A Cost-Effective Implementation of an ECC-protected Instruction Queue for Out-of-Order Microprocessors

Vladimir Stojanovic, R. Iris Bahar, Jennifer Dworak
Brown University, Division of Engineering
Providence, RI 02912
{vlada,iris,jdworak}@lems.brown.edu

Richard Weiss
Evergreen State College
Olympia, WA 98505
weissr@evergreen.edu

ABSTRACT

Major sources of transient errors in microprocessors today include noise and single event upsets. As feature sizes and voltages are reduced to create faster, more efficient, and computationally more powerful processors, these errors will increase significantly. We show that (contrary to conventional wisdom) error correction codes (ECC) can be efficiently utilized to handle these errors as instructions are being processed through the microprocessor pipeline. We will analyze some of the tradeoffs involved in a hardware implementation of ECC for the instruction queue with respect to performance, power, area, and reliability. Specifically, for an environment with high error rates, we show that we can correct all single bit errors with a negligible drop in performance. Our approach can be generalized to other data structures within the microprocessor, including the register file and reorder buffer.

Categories and Subject Descriptors: B.8.1 [Performance and Reliability]: Reliability, Testing, and Fault-tolerance

General Terms: Reliability, Design.

Keywords: Reliability, Error Correcting Codes, Instruction Queue.

1. INTRODUCTION

Error mitigation and fault tolerance are becoming increasingly significant for multiple reasons—many of which can be attributed to device scaling. For example, single event upsets (SEUs) generated by particle strikes may erroneously change the state of the machine—generating *soft errors.* Although soft errors are often considered a problem for memories, the sensitivity of combinational and sequential logic circuits (including latches and flip-flops) to soft errors is increasing with device scaling. In the future, these soft errors within the circuit logic will be an important contributor to overall soft error failures [1]. In addition, as operating voltages decrease and devices scale to smaller feature sizes, circuits become much more susceptible to *noise* and process variations—significantly increasing the likelihood of failures. Indeed, we expect error rates to increase by several orders of magnitude.

Much work has already been done in the area of fault tolerance for microprocessors. However, the use of hardware-based error correction codes is often dismissed as too expensive *a priori*. Yet, alternative approaches have their own costs and shortcomings. These alternative methods may employ large buffers [2, 3] and additional control logic [4] without necessarily capturing all errors or maintaining performance [5]. Other error correction schemes require the use of traps to the operating system [6]. Unfortunately, these traps can become costly as error rates increase. Finally, effective error handling strategies depend upon the type of processor being employed. For instance, some strategies (such as fetch halting and instruction squashing) that are acceptable for an in-order processor [7] may lead to unacceptable performance losses for an out-of-order processor [8].

In response to these observations, this paper evaluates a hardware implementation of a Hamming code for error correction in the instruction queue of an out-of-order processor without operating system intervention. The Hamming codes we use have the obvious advantage of being able to not only detect, but also correct single-bit errors in the instruction queue. However, the best way to implement these Hamming codes in a modern superscalar microprocessor is not necessarily obvious. Tradeoffs between error correction and the hardware, power, and performance costs must be made. This paper will explore some of these tradeoffs. Specifically, we will show that it is possible to use fewer ECC hardware checkers than the machine width with only a negligible drop in performance while still correcting all single bit errors in the instruction queue. We will also show that we can selectively drop instructions from the error checking logic and still obtain a 1–2 order of magnitude reduction in transient errors.

2. PROPOSED APPROACH

Instructions are typically read from on-chip instruction caches. The contents of these caches are most often protected with ECC bits stored along with the data. However, once the instruction is read from the cache and placed into the pipeline data structures, error protection is often no longer available. Instructions may reside in multiple pipeline data structures, including the instruction queue, reorder buffer, and branch predictor. Of these structures, we consider the instruction queue to be particularly vulnerable to soft errors because it is possible for instructions to reside here for hundreds of cycles before they are issued to an execution unit if their execution depends on the completion of long latency instructions. Furthermore, in [9] authors show that errors in the instruction queue have a very high probability of creating erroneous output. For these reasons, in this paper we will focus specifically on providing error protection for the instruction queue, although we can extend and modify our approach to other vulnerable structures as well.

Our approach is to store codewords in the instruction queue composed of the instruction itself and parity information obtained from the instruction. However, there are a number of factors that we must take into account in order to provide fault tolerance in a cost-

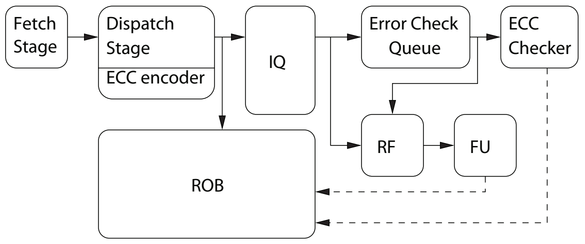


Figure 1: Pipeline flow

effective manner: performance impact, area cost, power cost, and expected error rates. More specifically, the error detection logic should be off the critical path of the main execution of the program so that it does not hamper the performance of the processor. In addition, we would like to minimize the amount of additional hardware and power dissipation required for ECC detection and correction. Thus, for example, we would prefer to have fewer ECC checkers than the width of the machine to reduce the area overhead and instantaneous power usage. Similarly, the error recovery latency should be limited so that error recovery does not add significantly to the program running time. Finally, the expected error rates and the criticality of the application introduce additional tradeoffs—allowing us to optionally not check some instructions while saving on the overall performance and power costs.

Taking these factors into account, we developed a hardware implementation for incorporating hardware error detection and recovery for the instruction queue into the processor. Our implementation requires an *error check queue* (ECQ) to be added as a buffer for instructions flowing between the issue and commit stages. This queue is designed to assist in detecting and correcting errors in instructions between the time they were first placed in the instruction queue to the time they are issued to functional units for execution. Varying the size of the queue as well as number of ECC checkers will trade-off performance and coverage penalties for additional hardware. These tradeoffs will be discussed further in Section 4. The flow of instructions through the pipeline using the ECQ is shown in Figure 1.

Before an instruction is stored in the instruction queue, ECC is generated over the opcode, source, destination register, and immediate fields and stored along with the instruction in the instruction queue. This can be done in parallel with decoding the instruction. When an instruction is ready to issue, it forwards its contents to both the register file as well as the ECQ. The register file will send operand values to the appropriate functional unit and the ECQ will eventually send the instruction to the error detection unit to recompute the ECC bits. Any mismatch with the stored and recomputed ECC bits indicates an error. Furthermore, an instruction will not be allowed to commit until the ECQ asserts that the instruction has no error. If the error detection unit found an error, the ECQ is responsible for reissuing the faulty instruction (after it has been corrected) as well as all instructions that were originally issued after it.

The idea behind our design is to insert instructions into the ECQ in a continuous stream as they are issued to execution units. While this would be quite straightforward for a single issue machine, this process is made more complicated for an out-of-order processor capable of issuing multiple instructions per cycle. To accomplish this, we follow the design shown in Figure 2.

We must guarantee there are enough free entries in the queue to accommodate the maximum allowable issue width for the processor before any new instructions can be issued to execution units (and forwarded to the ECQ). Also, since we may issue fewer instructions than the maximum allowable, the design includes a "collapsing buffer" such that instructions are first inserted into the buffer, realigned into groups of N instructions (where N is the width of the machine), and then forwarded to the error check queue up to

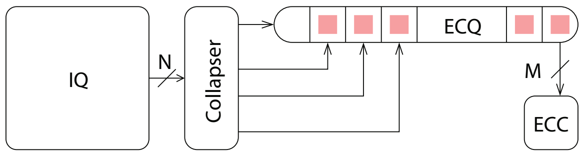


Figure 2: Streaming instruction ECQ design.

N instructions at a time. This scheme is similar to the collapsing buffer used in front-end fetch units designed to resolve misalignment problems as instructions are fetched from the instruction cache (see, for example [10]). Its main purpose is to allow for better utilization of the ECC checkers. Note that the ECQ is essentially a first-in-first-out structure (FIFO) and can be implemented as a circular queue; instructions are inserted starting at the tail of the queue and removed starting at the head. Each cycle, M instructions are removed from the ECQ and forwarded to the error detection unit, where M is the number of ECC checkers available in the unit. Note that M may be less than or equal to N. The actual ECC checking is done off the critical execution path of the machine and may be pipelined so as to relax timing requirements for the ECC circuitry.

If an instruction has been cleared by the error detection/correction unit, a signal is sent to the ROB that it is allowed to commit (i.e, it can be removed from the ROB provided it has completed execution and is the oldest instruction). If a fault is encountered, it is corrected and further issuing of new instructions is stalled until the instruction stream can be re-issued from the ECQ starting from the corrupted instruction. The instructions are issued one-at-a-time from the error check queue.

If an error is found while reissuing, the processor simply continues reissuing instructions starting from the new corrupted instruction. New instructions can begin issuing only after all instructions in the queue have completed execution. The number of cycles required to recover from a ECC error is a function of the number of instructions residing in the ECQ when the error is discovered and their execution latencies.

The scheme we have just presented is set up such that all instructions issued from the IQ are eventually checked for errors by the error detection unit (provided they are not identified as being on a mispredicted path first). If noise-induced error rates are relatively high, or manufacturing defects are present, then indeed, such rigorous checking of instructions can be justified. However, if error rates are relatively low and/or complete fault protection is not necessary, such extensive checking of all instructions may be too costly in terms of power, performance, or area. Instead, it may be more attractive to merely test for errors and detect/correct errors at a high enough rate so as to reduce error rates to an acceptable level.

Thus, we also propose a modified implementation such that instructions finish the error checking procedure only if the ECQ is not at its full capacity. Once the ECQ is full and there are new instructions to be inserted, the entries at the head of the ECQ will be "dropped" in order to accommodate the new instructions. The dropped instructions are effectively assumed to be error-free. Thus, instructions issued during periods of high instruction-level parallelism are more likely to be dropped. By dropping the instructions at the head of the ECQ we ensure that all instructions issued after an identified faulty instruction will be able to reissue. This scheme may also allow us to design the pipeline with fewer ECC checkers, and still obtain acceptable performance and reliability levels. These design issues will be discussed further in Section 4.

3. METHODOLOGY

We used a customized simulator derived from the SimpleScalar tool suite [11]. The simulator was modified to support statistical fault injection and detection. Statistical fault injection was imple-

Table 1: Simulated processor configurations.

Parameter	8-Wide Processor	4-Wide Processor
Inst. Window	64 IQ, 256 ROB, 64 LSQ	32 IQ, 128 ROB, 32 LSQ
Fetch Queue	32 instructions	16 instructions
Func. Units	6 ALUs + 2 mult/div 4 FP ALUs + 2 mult/div 3 Load/Store units	3 ALUs + 2 mult/div 3 FP ALUs + 1 mult/div 3 Load/Store units

mented by modeling particle strikes on the IQ where a particle can generate an error with a given probability. Note that in the baseline version of SimpleScalar, the instruction queue (IQ) and the reorder buffer (ROB) are implemented in a single structure, i.e., the RUU. We split the RUU into IQ and ROB components to allow us to obtain more realistic results. In addition, parity and ECC bits for each entry were added to the IQ along with a separate structure for detecting and correcting faults. We simulated both an 8-wide machine and a 4-wide machine, the configurations for which are reported in Table 1. For our simulations we selected a subset[1] of 16 benchmarks from the SPEC2000 integer and floating-point suite [12].

4. RESULTS

This section presents the results of experiments for both high- and low-fault environments. In the high fault environment case we used the error check queue in a more restrictive fashion, forcing every instruction to be checked. For the low-fault case, or when complete error protection is not necessary, if the error check queue is full when an instruction is issued then we make room for this instruction by dropping instruction at the head of the ECQ. Even though in this case complete error protection cannot be guaranteed for all instructions, our results show that enough instructions are checked so that the number of fault effects is reduced by up to two orders of magnitude. For each of these environments a 4-wide and an 8-wide microprocessor was considered.

For our simulations, we assumed a 0.01% single-bit error rate for the high-fault environment and 0.001% single-bit error rate for the low-fault case. We expect these error rates to become a reality in the future as devices scale and errors (especially those related to noise) increase. Using these high error rates also allows us to collect a statistically significant sample of data, as the number of instructions that can reasonably be run in our experiments is limited to approximately 100 million. We verified that the IPC drop remained consistent across a range of fault rates. Furthermore, for the instruction dropping scheme we found that the percentage of instructions dropped also remained consistent.

The initial analysis was performed with an ECQ sized comparably to the IQ, but the number of ECC checkers was varied up to the width of the machine. This corresponds to an ECQ with 32 entries for a 4-wide machine (and number of ECC checkers varying from 1–4), and an ECQ with 64 entries for an 8-wide machine (and number of checkers varying from 1–8). The results obtained from these experiments are shown in Figure 3(a). This graph presents the average IPC drop, grouped by either integer or floating point benchmark, as a function of the number of ECC checkers. Here we can observe that for the 4-wide integer simulations, having 2 ECC checkers corresponds to a 0.9% IPC drop while for the floating point benchmarks 3 ECC checkers are needed in the 4-wide machine for the drop in performance to be considered negligible. For the 8-wide machine 4 checkers are sufficient for the integer benchmarks as they give a 0.9% IPC decrease in performance, while for the floating point benchmarks we need 5 ECC checkers to have a tolerable 1% decrease in performance, on average.

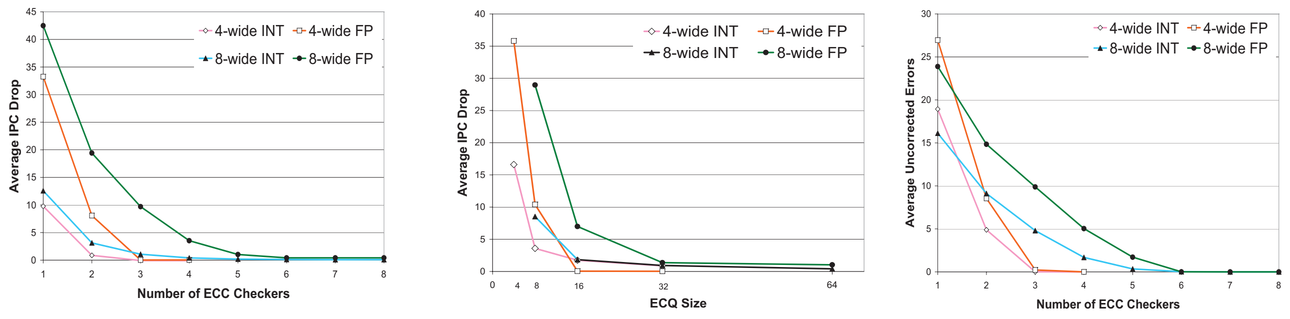
[1]The benchmarks we omitted were based on Fortran 90 code for which we don't have a compiler.

Once the optimal number of ECC checkers was established, the next analysis was aimed at determining how small the ECQ could be without significantly impacting performance. Thus, we started with the number of ECC checkers determined to be ideal in the previous experiment and varied the ECQ size. These results are presented in Figure 3(b). For the 4-wide machine running integer benchmarks with 2 ECC checkers, a 16-entry ECQ was sufficient, giving 1.7% performance degradation on average. In the case of floating point benchmarks we had already identified 3 ECC checkers as being required on a 4-wide machine. Using these 3 checkers, we once again found that the ECQ could be shrunk to 16 entries without significantly reducing performance. For an 8-wide machine and 4 ECC checkers, 32 entries is an appropriate ECQ size, limiting performance loss to only 0.9% for the integer benchmarks. Finally, for floating point simulations and the previously determined 5 ECC checkers, shrinking the queue to 32 entries still maintained an acceptable IPC with a drop of only 1.4%.

For the low-fault scenario we took the optimally sized ECQs and reran the simulations with varying numbers of error checkers. Note that for these simulations instructions currently being checked are dropped from the front of the ECQ when it is full and hence do not finish the error-checking procedure. This means that for the 8-wide simulations an ECQ of size 32 was used and checkers were varied in number from 1–8, while in the case of the 4-wide machine a 16-entry ECQ was used along with 1–4 checkers. Results are presented in Figure 3(c). In the case of the 8-wide processor, for integer benchmarks having 2 checkers reduces the amount of committed errors by an order of magnitude (compared to no error checking) while 3 ECC checkers are needed for the same level of protection in floating point simulations. In order to reduce the faults by two orders of magnitude 5 and 6 checkers are needed for integer and floating point benchmarks, respectively. For the 4-wide processor experiments 2 checkers are enough to reduce the number of faults by one order of magnitude, for both integer and floating point benchmarks, while 3 checkers give two orders of magnitude fault reduction for both integer and floating point benchmarks.

We note that while the results of our instruction-dropping scheme show that we can effectively reduce committed errors by 1–2 orders of magnitude even with selective error checking, this approach does not necessarily allow for fewer ECC checkers to be active without hurting performance. In particular, while the 8-wide processor went from requiring 4–5 ECC checkers with full fault protection to 2 checkers with an order of magnitude reduction in committed faults, for a 4-wide machine, about the same number of checkers are needed. So, from an area overhead perspective, there does not always seem to be an advantage of using selective error checking, even if error rates are low. On the other hand, if we also consider power implications, then selective error checking may still be advantageous. Since ECC checkers are implemented using XOR trees, we know that the switching activity for this logic will be high (XOR gates are guaranteed to toggle whenever any input changes). By preventing some instructions from flowing through the error checking datapath we are reducing the overall switching activity of this logic. For instance, for the 8-wide machine running integer benchmarks, we found that with only 2 checkers 16% of the instructions get dropped from the error checking logic. Similarly for floating benchmarks and 3 checkers, 18% of the instructions were dropped. We can approximate a similar amount of reduction in dynamic switching power from the ECQ and ECC checker since these instructions are never processed by these units.

Our last experiments tried to better quantify the cost of the hardware implementations by mapping our design onto a 250nm standard cell library obtained from Virginia Tech [13, 14]. The hard-



(a) (b) (c)

Figure 3: (a) Average IPC drop as a function of the number of ECC checkers, (b) Average IPC reduction as a function of ECQ size, (c) Percentage of faulty instructions committed with ECC to faulty instructions committed without ECC.

ware analysis was done for both an 8-wide processor with 5 checkers and a 4-wide processor with 3 ECC checkers. To accomplish this, we described an instruction queue (including all wakeup and select logic) and our ECQ design (including the checkers) in VHDL. The error checking buffer was sized at 16 entries for the 4-wide and 32 for the 8-wide processor. These designs were then synthesized and mapped using Synplicity's Synplify ASIC 3.3 tool. Our experiments indicated a 10.8% overhead in area for the 4-wide processor and a 8.3% overhead for the 8-wide design, compared to an instruction queue designed without the ECC-protection logic. This is a modest overhead considering the importance of the instruction queue as well as the size of the IQ relative to the entire processor.

5. FUTURE WORK

In the future, we plan to explore in more depth some additional choices with respect to our ECC implementation and the effect of dependencies. For example, we are investigating the potential advantages and disadvantages of issuing instructions from the instruction queue itself instead of the ECQ when an error is discovered. Using a single data structure for both issuing and error checking/correcting may have some design advantages, but may also limit the exposure of instruction-level parallelism due to the fact that post-issued instructions must stay in the instruction queue longer. We will also continue to investigate the types of errors that may occur and explore what impact different error types or error sources may have on a truly optimal implementation. In addition, we will obtain more accurate estimates of the hardware and power costs for our ECC technique.

6. CONCLUSION

We have presented two all-hardware implementations of an ECC-protected instruction queue in an out-or-order, multi-issue processor. Fault tolerance in the instruction queue is particularly important because errors in the opcode or register tags of an instruction are likely to propagate the error and cause incorrect operation of the program as a whole. The approach presented here is applicable to errors caused by a wide variety of sources—including transient faults due to radiation and signal noise as well as certain undetected manufacturing defects. Although double bit errors can be detected with our implementation, a reload of the instruction on a double bit error will be necessary.

We have shown that ECC can be implemented with the addition of a simple error check queue and with fewer checkers than the machine width without significantly impacting performance (easily held to less than 3%), thereby reducing the hardware and power cost of the technique. First estimates indicate that the additional hardware needed is only on the order of 10% of the size of the structures that are being protected. This estimate must also be considered in terms of the overall importance of the structure being protected, its size in relation to the entire microprocessor. In addition we can optionally reduce power further when error rates are low and a 1–2 order of magnitude reduction of committed faults is sufficient.

7. REFERENCES

[1] Robert Baumann. Soft errors in advanced computer systems. *IEEE Design and Test*, 22(3):258–266, May-June 2005.

[2] S. K. Reinhardt and S. S. Mukherjee. Transient-fault detection via simultaneous multithreading. In *Proc. of Intl. Symposium on Computer Architecture*, pages 25–36, June 2000.

[3] G.A. Reis, J. Chang, N. Vachharajani, S. Mukherjee, R. Rangan, and D.I. August. Design and evaluation of hybrid fault-detection systems. In *Proc. of Intl. Symposium on Computer Architecture*, 2005.

[4] T. J. Slegal, R. M. Averill, M. A. Check, C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, March-April 1999.

[5] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proc. of IEEE/ACM Intl. symposium on Microarchitecture*, pages 214–244, 2001.

[6] H. Ando, Y. Yoshida, A Inoue, I Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Knomoto, R. Yamashita, and H. Sugiyama. A 1.3GHz fifth generation SPARC64 microprocessor. In *IEEE Intl. Solid-State Circuits Conference*, 2003.

[7] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proc. of the Seventh Intl. Symposium on High-Performance Computer Architecture*, Munich, Germany, 2004.

[8] B. Gojman, V. Stojanovic, R. I. Bahar, and R. Weiss. Techniques for fault reduction in out-of-order microprocessors. In *International Workshop on Logic and Synthesis*, Lake Arrowhead, CA, 2005.

[9] 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. In *Proc. of the 36th Intl. Symposium on Microarchitecture*, pages 29–40, December 2003.

[10] T. Conte, K. Menezes, P. Mills, and B. Patel. Optimization of instruction fetch mechanisms for high issue rates. In *Proc. of Intl. Symposium on Computer Architecture*, pages 333 – 344, Santa Margharita Ligure, Italy, 1995.

[11] D. Burger and T. Austin. The simplescalar tool set. Technical report, University of Wisconsin, Madison, 1999. Version 3.0.

[12] John L. Henning. SPEC CPU2000: Measuring CPU performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.

[13] J. B. Sulistyo, J. Perry, and D. S. Ha. Developing standard cells for tsmc 0.25um technology under mosis deep rules. Technical report, Department of Electrical and Computer Engineering and Virginia Tech, November 2003.

[14] J. B. Sulistyo and D. S. Ha. A new characterization method for delay and power dissipation of standard library cells. *VLSI Design*, 15(3):667 – 678, 2002.