Reliability in the Shadow of Long-Stall Instructions

Vilas Sridharan David Kaeli
ECE Department
Northeastern University
Boston, MA 02115
{vilas, kaeli}@ece.neu.edu

Arijit Biswas
FACT Group
Intel Corporation
Hudson, MA 01749
arijit.biswas@intel.com

***Abstract*—Soft errors due to cosmic rays are now a major concern for both computer manufacturers and end users. Due to continually shrinking silicon manufacturing processes and greater chip integration, the chip- and system-level soft error rate is projected to continue increasing for the forseeable future. Due to these concerns, chip manufacturers have long designed cache structures (generally the largest on-chip structures in a high-performance processor) to include protection techniques such as parity or ECC. In order to meet SER targets, new chip designs are starting to incorporate such protection on physical register files. As the soft error rate increases, however, large processor pipeline structures will also require protection against soft errors. Unfortunately, many of these pipeline structures are latency-critical, needing to complete multiple accesses per processor cycle. Thus, many of these structures are ill-suited for protection techniques such as ECC, which can add latency to each access.**

In modern processors, these structures typically contain in-flight instructions, which can vary in their vulnerability contribution. Thus, uniform protection such as that provided by ECC may not be necessary. Our goal is to explore and explain some of the underlying causes of this variation in vulnerability among instructions. In this study, we examine the vulnerability contribution of instructions that are in-flight during (in the *shadow* of) long-stall instructions, which will define the maximum potential benefit of techniques that exploit these stall cycles.

I. Introduction

Soft errors due to cosmic rays are now a major concern for both computer manufacturers and end users. Due to continually shrinking silicon manufacturing processes and greater chip integration, the chip- and system-level soft error rate is projected to continue increasing for the forseeable future [2].

Due to these concerns, chip manufacturers have long designed cache structures (generally the largest on-chip structures in a high-performance processor) to include protection techniques such as parity or ECC [9], [11], [15]. In order to meet SER targets, new chip designs now incorporate such protection on physical register files [8], [11].

As the soft error rate increases, however, large processor pipeline structures such as the re-order buffer will also require protection against soft errors. Unfortunately, many of these pipeline structures are latency-critical, with frequent accesses that must be completed within a single processor cycle. Thus, these pipeline structures are ill-suited for protection techniques such as ECC, which can add latency to each access. As far as we are aware, few current or planned processors will include ECC protection on these core pipeline structures due to these constraints.

Unlike caches, however, many of these pipeline structures contain almost entirely speculative data; only some of this data (correct-path instructions) will affect processor state, while the rest (mis-speculated instructions) will not. Thus, the correctness of wrong-path instructions is not critical to correct operation of a processor. This has been noted previously in [12]. The broader implication is that in-flight instructions vary in their impact on processor correctness; some in-flight instructions are more *vulnerable* than others. This is true even among correct-path instructions, as some instructions are more susceptable to soft errors than others. A simple example is NOP instructions, which are in some sense "less" vulnerable than other instructions, since none of the operand or result fields are needed. Our goal is to explore and explain some of the underlying causes of this variation among correct-path instructions.

II. Related Work

Until recently, research into architectural reliability techniques dealing with pipeline structures has focused on providing as close to perfect coverage as possible. Thus, these studies have typically focused on full duplication, either temporal [14], [17], or spatial [16]. Although these techniques provide virtually 100% coverage against soft errors, they typically have close to 2x overhead in either area or performance. While this high cost may be acceptable for server- and mainframe-class applications where this level of protection is necessary, it may not be acceptable for desktop applications where full protection is not a requirement.

Recent studies have examined partial protection as a means of achieving a substantial increase in reliability while minimizing the performance and area penalty. In [7], the authors suggest a combination of explicit and implicit redundancy based on processor utilization. Thet suggest providing for explicit duplication of instructions during low-ILP phases such as L2 cache misses. Others have also suggested flushing pipeline structures such as the issue queue during cache misses [18].

The common theme in both studies is to use cycles where the processor is otherwise stalled (e.g., during cache misses) to provide partial protection with minimal performance overhead. However, neither study analyzes how vulnerable these instructions are to soft errors. In essence, these studies suggest providing protection when it is "easy", rather than when it is needed. Few studies have done extensive work on determining

the causes of vulnerability variation among instructions. In [6], the authors note asymmetry in instruction vulnerability, but the study does not delve into its causes. Our goal is to explore the causes of this variability; specifically, we would like to determine the maximum benefit of adding reliability enhancements during stall cycles. As such, we would like to determine the relative vulnerability of instructions behind stalled instructions in order to define the maximum potential benefit of techniques that exploit these stalls. Thus, we examine the vulnerability contribution of instructions that are in-flight during (in the *shadow* of) long-stall instructions.

III. Methodology

Long-stall instructions can be the result of either long-latency instructions (instructions that inherently take many cycles to execute) or resource-stalled instructions (instructions that stall due to an unavailable resource). Common types of long-stall instructions are resource stalls such as cache misses or long-latency instructions such as floating point square root or integer divide. In modern out-of-order processors, however, not all such operations result in processor stalls; this ability to overlap stall time with execution is the main benefit of out-of-order operation. As such, we only want to include in our study those operations that do stall the commit stage of the processor pipeline. We define a *long-stall instruction*, or LSI, as an instruction that stalls commit for more than 10 cycles without causing a pipeline flush. We define an LSI's *shadow* as all other in-flight instructions when that LSI commits. Instructions in the shadow of an LSI reside in the pipeline while the LSI completes, and thus the shadow will define the maximum number of instructions that could be affected by techniques that exploit LSIs. We choose the 10 cycle threshold because it is a lower bound on the latency of our L2 cache and our complex functional units. In addition, most vulnerability reduction techniques will require several processor cycles to work; thus, protecting on stalls of shorter than 10 cycles may not be profitable.

One example of an LSI is a cache miss that does not complete 10 cycles after it becomes the oldest ROB entry. On the other hand, a mispredicted branch instruction is not an LSI. Although the mispredicted branch stalls commit, it also results in a pipeline flush and therefore would have an empty shadow. Therefore, we do not treat a mispredicted branch as an LSI.

To measure reliability, we use the *vulnerability* metric presented in [1] and adapt it to measure the reliability of processor pipeline structures. This method defines a structure's vulnerability as:

$$\begin{aligned} Vulnerability = & \frac{W \times \sum_{i=1}^{N} CT_i}{TT} \\ & (W = word\ size\ in\ bits) \\ & (N = number\ of\ critical\ words) \end{aligned} \quad (1)$$

Measuring a structure's vulnerability is equivalent to measuring its *Architectural Vulnerability Factor* (AVF) [12]; the vulnerability of a structure is equal to its AVF times its size in bits. This metric is useful when performing comparisons between structures; AVF values (measured in percent) cannot be directly compared across structures.

Parameter	Value
Issue Width	8 instructions
Commit Width	8 instructions
Physical Registers	256 integer / 256 FP
Issue Queue	64 entries
Re-Order Buffer	192 entries
Load-Store Queue	32 loads / 32 stores
Frequency	2 GHz
L1 D-Cache	64 kB, 2 cycle access 2-way set-associative
L1 I-Cache	32 kB, 2 cycle access 2-way set-associative
L2 Cache	2 MB, 10 cycle access 8-way set-associative
Memory Latency	100 ns

TABLE I
Simulated Machine Configuration

For our study, we measure the vulnerability of four large pipeline structures: the re-order buffer (ROB); the issue queue (IQ); the load buffer (LB); and the store buffer (SB). These structures are large and latency-critical pipeline structures which must allow for multiple accesses per cycle, and providing error detection and correction is difficult without severely compromising performance targets. This motivates the need for other reliability enhancement techniques.

Next, we describe in detail our methodology for measuring the vulnerability of the ROB. Our methodology for the other structures is similar.

In our model, the ROB entry for every committed instruction is a *Critical Word* (CW), and the entry's *Critical Time* (CT) is its lifetime in the ROB. A ROB entry's Critical Time begins when it is created (when the instruction it contains is dispatched). Since each ROB entry contains information necessary for commit, the entry's CT ends only when the entry is deleted at commit. Note that mis-speculated instructions, which are squashed before commit, are not Critical Words and do not contribute to the overall vulnerability.

Note that different instructions differ in their use of various ROB fields. Thus, certain fields in the ROB will be vulnerable for some instructions but not for others. For example, NOP instructions do not use any of the register fields, store instructions do not use the destination register field, and so on. Since these differences in ROB usage are static, we account for them in our model by setting the W parameter on a per-opcode basis.

We assume a ROB width of 72 bits, comprised of the fields given in Table II. All status bits (10 bits) and the opcode bits (6 bits) are assumed to be critical for all committed instructions. The other fields are derated on a per-opcode basis. Load and store operations do not use the RC or Function field; branch instructions do not use the RB, RC, or Function fields; and ALU operations do not use the displacement field.

For the IQ, we assume an overall bit width of 100 bits. We divide both the load and store buffers into multiple sub-

Field	Width
Opcode	6 bits
RA	8 bits
RB	8 bits
RC	8 bits
Displacement	21 bits
Function (ALU)	11 bits
Status	10 bits

TABLE II
REORDER BUFFER FIELDS

structures: a 20-bit sub-structure for status and entry ID bits and a 64-bit sub-structure for the memory address. The store buffer has an additional 64-bit sub-structure to hold store data. The vulnerabilities of each of these sub-structures is measured separately and summed to form the total for each buffer.

Unlike ROB and SB entries, IQ and LB entries are not necessarily vulnerable until the entry is deleted. In the issue queue for example, an entry is not necessarily deleted once its associated instruction has issued; the entry may be retained in case the instruction must be replayed due to an execution fault or misprediction (e.g., a memory dependence misprediction). If the instruction faults and is replayed, the IQ entry remains vulnerable until the instruction re-issues. If the instruction executes successfully, however, the entry is only vulnerable until the instruction's issue time and not thereafter. Thus, a part of that entry's lifetime is non-critical, and must be treated as such. In general, an IQ entry is only vulnerable until its corresponding instruction issues for the final time.

Similarly, a load buffer entry is held until instruction retirement although the address is sent to the memory subsystem much earlier. Therefore, the LB status field is vulnerable until the load commits, while the address field is only vulnerable until the address is issued to the memory subsystem. A complication arises here, however, because subsequent loads and stores check prior load entries for address conflicts. Address entries are vulnerable to these lookups if the compared address differs by at most one bit (see the description on how to calculate tag address vulnerability in [1] and [3]). Care must be taken to account for this vulnerability; these cases are all tracked by our infrastructure.

IV. EXPERIMENTAL SETUP

For our study, we use the detailed CPU model provided in the M5 Simulator System [4]. This CPU models an Alpha 21264-like micro-architecture [10], with 256 physical integer and 256 physical floating point registers, a 64-entry issue queue, a 32-entry load buffer, a 32-entry store buffer, and a 192-entry re-order buffer. These parameters are chosen to be representative of a modern high-performance microprocessor. A summary of the relevant parameters of our simulated machine is shown in Table I. We extend this CPU model with a framework to measure the reliability of any on-chip structure.

In this study, we do not take into account the effects of dynamically-dead instructions [5]. These are committed instructions whose outputs are never read. Thus, a dynamically-dead load will have fewer critical bits in its ROB entry (the RA field will become non-critical). However, it is likely that these instructions will be present both inside and outside of LSI shadows, thus comparisons between the two will still be relatively accurate.

Our analysis uses a subset of the SPEC CPU2000 benchmarks, both integer and floating point, using the single early simulation points given by the Simpoint analysis [13]. For each benchmark, we warmup the cache structures for 100M instructions before beginning detailed execution. Although we are not directly measuring any cache statistics, startup misses would be treated as long-stall operations and would bias our results.

V. RESULTS

Figure 1 shows the overall vulnerability of each of the four structures across all benchmarks. From this figure, it is clear that the ROB and IQ have much higher vulnerabilities on average than the LB or SB. This is a somewhat intuitive result, since the ROB and IQ have many more entries than do the LB or SB. However, note that the ROB has only a slightly higher vulnerability than the IQ on average, despite having three times the number of entries. Figure 2 shows the overall AVFs for each of these structures; from this, it can be seen that the IQ has a slightly higher AVF on average than the ROB, while the LB and SB have much lower AVFs.

Figure 3 presents the relative contribution of LSI-shadow instructions to the overall vulnerability of the ROB, IQ, LB, and SB across all benchmarks. The figure shows that instructions in an LSI shadow contribute approximately 60% of the overall vulnerability on average across all structures. The vulnerability breakdown for each structure is similar; approximately 60% of the ROB, IQ, and LB vulnerability is from LSI-shadow instructions. In the SB, LSI-shadow instructions contribute slightly less than 50% of the vulnerability. The store buffer has a slightly lower percentage of shadow instructions because, unlike the other structures, the store buffer can contain committed state. Much of the SB vulnerability is due to committed instructions waiting to be written to memory that are not part of an LSI shadow.

Figure 4 shows the total instruction count based on the same groupings. This shows that instructions in an LSI shadow account for only 33% of the total instruction mix on average. This figure is much lower in general for the SPECint benchmarks, which have many fewer long-latency floating-point operations than the SPECfp benchmarks. We can also compute the vulnerability per instruction for each benchmark. The average vulnerability per LSI-shadow instruction across all benchmarks is 5.8, while the average non-LSI-shadow vulnerability is 1.03. This implies that on average each shadow instruction has approximately five and a half times the vulnerability of non-shadow instructions.

These results imply that a technique targeting instructions residing in an LSI-shadow can reduce pipeline vulnerability by up to 60% at most (for the SPEC benchmarks). This is an important result, as it gives an upper bound to the expected benefit of this type of technique for a general class

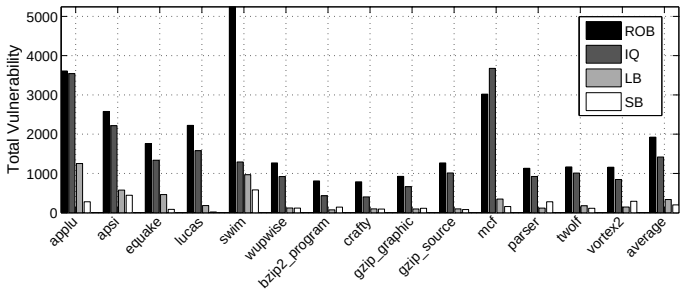

Fig. 1. Total Vulnerability by Structure. This shows the vulnerability of each of the four measured structures across all benchmarks.

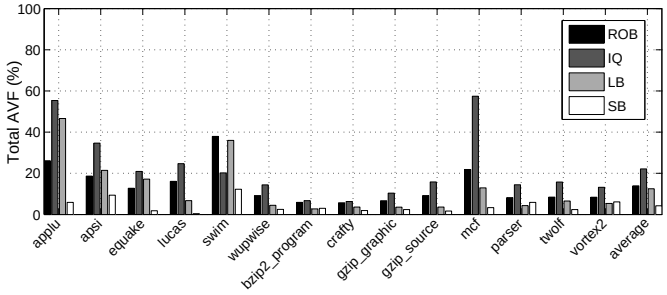

Fig. 2. Total AVF by Structure. This shows the Architectural Vulnerability Factor of each of the four measured structures across all benchmarks.

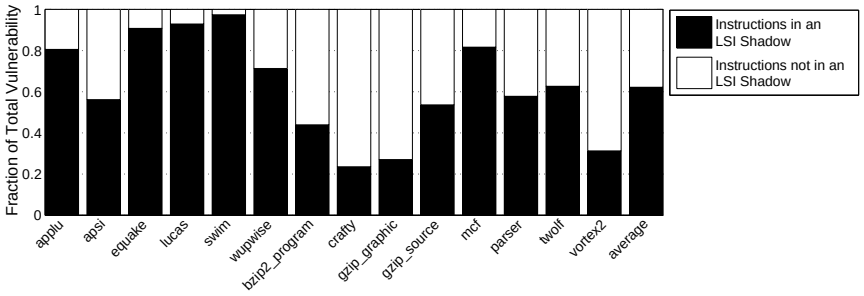

Fig. 3. Total Vulnerability. The fraction of the total vulnerability across all structures that comes from LSI-Shadow instructions.

Instruction Type	SPECint Percentage	SPECfp Percentage
FP Load Quadword	0.7	74.6
Load Quadword	49.2	9.0
FP ALU	4.9	16.0
Load Word	20.8	0.0
Load Byte	12.8	0.0
Load Longword	10.5	0.3
Other	<1.0	<0.1

TABLE III
PERCENTAGE OF TOTAL LSIS BY OPCODE

of workloads. It is certain that any real technique would show less vulnerability reduction, due to inherent overhead and less-than-perfect knowledge of the processor state.

We have also examined the types of instructions that result in LSIs; a summary of these results is detailed in Table III. For the SPECint benchmarks, more than 95% of the stalls were the result of load miss resource stalls. Most of the remaining stalls were due to long-latency floating point operations, which are present to some degree even in the SPECint benchmarks. For SPECfp, a higher percentage of stalls are due to floating-point operations such as divide and square root. But these still only account for 16% of the stalls, while load miss stalls account for the remaining 84% of stalls. It is clear from this data that our out-of-order processor is reasonably effective at preventing stalls from FP operations, but much less effective at hiding stalls due to cache misses (which can have a much longer latency).

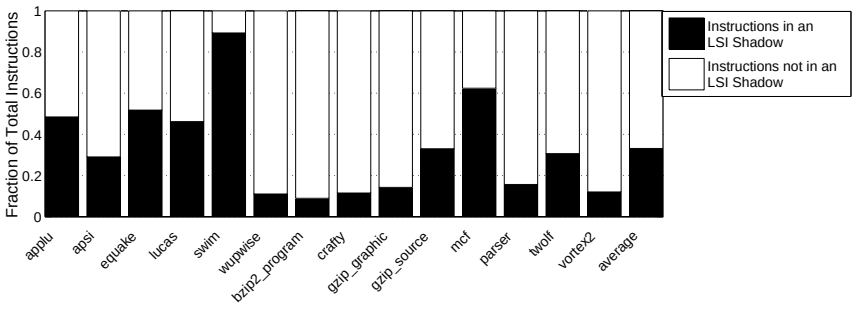

Fig. 4. Breakdown of Committed Instructions. The fraction of committed instructions that are in an LSI-Shadow for each benchmark.

VI. Conclusion and Future Work

Our study begins a systematic look at the underlying causes of instruction vulnerability variation in the pipeline of a modern, high-performance processor. We focus specifically on instructions in the shadow of instructions that stall the pipeline, and quantify the vulnerability associated with these instructions across the reorder buffer, load and store buffers, and issue queue. We show that on average over the SPECint2000 benchmarks, almost 60% of the vulnerability for most of those structures is due to instructions that reside in the shadow of a long-stall instruction.

Our next steps include characterizing other types of events that yield high-vulnerability instructions as well as methods for reducing the vulnerability of these instructions. We would also like to characterize more on-chip structures. The ultimate goal is to measure vulnerability for the entire CPU over a large range of workloads to give a comprehensive view of chip reliability. Then, we will be able to determine if the trends noted here hold across a full range of on-chip structures and a variety of workloads.

References

[1] G. Asadi, V. Sridharan, M. B. Tahoori, and D. Kaeli, "Balancing Performance and Reliability in the Memory Hierarchy," Proceedings of the International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 269-279, Austin, TX, March 2005.

[2] R. Baumann, "Radiation-Induced Soft Errors in Advanced Semiconductor Technologies," IEEE Transactions on Device and Materials Reliability, Vol. 5, No. 3, pp. 305 - 316, September 2005.

[3] A. Biswas, P. Racunas, R. Cheveresan, J. Emer, S. S. Mukherjee, and R. Rangan, "Computing Architectural Vulnerability Factors for Address-Based Structures," Proc. of the 32nd Annual Intl. Symp. on Computer Architecture (ISCA'05), pp. 532-543, 2005.

[4] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, S. K. Reinhardt, "The M5 Simulator: Modeling Networked Systems," IEEE Micro, Vol. 26, No. 4, pp. 52-60, Jul - Aug 2006.

[5] J. A. Butts and G. Sohi, "Dynamic dead-instruction detection and elimination," Proc. of the 10th Intl. Conf. on Architectural Support For Programming Languages and Operating Systems (ASPLOS'02), pp. 199-210, San Jose, CA, October 2002.

[6] X. Fu, J. Poe, T. Li, and J. A. Fortes, "Characterizing Microarchitecture Soft Error Vulnerability Phase Behavior," Proceedings of the 14th International Symposium on Modeling, Analysis, and Simulation (MASCOTS), pp. 147 - 155, Monterey, CA, September 2006.

[7] M. A. Gomaa and T. N. Vijaykumar. "Opportunistic Transient-Fault Detection," Proceedings of the 32nd Annual international Symposium on Computer Architecture (ISCA), pp. 172-183, Madison, WI, June 2005.

[8] G. Grohoski, "Niagara-2: A Highly-Threaded Server-on-A-Chip," HotChips 2006, Stanford, CA, August 2006.

[9] R. Kalla, S. Balaram, and J.M Tendler, "IBM Power5 Chip: a Dual-Core Multithreaded Processor," IEEE Micro, Vol. 24 , No. 2, pp. 40-47, Mar - Apr 2004.

[10] R. E. Kessler, E. J. McLellan, and D. A. Webb, "The Alpha 21264 Microprocessor Architecture," Proceedings of the International Conference on Computer Design (ICCD), pp. 90 - 95, Austin, TX, October 1998.

[11] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," IEEE Micro, Vol. 25, No. 2, pp. 21-29, Mar - Apr 2005.

[12] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor," Proceedings of the International Symposium on Micro-architecture (MICRO), pp. 29-40, San Diego, CA, December 2003.

[13] E. Perelman, G. Hamerly, and B. Calder "Picking Statistically Valid and Early Simulation Points," Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT), New Orleans, LA, September 2003.

[14] S. K. Reinhardt and S. S. Mukherjee, "Transient fault detection via simultaneous multithreading," Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA), pp. 25 - 36, Vancouver, CA, June 2000.

[15] S. Rusu, H. Muljono, and B. Cherkauer, "Itanium 2 processor 6M: higher frequency and larger L3 cache," IEEE Micro, Vol. 24, No. 2, pp. 10-18, Mar - Apr 2004.

[16] T.J. Slegel, E. Pfeffer, and J.A. MaGee, "IBM's S/390 G5 Microprocessor Design,", IEEE Micro, Vol. 19, No. 2, pp. 12-23, Mar - Apr 1999.

[17] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzyk, "Fingerprinting: Bounding the Soft-Error Detection Latency and Bandwidth," Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 224 - 234, Boston, MA, October 2004.

[18] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor" Proceedings of the International Symposium on Computer Architecture (ISCA), pp. 264-275, Munich, June 2004.