.

1. Introduction

Single bit upsets from transient faults have emerged as a key challenge in microprocessor design. These faults arise from energetic particles—such as neutrons from cosmic rays and alpha particles from packaging material—generating electron-hole pairs as they pass through a semiconductor device. Transistor source and diffusion nodes can collect these charges. A sufficient amount of accumulated charge may invert the state of a logic device—such as an SRAM cell, a latch, or a gate—thereby introducing a logical fault into the circuit's operation [32]. Because this type of fault does not reflect a permanent failure of the device, it is termed soft or transient.

Soft errors will be an increasing burden for microprocessor designers as the number of on-chip transistors continues to grow exponentially. The raw error rate per latch or SRAM bit is projected to remain roughly constant or decrease slightly for the next several technology generations [9][11]. Thus, unless we add error protection mechanisms or use a more robust technology (such as fully-depleted SOI), a microprocessor's error rate will grow in direct proportion to the number of devices we add to a processor in each succeeding generation.

Figure 1 illustrates the possible outcomes of a single-bit fault. Outcomes labeled 1-3 indicate non-error conditions. The most insidious form of error is silent data corruption (SDC) (outcome 4), where a fault induces the system to generate erroneous outputs. To avoid SDC, designers often employ basic error detection mechanisms, such as parity. With the ability to detect a fault but not correct it, we avoid generating incorrect outputs, but cannot recover when an error occurs. In other words, simple error detection does not reduce the error rate, but does provide fail-stop behavior and thereby avoids any data corruption. We call errors in this category detected unrecoverable errors (DUE).

We subdivide DUE events according to whether the detected error would affect the final outcome of the execution. We call benign detected errors false DUE events (outcome 5 in Figure 1) and others true DUE events (outcome 6). In most situations, it is impossible for a processor to determine at the time an error is detected whether it is benign. The conservative approach is to signal all detected errors as processor failures (e.g., via a machine-check exception).

A direct approach to reducing error rates involves adding error correction or recovery mechanisms to a design, eliminat-

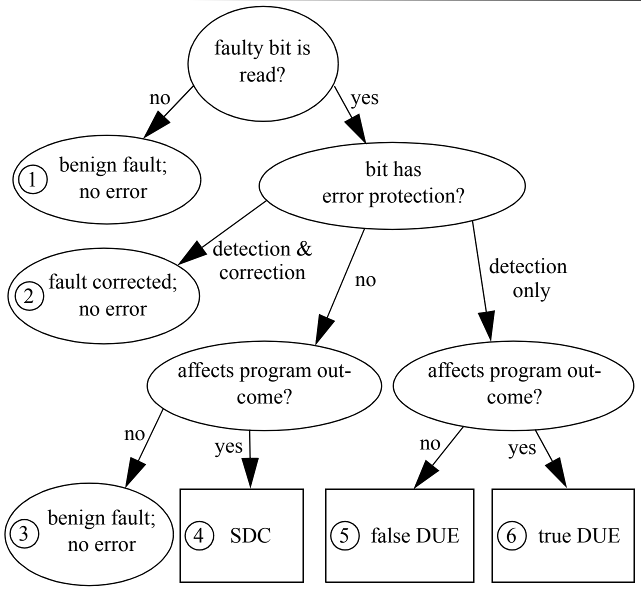

Figure 1. Classification of the possible outcomes of a faulty bit in a microprocessor. SDC = silent data corruption. DUE = detected unrecoverable error.

ing outcomes 3 through 6 from Figure 1. Unfortunately, these mechanisms come at a significant cost in power, performance, and area. Circuit-level recovery mechanisms, such as the DICE cell [6], can double the number of transistors per storage element. Error correction codes in memory have significant storage overhead and incur additional latency in generating and checking the codes. Recovery with redundant execution schemes (e.g., [2], [29], [8]) requires duplication of a thread or processor. Furthermore, these techniques may be overkill for most of the microprocessor market, which requires good reliability but not bulletproof operation.

This paper focuses on an alternative mechanism to lowering error rates: reducing the likelihood that a transient fault causes the processor to declare an error condition. We propose two variations on this theme, with specific application to a microprocessor instruction queue.

Our first approach reduces the probability that a transient fault affects valid state (e.g., an instruction) by keeping valid state out of vulnerable structures. This reduces the exposure of valid state to radiation and, hence, lowers the soft error rate of the affected structures. For example, most bits in an invalid instruction queue entry (other than the valid bit itself) will never be read; thus a fault in an invalid entry will not result in an error (outcome 1 from Figure 1). By filling the instruction queue with invalid entries instead of valid instructions during lengthy stalls, we reduce the probability of a neutron or alpha particle strike corrupting a valid instruction. In general, we use situations that presage long delays as triggers to carry out actions that reduce the exposure of valid state to potential faults. In this paper, we examine cache-miss triggers and *squashing* actions to remove existing instructions from the instruction queue. Our results with an Itanium®2-like processor running the SPEC CPU2000 show that these techniques can reduce the instruction queue's soft error rate by 18-34% for only a 2-10% decrease in overall IPC.

Because such stalling operations may reduce performance, we must determine whether the performance loss is justified for the corresponding gain in reliability. To quantify this trade-off, we introduce a new metric, Mean Instructions to Failure (MITF). We show that as long as the relative decrease in vulnerability is larger than the corresponding decrease in IPC (instructions per cycle), our technique will allow a CPU to complete more work on average before encountering an error. Our results show that instruction squashing can increase the instruction queue's MITF by 15-39%, possibly justifying the use of these techniques.

Our second approach addresses the distinction between false and true DUE events shown in Figure 1. In a microprocessor, false DUE events could arise from strikes on wrong-path instructions, falsely predicated instructions, and on correct-path instructions that do not affect the final program state, including no-ops, prefetches, and dynamically dead instructions. Mukherjee, et al. [18] and Wang, et al. [30] describe several instances of such faults that do not manifest as errors in the final output of a program. Our analysis shows that false DUE events account for as much as 52% of the total DUE rate of an instruction queue protected only with parity.

To track false DUE events, we propose attaching a bit called the π bit (*pi* for *p*ossibly *i*ncorrect) to every instruction and potentially to various hardware structures. When an error is detected, the hardware will set the π bit of the affected instruction instead of signaling the error. Later, by examining the π bit and identifying the nature of the instruction, the hardware can decide if indeed a visible error has occurred. Thus, for example, the retire unit in a pipeline can ignore the π bit of a wrong-path instruction and avoid signaling a false error. Among the different categories of instructions, identifying dynamically dead instructions is most involved. We propose several mechanisms, including a Post-commit Error Tracking (PET) buffer and the optional use of π bits on register files, various hardware structures, and in the memory system, to provide good coverage of false DUE from dynamically dead instructions. Using the above techniques, we can cover 100% of the false DUE events. We show that the combination of the exposure and false DUE reduction techniques reduce the SDC rate of an unprotected instruction queue by 26% and the DUE rate of an instruction queue protected with parity by 57%.

Overall this paper makes four contributions. First, we propose a new approach to reducing soft error rates by reducing the exposure of valid state to neutron or alpha strikes. Second, we introduce the term MITF (mean instructions between failures) to reason about the trade-off between reliability and performance. Third, we identify the existence of false DUE events. And, fourth and final, we describe several new mechanisms, such as the π bit, to avoid flagging errors on false DUE events.

The rest of the paper is organized as follows. Section 2 discusses how to compute the SDC and DUE rates of a microprocessor. Section 3 describes how to reduce the error rate of a processor by reducing the amount of time an instruction sits in a vulnerable storage structure. Section 4 describes how to reduce false DUE rate by marking instructions as possibly incorrect. Section 5 describes our methodology and Section 6 shows our results. Section 7 discusses related work. Finally, Section 8 presents our conclusions.

2. Computing the SDC and DUE Rates

This section outlines how to compute a microprocessor's SDC and DUE rates, given the raw soft error rate of the underlying circuit technology. Vendors typically specify targets for both SDC and DUE rates of a processor [4]. A chip's raw soft error rate arises from neutron and alpha particle strikes, and depends on both its circuit characteristics and the particle flux encountered. The neutron flux from cosmic radiation depends on the environment. For example, at an altitude of 1.5km—e.g., in Denver, Colorado—the neutron flux is 3 to 5 times higher than at sea level. The alpha particle flux depends on the shielding used and on the packaging material. Circuit parameters that influence the error rate include the amount of charge stored, the vulnerable cross-section area, and charge collection efficiency [24]. Raw error rates as well as SDC and DUE rates are typically expressed in FIT (Failures in Time). One FIT equals one failure in a billion hours. FIT is inversely related to MTBF (mean time between failures). An MTBF of one year equals 114,155 FIT (= 10^9 / (24 * 365)).

In this paper, we consider only errors caused by single-bit faults. Multi-bit faults—caused by a single particle strike affecting multiple cells, or by multiple strikes—can cause SDC events in structures with single-bit error detection (e.g., parity) and DUE events in structures with single-bit error correction (e.g., SECDED ECC). The probability of multi-bit faults is, however, orders of magnitude lower than that of single bit faults. Additionally, careful design, such as interleaving cells from different entries in the physical layout or scrubbing a structure periodically, can make multi-bit faults in the domain of a single parity- or ECC-protected block extremely unlikely [16]. To simplify our analysis, we assume that there is zero probability of multi-bit faults that can defeat any error detection or correction scheme.

2.1. Computing the SDC Rate

To compute the SDC rate of a processor, we use the following equation:

$$\text{SDC rate} = \sum_{\text{all devices } d} \text{error rate}_d \times \text{SDC AVF}_d$$

That is, the contribution of each device in the system to the overall SDC rate is the product of that device's raw error rate with its SDC architectural vulnerability factor, or AVF. A device's SDC AVF expresses the probability that a strike affecting that device eventually results in an error in a program's output. A device that is protected by an error detection or correction mechanism cannot cause an SDC event (assuming single-bit errors), so its SDC AVF—and its contribution to the overall SDC rate—is zero. The SDC AVF of unprotected devices varies according to their utilization. For example, an upset in a branch predictor bit will not result in a user-visible error; therefore, its SDC AVF is zero. Similarly, an upset in the program counter will most likely result in executing the wrong instructions; therefore, the SDC AVF of the program counter is practically 100%. Computing SDC AVFs for other structures, such as the instruction queue, is slightly more involved because in different processor cycles the same bit may contain information that may or may not affect the final outcome of a program. For example, an instruction queue entry containing information pertaining to a wrong-path instruction will have a zero AVF. At some other point in time, the same physical entry may contain a vital correct-path instruction, resulting in a high AVF.

Mukherjee, et al. [18] introduced the concept of architecturally correct execution (ACE) to compute the SDC AVF of such structures. Architecturally correct execution encompasses any execution that generates results consistent with the correct operation of the system as observed by a user. Individual instructions may generate incorrect results without violating ACE if those results are never observed outside the system (e.g., they are dead values). Recent work has shown that even executing the wrong instructions need not violate ACE [30].

A bit is called an ACE bit when it contains information that will affect the final outcome of the program, and called an un-ACE bit otherwise. The SDC AVF of a storage cell is the fraction of cycles that it contains an ACE bit. If a program executes for 10 million cycles and a storage cell contains an ACE bit for 1 million of those cycles (and, hence an un-ACE bit for the rest of the 9 million cycles), then the SDC AVF of that cell is 1 / 10 = 10%. The SDC AVF of a structure is the average of the SDC AVFs of all cells in that structure. Mukherjee, et al. computed an SDC AVF of 28% for an unprotected instruction queue in an Itanium® 2-like microprocessor.

2.2. Computing the DUE Rate

We compute the DUE rate of a microprocessor using an equation similar to the SDC rate:

$$\text{DUE rate} = \sum_{\text{all devices } d} \text{error rate}_d \times \text{DUE AVF}_d$$

The raw soft error rate used in the DUE equation is same as the one used for SDC. The DUE AVF is the probability that a strike will result in a detected unrecoverable error. Only devices that have error detection but not error correction (e.g., parity) will have non-zero DUE AVFs. The DUE AVF itself can be rewritten as:

$$\text{DUE AVF} = \text{true DUE AVF} + \text{false DUE AVF}$$

As the names indicate, true DUE AVF arises from true DUE events, while false DUE AVF arises from false DUE events.

Interestingly, protecting a structure with an error detection mechanism increases the overall error contribution from the structure. A fault that would have been an SDC event now becomes a true DUE event. Thus, we have:

$$\text{True DUE AVF (with error detection)}_d = \text{SDC AVF (with no error detection or correction)}_d$$

However, some faults that would have been benign because the program outcome was unaffected will now be detected, generating false DUE events. Thus the total DUE AVF of the protected structure will be at least as large as, and probably greater than, the SDC AVF of the unprotected version. Furthermore, error detection schemes generally add extra bits, e.g., parity bits; the additional area consumed by these bits will raise the raw error rate of the structure as well.

3. Reducing Exposure to Radiation

This section discusses our first approach to reducing AVF by reducing the exposure of ACE objects to neutron and alpha radiation. The basic idea is to keep ACE objects in protected memory and fetch them to vulnerable storage only when needed. If the vulnerable storage has no protection, then its error rate would contribute towards the SDC rate of the processor. If the vulnerable storage has error detection, but no recovery, then its error rate would contribute towards the DUE rate of the processor.

For example, microprocessors often aggressively fetch instructions from protected memory, such as main memory protected with ECC or a read-only instruction cache protected with parity (but recoverable because instructions can be re-fetched from main memory on a parity error). However, these instructions may stall in the instruction queue due to pipeline hazards, such as lack of functional units or cache misses. The longer such instructions reside in the instruction queue, the higher the likelihood that they will get struck by a neutron or alpha particle. There are many events, such as data cache misses in in-order pipelines, which always result in pipeline stalls. In such cases, we could squash (or remove) instructions from the instruction queue and bring them back when the pipeline resumes execution. This reduces an instruction's exposure to radiation, thereby lowering the instruction queue's SDC or DUE rate, as the case may be.

Section 3.1 discusses the mechanisms we propose to reduce exposure. These mechanisms will sometimes degrade performance; Section 3.2 provides an analysis of how to reason about the trade-off between performance and soft error rates.

3.1. Triggers and Actions

We characterize mechanisms to reduce exposure to radiation using two dimensions: triggers and actions. A trigger is an event that initiates an action to reduce exposure. In this paper, we study one trigger and two actions to reduce the instruction queue's exposure to radiation.

Our goal is to avoid having instructions sit needlessly in the instruction queue for long periods of time, so our trigger must be an event that indicates that queued instructions will face a long delay. We choose cache misses as our trigger. Specifically, the in-order Itanium® 2-like processor we use for our evaluation has three levels of caches: L0, L1, and L2—each successively bigger with increasing latency of access. In this paper we study two triggers: an L0 cache miss, whose latency is 10 cycles, and L1 cache miss, whose latency is about 25 cycles. Instructions following a cache-miss load cannot make progress while the miss is outstanding, particularly in an in-order machine such as ours. The situation is similar, though not as pronounced, for out-of-order machines in which instructions dependent on a load miss cannot make progress until the load returns data. We thus expect that removing instructions from the pipeline during the miss interval should not degrade performance significantly.

Once the processor incurs a cache miss, one possible action is to remove existing instructions from the instruction queue to be re-fetched later. Such instruction squashing attempts to keep instructions from sitting needlessly in the instruction queue for extended periods. To avoid removing instructions that could be executed before the miss completes, the instruction queue should squash only those instructions that are younger than the load that missed. In this paper, we examine one such squashing policy, similar to the one proposed by Tullsen and Brown [28]: following a load miss, squash all instructions in the instruction queue. Because we examine an in-order machine in this paper, squashing all instructions after a load miss should have minimal impact on performance. At the same time, it should lower the AVF by reducing the exposure of instructions to neutron and alpha strikes.

We also studied fetch throttling as a second action. Fetch throttling prevents new instructions from being added to the pipeline by stalling the front end of the machine. However, in our machine model, from this mechanism we did not observe significant reduction in AVF beyond what instruction squashing already provides. Hence, we do not report fetch throttling numbers in this paper.

3.2. Impact on Performance

Traditionally, the fault tolerance community has used the terms MTBF (mean time between failures) and MTTF (mean time to failure) to reason about error rates in processors and systems. These are usually expressed in years. Typically, MTTF corresponds to system uptime and is related to MTBF as follows: MTBF = MTTF + MTTR, where MTTR is the mean time to repair. Because MTTF is usually orders of magnitude greater than MTTR, people often use MTBF and MTTF synonymously. Nevertheless, MTTF is a more appropriate term for processor vendors, such as Intel or AMD, because such vendors do not have control over system-level MTTR-related features, which typically reside outside the processor chip. In this paper, we use SDC MTTF and DUE MTTF to denote the separate MTTFs from the two different error components.

While MTTF provides a metric for error rates, it does not allow us to reason about the trade-off between error rates and the performance of a processor. We introduce the concept of MITF or Mean Instructions To Failure as one approach to reason about this trade-off. MITF tells us how many instructions a processor will commit, on average, between two errors. MITF is related to MTTF as follow:

$$\begin{aligned}\text{MITF} &= \frac{\text{number of committed instructions}}{\text{number of errors encountered}}\\ &= \frac{\text{number of committed instructions}}{\frac{\text{total execution time in cycles}}{\text{frequency} \times \text{MTTF}}}\\ &= \text{IPC} \times \text{frequency} \times \text{MTTF}\end{aligned}$$

As with SDC and DUE MTTFs, we have corresponding SDC and DUE MITFs. Hence, for example, a processor running at 2 GHz with an average IPC of 2 and DUE MTTF of 10 years [4] would have a DUE MITF of 1.3×10^{18} instructions.

A higher MITF implies a greater amount of work done between errors. Assuming that, within certain bounds, increasing MITF is desirable, then we can use MITF to reason about the trade-off between performance and reliability. Since $\text{MTTF} = 1 / \langle \text{raw error rate} \times \text{AVF} \rangle$, we have:

$$\text{MITF} = \frac{\text{IPC} \times \text{frequency}}{\text{raw error rate} \times \text{AVF}} = \frac{\text{frequency}}{\text{raw error rate}} \times \frac{\text{IPC}}{\text{AVF}}$$

Thus, at a fixed frequency and raw error rate, MITF is proportional to the ratio of IPC to AVF. More specifically, SDC MITF is proportional to IPC / (SDC AVF) and DUE MITF is proportional to IPC / (DUE AVF). It can be argued that mechanisms that reduce both the AVF and the IPC, such as the one proposed in the previous section, may be worthwhile only if they increase the MITF; that is, if they increase the IPC-to-AVF ratio by reducing AVF relative to the base case to a greater degree than they reduce IPC.

Although we can use MITF to reason about performance versus AVF for incremental changes, we need to be cautious not to misapply it. For example, it could be argued that doubling processor performance while reducing the MTTF by 50% is a reasonable trade-off, as the MITF would remain constant. However, this explanation may not be adequate for customers who see their equipment fail twice as often.

4. Reducing False DUE via Tracking

False DUE events arise from detected unrecoverable errors that would not have affected the system's final output in the absence of any error detection. For example, a fault in a wrong-path instruction in the instruction queue would not affect any user-visible state. However, the processor is unlikely to know in the issue stage whether or not an instruction is on the correct path, and thus may be forced to raise a machine check exception on detecting any instruction-queue parity error. Figure 1 relates false DUE events to other possible fault outcomes.

This section examines techniques to track false errors and thereby reduce the false DUE AVF. The rest of this section is organized as follows. Section 4.1 discusses the sources of false errors. Section 4.2 discusses the propagation of false error information along with instructions and other objects in a processor, so that later mechanisms (described in Section 4.3) can be used to identify if the error was indeed a false error.

4.1. Sources of False DUE events

Similar to our earlier evaluation [18], we computed an average SDC AVF of 29% for an unprotected instruction queue using an Itanium®2-like microprocessor and dynamic slices of the CPU2000 benchmarks. Thus, on average, a bit in the queue is ACE 29% of the time. The remaining 71% consisted of 30% idle time, 8% Ex-ACE state, and 33% valid un-ACE state. Ex-ACE state denotes bits that were formerly ACE but have been read for the last time; e.g., an ACE instruction in the instruction queue that has been issued for the last time but has not yet been evicted. The instruction may persist in Ex-ACE state just in case it has to be replayed. Un-ACE state is state that is valid (non-idle) from the microarchitecture's perspective, but is unnecessary for correct execution.

Using the above numbers and the analysis from Section 2.2, we can compute the effect of adding an error detection mechanism such as parity on the DUE AVF of the instruction queue, assuming that any detected parity error is declared as a processor error. The true DUE AVF would be the same as SDC AVF without error detection, i.e., 29%. Entries with idle state or Ex-ACE state do not contribute towards either the SDC AVF or the DUE AVF because the hardware never reads such entries (outcome 1 in Figure 1). Faults in un-ACE state correspond to false DUE events, leading to a false DUE AVF of 33%. Thus, adding error detection to the instruction queue changes the SDC AVF from 29% to 0% and the DUE AVF from 0% to 29% + 33% = 62%. Adding parity not only changes the error classification from SDC to DUE, but increases the overall error rate by more than a factor of two.

To eliminate such false errors from the instruction queue and other structures, we need to identify the sources of the errors. Our earlier classification [18] identifies three sources of false errors for the instruction queue:

- *Instructions whose results the microarchitecture will never commit.* Examples of such instructions are wrong-path instructions and predicated-false instructions.
- *Instruction types that are neutral to errors.* No-ops, prefetches, and branch prediction hint instructions, for example, do not affect correctness. Consequently, faults in bits other than the opcode bits will not affect a program's final outcome.
- *Dynamically dead instructions*. These instructions generate values that ultimately do not affect the result. We classify dynamically dead instructions as first-level or transitive. First-level dynamically dead (FDD) instructions are those whose results are not read by any other instruction. Transitively dynamically dead (TDD) instructions are those whose results are used only by first-level dynamically dead instructions or other transitively dynamically dead instructions. Depending on whether the instruction writes a register or a memory location, we classify the dynamically dead instructions as being tracked via register or memory, respectively. A strike on any bit on a dynamically dead instruction, except the destination register specifier bits, will not change the final outcome of a program. Similarly, a strike on the result (e.g., register or memory value) of a dynamically dead instruction will also not affect the program's outcome. On average, dynamically dead instructions account for 20% of all instructions in our binaries. Our earlier paper [18] provides a detailed breakdown of FDD and TDD instructions tracked via both registers and memory.

In the next two subsections we describe techniques to avoid raising a machine check exception on these classes of false errors.

Wang, et al. [30] have identified an additional source of false errors arising from conditional branches. They found that in 40% of the dynamically executed conditional branches in CPU2000 benchmarks, it did not matter which direction the branch went. The techniques described in this paper do not track such false errors. Hence, in this paper, we group these under true DUE AVF when the instruction queue is protected with parity. Back-of-the-envelope calculations show that such conditional branch instructions would reduce the AVF by not more than a few percentage points.

4.2. Mechanism to Propagate Error Information

The key challenge in distinguishing false errors from true errors is that the processor may not have enough information to make this distinction at the point it detects the error. For example, when the instruction queue detects an error on an instruction, it may not be able to tell whether the instruction was a wrong-path instruction or not. Consequently, we need to propagate the error information down the pipeline and raise the error when we have enough information to make this distinction. In this section, we discuss how to propagate this error information for later use. In the next section we will discuss what information we would need to identify false errors.

To propagate error information between different parts of the microprocessor hardware we introduce a new bit called the π bit, which stands for the *possibly incorrect* bit. A π bit is logically associated with each instruction as it flows down the pipeline from decode to retirement. We initially clear the π bit to indicate the absence of any error. When the instruction queue receives the instruction it stores the π bit along with the instruction. On detecting an error (possibly via parity), the instruction queue sets the affected instruction's π bit instead of raising a machine check exception. Subsequently, the instruction issues and flows down the pipeline. When the instruction reaches commit point, we can determine if the instruction was on the wrong path. If so, we can ignore the π bit, avoiding a false DUE event if the bit was set. If not, we have the option to raise the machine check error at the commit point of the instruction. Note that a strike on the π bit itself will result in a false DUE event.

We can easily generalize the π bit mechanism and attach the π bit to different objects flowing through the pipeline, as long as the π bits are propagated correctly from object to object. For example, modern microprocessors typically fetch instructions in multiples, sometimes called chunks [17]. Chunks flow through the front end of the pipeline until they are decoded. We can attach a π bit to each fetch chunk. If the chunk encounters an error, we can set the π bit of the chunk. Subsequently, when the chunk is decoded into multiple instructions, we can copy the π bit value of the chunk to initialize the π bit of each instruction. Thus, we can use the π bit to avoid false DUE events on structures in the front end of the pipeline before individual instructions are decoded.

We can also transfer π bit information from instructions to registers, and thereby avoid false DUE events resulting from dynamically dead instructions on the register file. Instead of raising an error if an instruction's π bit is set, we can transfer the instruction's π bit to the destination register it writes. If no subsequent instruction reads this register, then obviously the π bit of the register will not be examined and, therefore, we will avoid raising an error on an FDD (first-level dynamically dead) instruction that wrote the register. However, when a subsequent instruction reads a register with the π bit set, we can signal an error. This mechanism is similar to Rogers and Li's poison bit [21], where a load destination register was poisoned if the load would have caused a page fault and Mahlke, et al.'s speculative modifier bit [15], which helped avoid exceptions on code scheduled speculatively by the compiler. The Itanium® 2 architecture has a similar mechanism with its NaT (Not a Thing) bit, which is used to track deferred speculative executions. The proposed, but eventually cancelled, Alpha 21464 microarchitecture had a similar mechanism to replay instructions dependent on a load miss.

Alternatively, instead of raising the error if a register's π bit is set, an instruction reading the register could OR the π bits of all its source registers with its own π bit and carry it along the pipeline. This approach would propagate the π bit along dependence chains and allow a processor to track TDD (transitively dynamically dead) instructions as well. Eventually, we can signal an error when the π bit (set to one) propagates to a store or I/O instruction that writes memory or I/O devices. This propagation would not only avoid false DUE for TDD instructions on the register file, but also other structures along the pipeline through which the instructions and values flow.

Similarly, we can transfer the π bit from an instruction or a register to memory values to track false DUE events in memory structures, such as store buffers and caches. We can attach a π bit to each cache block and when a store instruction writes an address, we can transfer the store instruction's π bit to the cache block. Subsequently, when a load reads the cache block it could either examine the π bit or transfer the π bit to the register it is loading. If the π bit is transferred to the register, then we can also avoid signalling false DUE events arising out of dynamically dead memory values.

In general, a π bit can be attached to any object flowing through the pipeline or to any hardware structure, but the granularity of the π bit depends on the implementation. For example, if we attach a π bit to a 64-bit register value, then a single π bit can only tell us that there may have been an error in one of the 64 bits. Alternatively, if we had a π bit per byte, then we can identify which byte among the 64 bits may have had an error. This may be important to instruction sets that allow byte-level writes. More generally, the granularity of the π bit can be refined to isolate the location of errors in the hardware.

We do not, however, expect all hardware structures in a processor or an entire system to be populated with π bits. For example, an implementation may choose to have π bits in caches, but not in main memory. Consequently, when we write-back cache blocks from a cache to main memory, we would lose the π bit information. In such a case, the π bit will go out of scope. When the π bit goes out of scope, an implementation should flag an error if the π bit is set because the system can no longer track the error.

4.3. Distinguishing False Errors from True Errors

As discussed in Section 4.1, false errors arise in the instruction queue from three categories of instructions. In this section we discuss how we can use the π bit information to avoid false errors on these three instruction categories and, thereby, reduce the false DUE rate of the instruction queue.

4.3.1. False Errors on Uncommitted Instructions

Given the π bit, it is relatively straightforward to avoid false errors on instructions that will never commit their results. As explained earlier, the retire unit can ignore the π bit for the wrong-path and falsely predicated instructions, thereby avoiding false errors on such instructions. The retire unit must, however, examine the π bit of instructions on the correct path and flag an error if the π bit is set. In the next two subsections, we will see how to avoid false errors on instructions on the correct path.

4.3.2. False Errors on Neutral Instruction Types

Many instructions, such as the no-ops, prefetches, or branch predict hints, will never affect the final outcome of a program and, therefore, the hardware need not raise an error on non-opcode bits of such instructions. However, to identify such instructions the hardware must decode the instruction at every place it wants to avoid a false error. Instead, we propose using another bit called the anti-π bit, which is associated with every instruction when we decode it. We set the anti-π bit for neutral instruction types and clear it for others. Then, when the instruction queue gets a parity error on non-opcode bits of an entry, it identifies neutral instructions using the anti-π bit, and does not set the π bit on that instruction. In other words, the anti-π bit neutralizes the π bit for those entries. Alternatively, the instruction queue could set the π bit, but carry both the anti-π bit and π bit to the retire unit and take the appropriate decision there.

Note that the hardware could also avoid the anti-π bit on every instruction if it decoded the instruction again at the retire unit. Unfortunately, this means that an instruction must be read after it has been issued and completed. This would force us to include the Ex-ACE time as part of the False AVF calculation, thereby raising the false DUE AVF from 33% to 41% (Section 4.1).

Interestingly, the anti-π bit can be generalized to hardware activities that do not affect the correctness of a program. For example, we could attach an anti-π bit to the command and address generated by a hardware data prefetcher. Any soft error on such an activity can be ignored. The anti-π bit provides a concise mechanism to identify such activities.

4.3.3. False Errors on Dynamically Dead Instructions

Avoiding false errors on dynamically dead instructions is slightly more complex compared to the techniques described in the last two subsections. There are two reasons for this complexity. First, to determine whether an instruction is dynamically dead, we must know whether the instruction's result will ever be used in future by an instruction that is not a first-level dynamically dead (FDD) or a transitively dynamically dead (TDD) instruction (Section 4.1). To determine this, we need to keep information about instructions even after they commit. Current processors usually do not support such a mechanism.

Second, dynamically dead instructions present a trade-off between coverage of false errors versus our ability to precisely point to which instruction encountered a true error and produced the incorrect result. We illustrate this trade-off better using the four designs we outline below.

(1) The Post-commit Error Tracking (PET) Buffer. The PET buffer mechanism avoids signalling errors on a subset of FDD instructions, but can precisely determine the offending instruction that may have encountered a true error. The PET buffer is basically a log of all instructions after they retire. Specifically, a PET buffer entry contains an instruction along with its π bit. When an instruction retires, the retire unit enters the instruction into the PET buffer. If the PET buffer is full, it must evict the oldest instruction from the PET buffer. At this point, it examines the π bit of the instruction to be evicted. If the π bit is clear, then it simply evicts the instruction. However, if the π bit is set, the hardware scans the instructions in the PET buffer to determine if the result produced by the instruction was overwritten by a subsequent instruction in the PET buffer before any intervening read. If so, the instruction is an FDD instruction and consequently the error was a false error. Hence, the processor need not signal the error. However, if the PET buffer cannot verify that the instruction was an FDD instruction—because of an intervening read or the absence of an overwriting instruction in the buffer—then it must signal an error on this instruction.

Obviously, the PET buffer's coverage of false errors on FDD instructions depends on the number of instructions logged in the PET buffer. Across dynamic slices of the CPU 2000 benchmarks, we find that a PET buffer with 512 entries can cover about 32% of the FDD instructions. In Section 6 we discuss the impact of the size of the PET buffer on its coverage of false errors.

In a sense, the PET buffer allows the system to defer committing an instruction's error state past the point where it has committed its result value. Thus, in its design, the PET buffer is similar to the history buffer described by Smith and Plezkun [27]. Nevertheless, the PET buffer is a much simpler structure because it is FIFO and needs to be scanned only when a π bit is set for an instruction being evicted. Such scans should not affect performance because errors in an individual system occur infrequently (on the order of days).

Note that the PET buffer differs significantly from Butts and Sohi's dynamically dead instruction predictor [5]. The latter predicts whether an instruction is dynamically dead before the instruction executes. The PET buffer proves for certain that an instruction is dynamically dead long after it commits.

(2) π bit per register. Instead using the PET buffer, we can allocate a π bit for every register. An instruction's π bit is propagated to its destination register. An error is signalled when an instruction reads a source register with a set π bit. If no instruction reads the register before it is overwritten, the instruction is FDD, and no error is signalled. This mechanism provides 100% coverage on all FDD instructions. However, unlike the PET buffer, when we signal an error, we cannot determine which instruction originally caused the error. This lack of information may complicate some recovery schemes.

(3) π bits on every structure inside the chip, except the memory system. Although the above two mechanisms avoid false errors on FDD instructions tracked via registers, they do not cover instructions that are transitively dead (TDD) via registers. One easy way to track TDD instructions is to declare the error only when a processor interacts with the memory system or I/O devices. Thus, if we have π bits on every structure in a processor—except caches and main memory—and follow the same propagation rule for π bits as described earlier, then we can avoid false errors on TDD instructions as well. This would mean signalling errors only when a store instruction or an I/O access is about to commit its data to the caches, memory system, or I/O device. In this case, we get complete coverage of false errors on TDD instructions tracked via registers, but like the previous mechanism lose our ability to precisely determine which instruction originally encountered the error.

(4) π bit on caches and memory. Finally, if we have π bits on the entire chip and memory system, then we can track false errors on both FDD and TDD instructions tracked via memory as well. In such a case, we would only raise an error when the processor makes an I/O access (e.g., uncached load or store) that has its π bit set. This technique would also allow us to track errors across multiple processors in a shared-memory multiprocessor system.

As the above discussions suggest, the π bit is a powerful mechanism to propagate error information, so that the error can be raised at a later point in time when we can determine whether the error was actually a false or true error. Thus, it decouples the detection of an error from the signalling of the error. This allows a microprocessor designer the choice to raise the error either on the use of a value or when the π bit for a value goes out of scope.

5. Methodology

For our evaluation, we use an Itanium®2-like IA64 processor [13] scaled to current technology. The processor we modeled has a 25-cycle pipeline, runs at 2.5 GHz, and has an issue width of six instructions. It has three levels of cache: an 8KByte L0 cache with 2-cycle hit latency, a 256KB L1 cache with 10-cycle hit latency, and a 10MB L2 cache with a 25-cycle hit latency.

The processor is modeled in detail in the Asim framework [7]. The benchmarks are run on Red Hat Linux 7.2 via an OS simulation front-end. For wrong paths, we fetch the mis-speculated instructions, but do not have the correct memory addresses that a load or store may access.

Table 2 lists the skip interval and input set selected for each of the SPEC CPU2000 programs used for our analysis. The benchmarks were compiled with Intel's electron compiler (version 7.0) with the highest level of optimization. We obtained the number of instructions to skip using Sherwood, et al.,'s [23] SimPoint analysis modified for the IA64 instruction set architecture. For each benchmark we obtained a number of SimPoints, but here we present numbers only for the first SimPoint of each benchmark. We ran each SimPoint for 100 million instructions, which included no-ops.

Table 2: SPEC2000 benchmarks in this paper. M = 1 million.

Integer Benchmarks	Instructions Skipped	Floating Point Benchmarks	Instructions Skipped
bzip2-source	48,900 M	ammp	50,900 M
cc-200	16,600 M	applu	500 M
crafty	120,600 M	apsi	100 M
eon-kajiya	73,000 M	art-110	36,400 M
gap	18,800 M	equake	1,500 M
gzip-graphic	29,000 M	facerec	64,100 M
mcf	26,200 M	fma3d	23,600 M
parser	71,400 M	galgel	5,000 M
perlbmk-makerand	0 M	lucas	123,500 M
twolf	185,400 M	mesa	73,300 M
vortex-lendian3	59,300 M	mgrid	200 M
vpr-route	49,200 M	sixtrack	4,100 M
		swim	78,100 M
		wupwise	23,800 M

Note that binaries used in this paper are slightly different from the ones used for our earlier paper [18] and, hence, some of the results are different from our earlier evaluation.

6. Results

This section discusses our results from reducing exposure to radiation (Section 6.1) and tracking false DUE events (Section 6.2). Section 6.3 shows the impact of combining the above two techniques.

6.1. Reducing Exposure to Radiation

In this section we show how we can reduce the AVF for our 64-entry instruction queue using squashing and fetch throttling. Table 1 shows how the average IPC and average AVFs change when we squash all instructions in the instruction queue after a load miss in the L1 and the L0 caches. When we squash instructions on load misses in the L1 cache, the IPC decreases only by 1.7% (from 1.21 to 1.19) for a corresponding reduction in SDC and DUE AVFs by 26% (from 29% to 22%) and 18% (from 29% to 1.09%), respectively. However, when we squash instructions on L0 misses, the IPC decreases by 10% for a corresponding reduction in SDC and DUE AVFs of only 35% and 23%, respectively.

Overall, squashing on L1 misses appear more profitable because the SDC MITF (proportional to IPC / SDC AVF) and DUE MITF (proportional to IPC / DUE AVF) go up 37% and

Table 1: Impact of squashing on IPC and our instruction queue's SDC and DUE AVFs. Numbers are averaged across all benchmarks.

Design Point	IPC	SDC AVF	DUE AVF	IPC / SDC AVF	IPC / DUE AVF
No squashing	1.21	29%	62%	4.1	2.0
Squash on L1 load misses	1.19	22%	51%	5.6	2.3
Squash on L0 load misses	1.09	19%	48%	5.7	2.3

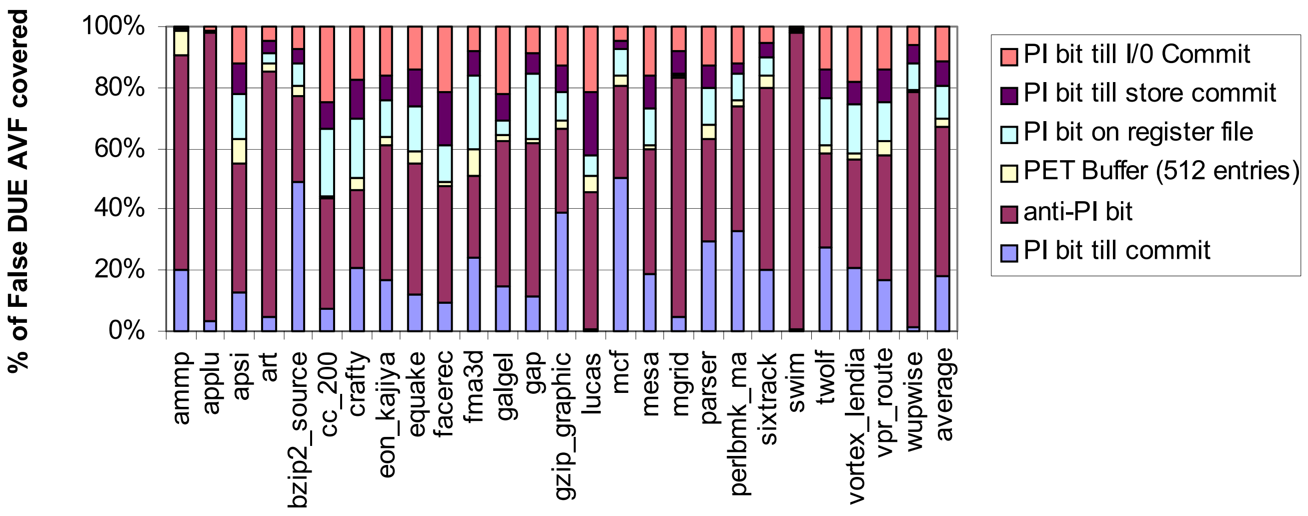

Figure 2. Coverage of the instruction queue's false DUE AVF using various tracking techniques. π = PI.

15%, respectively. In contrast, squashing on L0 misses add very little MITF gain over what squashing on L1 misses achieve.

6.2. Tracking False DUE Events

This section shows the impact of avoiding false errors on an instruction queue protected with parity. Our techniques to avoid false errors could apply to other structures, such as register files and caches, as well. As described in Section 4.3, we use the π bit mechanism to propagate error information and a variety of other information (e.g., anti-π bit) and structures (e.g., PET buffer) to avoid signalling errors on false DUE events.

Figure 2 quantifies how these techniques can avoid false errors and, thereby lower the false DUE AVF of the instruction queue. Propagating the π bit to the commit point allows us to avoid false errors on wrong-path and falsely predicated instructions. On average, this reduces the false DUE AVF for the instruction queue by 18%. However, as the figure shows, the impact is greater for integer benchmarks, which have a higher fraction of such instructions.

In contrast, the anti-π bit has a bigger impact on the floating-point benchmarks than on the integer ones. On average, for floating point benchmarks, the anti-π bit reduces the false DUE AVF by 60% compared to 35% for the integer ones. Across all benchmarks, the anti-π bit reduces the false DUE AVF by 49%. Both no-ops and prefetches have a bigger impact on the false DUE AVF for floating point benchmarks.

Figure 2 also shows the effects of a 512-entry PET buffer. A relatively small PET buffer works because in majority of cases, instructions that write the same register without an intervening read occur within a few hundred committed instructions. On average, the 512-entry PET buffer reduces another 3% of the false DUE AVF. This accounts for about 32% of the FDD (first-level dynamically dead) instructions tracked through registers. Figure 3 shows that two other things can further increase the coverage of the PET buffer: instructions considered for false errors and larger number of entries. FDD instructions tracked via registers can be classified into two categories: ones that are FDD because of a procedure return, so that registers written in the procedure but never used are dead, and the rest of the FDD registers tracked via registers. Finally, we have the third category of FDD instructions tracked via memory. Figure 3 shows the results of all of these with increasing number of entries for the PET buffer. As this figure shows, increasing the number of entries to about 10,000 and including FDD instructions tracked by returns could cover most of the FDD instructions. Of course, a 10,000-entry PET buffer may not be implementable.

The next set of results are for techniques that do not allow the precise determination of the corrupted instruction, but allows a processor to reduce its overall DUE rates. Adding the π bit to the register file allows us to track all FDD instructions. This allows us to reduce the false DUE AVF of the instruction queue by an additional 11% beyond what the PET buffer offers.

We can further enhance the coverage of false errors by tracking TDD (transitively dynamically dead) instructions and the registers they write. If we carry the π bit information throughout the pipeline till the store buffer and examine the π bit only when a store commits its data or a load attempts to get its data from the store buffer, then we can avoid false errors resulting from TDD instructions tracked via registers. This, on average, reduces the instruction queue's false DUE AVF by another 8%.

Finally, if we carry the π bit through the entire processor and memory system, then we have to signal an error only when the processor interacts with an I/O device (e.g., send the final output of a program to the console). This would entirely remove the rest of the false errors by avoiding signalling errors

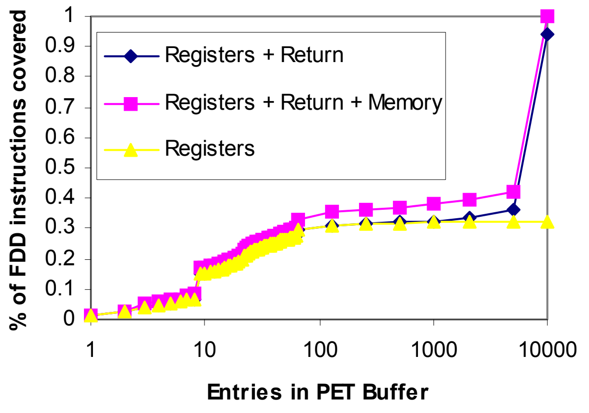

Figure 3. Coverage of FDD (First-Level Dynamically Dead) Instructions from PET buffers of varying size.

on FDD and TDD instructions tracked via memory. This would further reduce the instruction queue's false DUE AVF by 12%.

6.3. Combining Both Techniques

This section summarizes and combines the results from the two prior subsections to show how our techniques can reduce the overall SDC AVF of an unprotected instruction queue and DUE AVF of a parity-protected instruction queue.

To reduce exposure, we choose to squash instructions on an L1 cache miss. This action is triggered as soon as the instruction queue receives a signal from the memory system indicating that an L1 cache miss has occurred. This design decreases IPC by about 2% relative to no squashing. For the parity-protected queue only, we add the π-bit implementation that carries the π bit until the store commit point (option 3 in Section 4.3.3). The π bit does not degrade performance.

As Figure 4 shows, we get an average improvement of 26% in SDC AVF on the unprotected queue from squashing alone. The effect squashing is particularly pronounced in ammp, which gets a 90% reduction in the SDC AVF for only a 7% decrease in IPC. This happens because instructions get queued behind a few critical misses. Hence, squashing and refetching the instructions in the shadow of those misses reduces the AVF dramatically without reducing the IPC significantly.

Combining both squashing and π-bit tracking mechanisms on a parity-protected queue gives a 57% reduction in DUE AVF. In both cases, the IPC impact is 2% on average due to squashing.

7. Related Work

This work is related to four broad areas of research in computer architecture. The first area relates to computing AVFs. We use our own methodology [18] to compute AVFs using ACE analysis with a performance model. Kim and Somani [12] and Wang, et al. [31] describe alternate ways to compute AVFs using statistical fault injection into RTL models. This paper extends the AVF framework by showing how to compute DUE rates and how the DUE AVF is the sum of false and true DUE AVFs.

The second area relates to fault detection and correction strategies. We rely on parity to protect against DUE events and ECC in memory to provide protection against single bit upsets [25]. At the processor level, industrial designs have typically used lockstepping [26][25]. Recently, researchers proposed numerous cost-effective techniques to both detect and recover from transient faults at the thread or processor level (e.g., [3], [22], [20], [29], [17], [8], [19], [14]). However, this paper does not propose any new error detection or correction strategy. Instead, this paper examines strategies to reduce the overall soft error rate of a processor by reducing the amount of time unprotected state is exposed to radiation and by avoiding false errors.

To reduce exposure to radiation we borrow the fetch throttling and instruction squashing schemes used by Tullsen and Brown [28] to improve the performance of a simultaneous multithreaded processor. Fetch throttling has also been proposed by numerous researchers to reduce power consumption of a microprocessor (e.g., [3]).

The fourth area relates to how we avoid signalling false DUE events via the use of the π bit, anti-π bit, PET buffer, and the π bit on a variety of structures, such as the register file, pipeline structures, caches, and main memory. The π bit propagation is most similar to the propagation of the parity bit along the pipeline in the Fujitsu's recent SPARC processor [1]. However, instead of tracking false DUE events, Fujitsu's SPARC uses the parity bit to restart the processor from the instruction that received the parity error. Also, unlike the π bit, the parity bit in the Fujitsu machine is not propagated from instructions to other structures, such as the register file or caches.

Our transfer of the π bit from an instruction to a register is similar to Rogers and Li's poison bit [21], Mahlke, et al.'s speculative modifier bit [15], and the Itanium® 2 architecture's NaT (Not a Thing) bit [10]. Unlike the π bit, Rogers and Li used the poison bit mechanism to avoid raising page faults on speculative loads, but did not propagate the poison bit from registers to instructions. Similarly, Mahlke, et al. used the speculate modifier bit to avoid raising exceptions on code speculatively scheduled by the compiler, but did not propagate the speculative modifier bit from registers back to instructions. The Itanium® 2 architecture uses the NaT bit to track deferred speculative executions. However, unlike Rogers and Li's poison bit, but like the π bit, the NaT bit can be propagated between registers. Nevertheless, the NaT bit is restricted to registers and are not transferred to other microarchitectural structures, unlike the π bit. Additionally, the NaT bit is visible to the programmer, but the π bit is not.

Finally, although we examined the concepts of false DUE AVF in the context of micro-level error detection schemes, such as parity, they also apply to macro-level fault detection schemes, such as lockstepping [26] and Redundant Multithreading (RMT) [17] as well. For example, two lockstepped processors may get a false error when a branch predictor bit is struck with a neutron or alpha particle. The strike on the branch

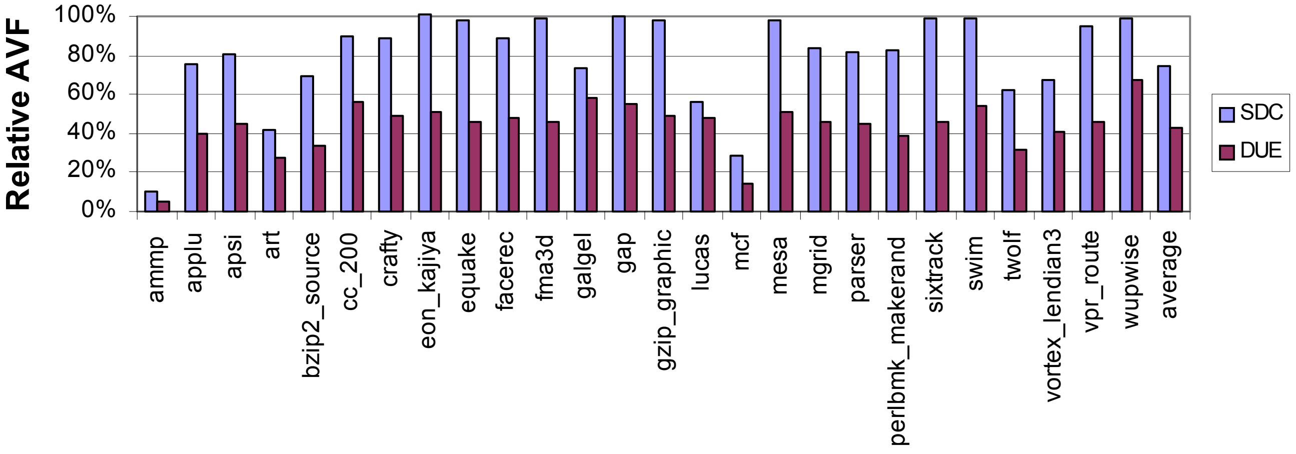

Figure 4. Impact of exposure and false DUE reduction techniques on an instruction queue's SDC and DUE AVFs. Relative SDC AVF = SDC AVF with optimizations / SDC AVF with no optimizations. Relative DUE AVF is defined similarly.

predictor of one of the processors will not result in incorrect execution, but it may cause a divergence in what each processor produces in subsequent cycles. Because cycle-by-cycle lockstepping expects the same outputs from both processors in every cycle, such a divergence will create a mismatch and, hence, a false error. In contrast, RMT will not produce a false error on a branch predictor strike because it compares committed instructions, which are not affected by branch mispredictions. Nevertheless, in the absence of any special mechanism, some implementations of RMT may produce a false error (e.g., when comparing every instruction) if a dynamically dead instruction gets corrupted.

8. Conclusions

Soft errors induced by neutron or alpha particle strikes are emerging as a significant obstacle to increasing processor transistor counts in future process technologies. Although fault rates of individual transistors are not projected to rise, incorporating more transistors into a device makes that device more likely to encounter a fault. As a result, we expect that maintaining processor error rates at acceptable levels will require increasing design effort.

Soft errors in microprocessors can be classified as silent data corruption (SDC), where a fault induces the system to generate erroneous outputs, and detected unrecoverable error (DUE), where the hardware detects the error, but cannot recover from it. The SDC rate of a processor can be computed as the sum over the product of the raw error rate per device and a device's SDC AVF, or architectural vulnerability factor. SDC AVF expresses the probability that a strike affecting the device eventually results in an error in a program's output. Similarly, the DUE rate of a processor can be computed as the sum over the product of the raw error rate per device and a device's DUE AVF, where DUE AVF is the probability that a strike will result in a DUE event. The DUE AVF can be expressed as the sum of the true DUE AVF and false DUE AVF. False DUE AVF arises from false errors in a microprocessor, such as a strike affecting a wrong-path instruction.

This paper proposed two simple approaches to reduce the SDC and DUE AVFs, and hence the soft error rates, of a microprocessor. We evaluated the application of these approaches using a microprocessor instruction queue. The first approach reduced the amount of time instructions sit in vulnerable storage structures by selectively squashing instructions when long delays are encountered. A fault is less likely to cause an error if the structure it affects does not contain valid instructions. We introduced a new metric, MITF (Mean Instructions To Failure), to capture the trade-off between performance and reliability introduced by this approach. We found that instruction squashing on L1 cache misses could reduce the SDC AVF for an unprotected instruction queue by 28% or DUE AVF of a parity-protected instruction queue by 18% for only a 2% decrease in IPC. This increases the SDC MITF by 37% or the DUE MITF by 15%, thereby allowing more work to be done between two soft errors.

The second approach reduces the false DUE AVF by reducing false errors. In the absence of a fault detection mechanism, such errors would not have affected the final outcome of a program. For example, a fault affecting the result of a dynamically dead instruction would not change the final program output, but could still be flagged by the hardware as an error. To avoid signalling such false errors, we modified a pipeline's error detection logic to mark affected instructions and data as possibly incorrect—via the use of the π bit—rather than immediately signaling an error. Then, we signal an error only if we determined later that the possibly incorrect value could have affected the program's output. We proposed several alternatives, such as the anti-π bit, the post-commit error tracking (PET) buffer, and incorporation of the π bits in register files, caches, and main memory, to detect false errors arising out of different kinds of instructions. Our results showed that using these mechanisms we can cover 100% of such false DUE events without any performance degradation.

Overall, our mechanisms reduced the SDC AVF of an unprotected instruction queue by 26% and the DUE AVF a parity-protected instruction queue by 57% for only a 2% decrease in overall IPC. Once these mechanisms are in place, they can also reduce the AVF of other structures, such as the register file.

Acknowledgments

We would like to thank Mike Smith for inspiring the concept of false errors and Tryggve Fossum for the MITF metric, Sudhanva Gurumurthi, Geoff Lowney, and Paul Racunas for their helpful comments on early drafts of this paper, Intel's Softsdv and Asim groups for help with the simulation system, and Robert Cohn and Harish Patil for helping to reproduce the SimPoint analysis on IA64.

References

[1] H. Ando, et al., "A 13 GHz Fifth Generation SPARC64 Microprocessor," International Solid-State Circuits Conference (ISSCC), 2003.

[2] T. Austin, "DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design," ACM/IEEE 32nd Annual Symposium on Microarchitecture (MICRO-32), November 1999.

[3] A. Baniasadi and A. Moshovos, "Instruction Flow-Based Front-end Throttling for Power-Aware High-Performance Processors," International Symposium on Low Power Electronics and Design, 2001.

[4] D. Bossen, "CMOS Soft Errors and Server Design," 2002 IRPS Tutorial Notes - Reliability Fundamentals," April 7, 2002.

[5] J. Adam Butts and G.S. Sohi, "Dynamic Dead-Instruction Detection and Elimination," 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 2002.

[6] T. Calin, et al., "Topology-Related Upset Mechanisms in Design Hardened Storage Cells," Radiation and Its Effects on Components and Systems, 1997. RADECS 97. Fourth European Conference on , 15-19 Sept. 1997.

[7] J. Emer, P. Ahuja, N. Binkert, E. Borch, R. Espasa, T. Juan, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, and S. Wallace, "Asim: A Performance Model Framework," IEEE Computer, 35(2):68-76, Feb. 2002.

[8] M. Gomaa, C. Scarbrough, T.N. Vijaykumar, and I. Pomeranz, "Transient Fault Recovery for Chip Multiprocessors," International Symposium on Computer Architecture (ISCA), 2003

[9] S. Hareland, J. Maiz, M. Alavi, K. Mistry, S. Walstra, and C. Dai, "Impact of CMOS Scaling and SOI on soft error rates of logic processes," VLSI Technology Digest of Technical Papers, 2001.

[10] Intel, "Intel Itanium® Architecture Software Developer's Manual," Intel Corporation, 2002.

[11] T. Karnik, B. Bloechel, K. Soumyanath, V. De, and S. Borkar, "Scaling trends of Cosmic Rays induced Soft Errors in static latches beyond 0.18μ," Symposium on VLSI Circuits Digest of Technical Papers, 2001.

[12] S. Kim and A.K. Somani, "Soft Error Sensitivity Characterization for Microprocessor Dependability Enhancement Strategy," International Conference on Dependable Systems and Networks (DSN), 2002.

[13] K. Krewell, "Intel's McKinley Comes Into View," Microprocessor Report, Volume 15, Archive 10, October 2001.

[14] S.-C. Lai, S.-L. Lu, K. Lai and J.-K. Peir, "Ditto Processor," International Conference on Dependable Systems and Networks (DSN), 2002.

[15] S. A. Mahlke, W. Y. Chen, W. W. Hwu, B. R. Rau, and M. S. Schlansker, "Sentinel Scheduling for VLIW and Superscalar Processors," 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Oct. 1992.

[16] S. S. Mukherjee, T. Fossum, J. Emer, and S. K. Reinhardt, "Cache Scrubbing in Microprocessors: Myth or Necessity?" 10th International Symposium on Pacific Rim Dependable Computing (PRDC), Papeete, Tahiti, March 2004.

[17] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed Design and Evaluation of Redundant Multithreading Alternatives," 29th Annual International Symposium on Computer Architecture (ISCA), 2002.

[18] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," 36th Annual International Symposium on Microarchitecture (MICRO), December 2003.

[19] J. Ray, J. Hoe, and B. Falsafi, "Dual Use of Superscalar Datapath for Transient-Fault Detection and Recovery," International Symposium on Microarchitecture (MICRO), December 2001.

[20] S. K. Reinhardt and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," 27th Annual International Symposium on Computer Architecture (ISCA), June 2000.

[21] A. Rogers and K. Li, "Software Support for Speculative Loads," 5th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 38-50, Oct. 1992.

[22] E. Rotenberg, "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors," Proceedings of the Fault-Tolerant Computing Systems (FTCS), 1999.

[23] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, "Automatically Characterizing Large Scale Program Behavior," 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October 2002.

[24] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi, "Modeling the Effect of Technology Trends on the Soft Error Rate of Combinatorial Logic," Dependable Systems and Networks (DSN), 2002.

[25] D. R. Siewiorek and R. S. Swarz, "Reliable Computer Systems Design and Evaluation," Published by A.K.Peters, Ltd., 1998.

[26] T.J. Slegel, et al., "IBM's S/390 G5 Microprocessor Design," IEEE Micro, pp 12-23, Mach/April 1999.

[27] J.E. Smith and A.R. Plezkun, "Implementing Precise Interrupts in Pipelined Processors," IEEE Transactions On Computers, 37:5 (May), pp. 562-573, 1988.

[28] D. Tullsen and J. A. Brown, "Handling Long-Latency Loads in a Simultaneous Multithreaded Processor," 34th Annual International Symposium on Microarchitecture (MICRO), Dec. 2001.

[29] T.N. Vijaykumar, I. Pomeranz, and K. Cheng., "Transient Fault Recovery via Simultaneous Multithreading," International Symposium on Computer Architecture (ISCA), 2002

[30] N. Wang, M. Fertig, and S. Patel, "Y-Branches: When You Come to a Fork in the Road, Take It," 12th International Conference on Parallel Architectures and Compilation Techniques (PACT), 2003.

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

[32] J.F. Ziegler, et al., "IBM experiments in soft fails in computer electronics (1978 – 1994)," IBM Journal of Research and Development, pp. 3 – 18, Volume 40, Number 1, January 1996.