

# Techniques for Fault Reduction in Out-of-Order Microprocessors

Benjamin Gojman, Vladimir Stojanovic,  
R. Iris Bahar  
Division of Engineering, Brown University,  
Providence, RI 02912  
bgojman@cs.brown.edu,  
{vlada, iris}@lems.brown.edu

Richard Weiss  
Hampshire College, School of Cognitive  
Science, Amherst, MA 01002  
rweiss@hampshire.edu

## ABSTRACT

This paper addresses the issue of reducing transient faults that affect instructions while they are in the instruction queue waiting to be executed. Previous work has shown that for an in-order processor, squashing instructions triggered by a cache miss can reduce the number of transient faults. This paper shows that for an out-of-order processor, reducing the size of the instruction queue can have a bigger impact than more adaptive techniques such as fetch halting. Ongoing work will explore more effective techniques for selective fetch halting to provide a reduction in faults committed while having a minimal impact on performance.

## 1. INTRODUCTION

Fault tolerant architectures will prove to be vital to microprocessor design if we are to successfully take advantage of current technological advances in the field. It is estimated that the number of faults in individual devices will either remain constant or increase which implies higher error rates for the microprocessor as the density and number of devices grow. One type of error comes from transient faults which appear sporadically and currently arise from high energy particles that pass through a device and alter its state. As device sizes shrink and supply voltage levels are further reduced, noise margins are also reduced, thus making devices more susceptible to such transient faults.

In this paper we propose a way to reduce the number of transient faults experienced by an out-of-order microprocessor. The basic pipeline flow of an out-of-order processor is shown in Figure 1. Current out-of-order microprocessors tend to have a large instruction queue (IQ) and an aggressive fetch stage in order to supply the IQ and keep up with the out-of-order execution stage. The purpose of having a large IQ is to take advantage of instruction level parallelism (ILP). However, the vulnerability of such a structure to transient faults is proportional to the average occupancy by valid instructions waiting to be executed. Consider the case of instructions that depend on a load that misses to main memory, the latency of which is on the order of 200 cycles. These instructions will spend an order of magnitude more cycles in the IQ than most instructions spend in the entire pipeline. Therefore, we would expect them to have a much higher probability of experiencing a single bit transient fault. There are two types of approach to avoid this problem:

1. squash instructions from the queue when the load miss occurs and refetch them later, or
2. delay fetching them for most of the latency of the load miss.

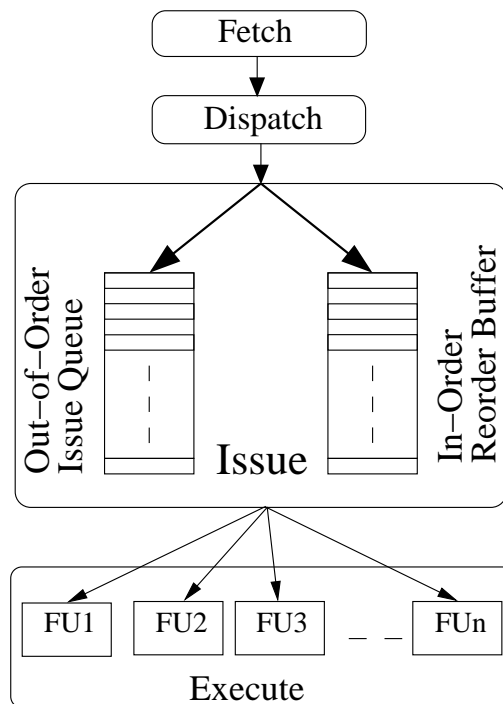


Figure 1: Modern microprocessor pipeline

This paper will focus on the latter. Within the latter class of solutions, we look at both fetch halting and reducing the size of the instruction queue.

Criticality of an instruction is a measure of whether or not delaying the completion of that instruction will impact the performance of the processor on a given application. While the latencies of arithmetic computations are measured in a few clock cycles, an access to main memory can take hundreds of cycles to complete. Wide super-scalar out-of-order processors can hide the latency of a memory access by continuing to issue instructions. However if the incomplete instruction has many dependencies, a CPU will likely be stalled for a long time waiting for data from memory. Nevertheless, for an out-of-order processor, not all long latency instructions will result in a stall. It is useful to identify such a “stalling instruction” as “critical” ahead of time. When this happens the culprit instruction can be marked as critical. This can be applied to in-order

processors as well.

When a long latency instruction is correctly predicted to be critical, fault reduction options can be applied. The dependent instructions can be squashed or the fetch unit can be halted or throttled. Simply reducing the size of the IQ will also have the effect of reducing vulnerability to faults, but it could have a greater impact on performance. In order to take advantage of instruction-level parallelism (ILP), the IQ is often sized very large even at the expense of a considerable amount of power. This paper examines the trade-offs between fault reduction and processor performance by varying IQ size and introducing fetch halting. The implementation of fetch halting is based on a combination of software and hardware prediction of criticality for load instructions.

In today's out-of-order machines with large instruction windows it is hard to correctly predict which instructions will be critical, and the delay that they will cause in computation. Also, the penalties for misprediction can be great. For a false positive, i.e. incorrectly predicted to be critical, the impact on average performance can be 2–75% with an average of 18% [1]. On the other hand, a false negative would increase the probability of an error significantly. In that work, a Bloom filter was used to accurately predict whether a load would miss in the L2 cache. Two figures of merit were used to predict given that a load would miss, whether it would be critical or not. These figures of merit were dead cycles (the number of cycles the processor would stall) and fetch-issue count (the number of instructions fetched and issued while the cache miss was unresolved). In [1] fetch halting based on these metrics was used to reduce the occupancy rates of the issue queue and reorder buffer by as much as 24%.

Past approaches to improving processor reliability have focused on error detection and correction [2–4]. Alternatively, other approaches have focused on fault prevention, e.g., [5]. In this paper, we extend the work done for an in-order Itanium-like processor by Weaver *et al.* [5] which proposes reducing the time valid instructions sit in vulnerable structures such as the instruction queue. They take the approach of squashing instructions and re-fetching them. They also look at loads which have a medium latency, i.e., a latency of 10–25 cycles as a result of an L1 or L2 cache miss. This means the refetching of squashed instructions can be initiated shortly after this event. We also extend ideas proposed in the work of [1] for keeping track of criticality in order to better tune when to halt instruction fetching. However, instead of using fetch halting for reducing power dissipation in the IQ, we propose using fetch halting to reduce the fault rate for committed instructions.

The rest of the paper is organized as follows. In Section 2 we discuss related work and the single bit error model. Section 3 provides a more detailed discussion of our proposed approach. Our experimental setup is presented in Section 4 followed by results given in Section 5. Finally, conclusions and future work are discussed in Section 6.

## 2. BACKGROUND

Currently, the main source of transient faults in microprocessors is energetic particles such as alpha particles and neutrons that create charge as they pass through a device. As particles strike, a device accumulates these charges and if enough charge gathers the state of the device may flip generating a single bit fault. However, this single bit fault does not necessarily affect the outcome of the computation. Figure 2 presents this idea more formally.

We observe that in only two out of the possible six outcomes does the single-bit fault affect the computation. The first is the case when an undetected error corrupts the output. The second case is when the error is detected yet it affects the output because no

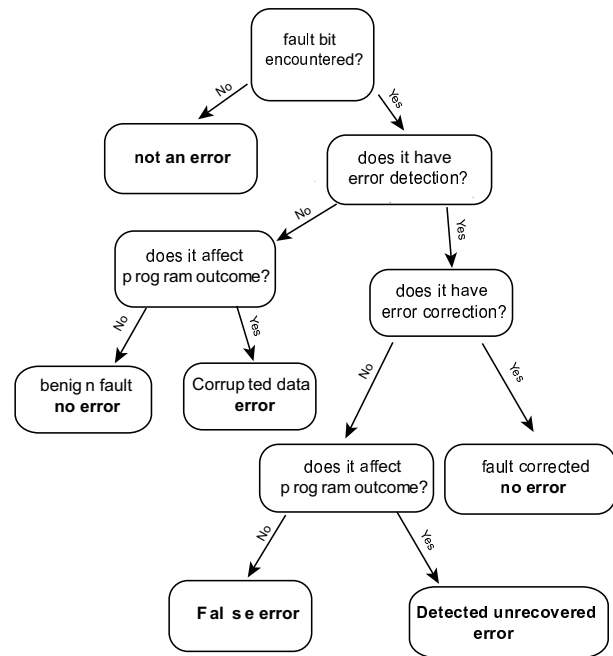


Figure 2: Possible effect of a faulty bit in a microprocessor, taken from [5].

correction or recovery is implemented.

In order to reduce the number of these errors that propagate through the computation, aggressive error detection and recovery, or error correction must be added to the system. However it has been shown that both approaches have a significant impact on power, performance and area. Thus, we propose an alternative way to lower the error rate.

Weaver *et al.* [5] propose a way to reduce, in an in-order machine, the amount of time instructions sit idle in the pipeline. They present a technique which they call a trigger and action mechanism with cache-misses being the trigger and instruction squashing being the action. Since in an in-order processor cache-misses are always critical, the processor can take advantage of this time to squash instructions waiting in the queue. Furthermore, since it is in-order, it is easy to determine which instructions have not been executed and thus can be safely squashed without significantly affecting performance. This is not necessarily the case for an out-of-order processor. In an out-of-order processor during a cache-miss, execution may continue if the instruction is not dependent on the load. Thus, indiscriminately squashing instructions waiting in the queue could greatly affect performance. Furthermore, implementing squashing within an out-of-order processor may significantly increase power consumption.

In [6] and [7] the notion of criticality is used for scheduling policies in clustered processors. Tune *et al.* in [7] optimize the performance by sending entire critical chains of dependent instructions through the same cluster. Fields *et al.* in [6] go one step further by scheduling for execution an instruction that is predicted to be critical before non-critical instructions. Therefore critical instructions will not contend for functional units. If an instruction is predicted to be critical both in [6] and [7] value prediction is used in order to break data-flow dependencies thus allowing dependent instructions to execute speculatively in hope of performance improvement. In more recent work [8], Fields *et al.* estimate slack, which is the number of cycles an instruction can be delayed without impacting

performance. They use this information to schedule instructions which have enough slack on slower functional units.

Several techniques for reducing the occupancy of the instruction queue are explored in this paper. One way of reducing occupancy is by fetch halting when we see that the instruction queue is stalling. We propose to implement this using a trigger-action pair where the trigger is a miss on the L1 or L2 data cache and the action is fetch halting. However, we observe that in an out-of-order machine, unlike in an in-order machine, not every miss will stall the pipeline. Therefore, simply halting on every miss will probably affect performance significantly. To address this issue, we augment the trigger to include a measure of how critical the instruction is. This second method is referred to as *selective halting* since upon a miss to the L1 or L2 cache we take into consideration heuristics developed in [1] before deciding to halt. The selective halting technique is effective when preserving the performance is of utmost importance. However if reducing the error rates and thus maintaining the correctness of the program execution is critical we have found that reducing the microprocessor resources is very effective.

### 3. PROPOSED APPROACH

As mentioned in Section 2, the key to reducing the frequency of single-bit faults affecting the computation is reducing the amount of time an instruction spends in the pipeline. That is, by reducing the time valid data spends in vulnerable structures we reduce the probability that the data gets corrupted and subsequently affects the outcome of the computation. In our proposed approach, we specifically aim at reducing the amount of time an instruction spends in the instruction queue. We consider the IQ a particularly vulnerable structure since 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.

The best way to prevent instructions from sitting in the IQ for a long period of time depends on the machine configuration and the application being run on the machine. As was shown in the work of [5], for an in-order processor such as the Itanium, squashing all instructions following a long latency load instruction (i.e., a load that misses in the L1 cache) and refetching them once the load complete, is a reasonable strategy.

For an out-of-order processor, the instruction queues tend to be significantly larger than those of in-order processors in order to gain the most performance advantage. The main reason is to expose more instruction-level parallelism inherent in a particular program. However, by doing so, this consequently sets up situations where a large number of older instructions are left to sit idly in the IQ waiting for dependent instructions to complete while newer instructions can jump ahead of them and execute earlier. Overall, the idea is to keep the execution units busy and allow the application to complete earlier, but at a cost of greater vulnerability to errors.

Probably the easiest way to reduce instruction latency (i.e., the amount of time an instruction spends in the IQ), is to reduce the size of the IQ itself. This has obvious repercussions, namely reduced performance, but only for applications that require a larger IQ in order to keep the execution units busy. On the other hand, we can justify some drop in performance if this is accompanied by a significant drop in the likelihood that a single bit fault will affect the outcome of a program.

If a program's performance is particularly sensitive to instruction queue size, then a more sophisticated means of reducing instruction latency is required. Simply halting the fetch unit whenever the processor detects that a load instruction will, or has already, missed in the cache may be reasonable. But again, if the program has enough ILP such that other instructions may execute while the

miss is being serviced, then performance will still suffer. A more conservative approach will apply selective fetch halting only when it is determined that a trigger event (i.e., a load miss) will cause the execution units to remain idle, effectively stalling the pipeline.

It is obvious that having different approaches to reducing instruction latency in the IQ will give us overall the best results. That is, based on the particular application and its execution profiling, we should be able to choose reducing the size of the IQ, fetch halting on every load miss, or selective fetch halting, to give us the best overall ratio.

In order to use selective fetch halting, instruction criticality has to be measured first. The process of identifying critical static instructions starts with software profiling. Statistics are first collected for individual instructions from profiling runs of applications using training input sets. Individual instructions are then marked as critical depending on the values of these statistics. Finally, these instructions are annotated within the program code using annotation bits, and the program is then ready to be executed with the reference input set.

During the fetch stage, if a newly fetched instruction is a load, an oracle is accessed to determine if it will miss in the L2 cache. Alternatively, we could use a Bloom filter, as was done in [1], for a more realistic predictor. Once a load instruction that is predicted to miss reaches the instruction stage, if it is marked as critical through software profiling a control signal is sent to the fetch unit such that instruction fetch is suspended starting on the next cycle. Fetching resumes when the load instruction completes or is squashed if it is a wrong-path instruction.

In this work two profiling heuristics for measuring a load instruction's criticality were considered: *dead cycles* and *fetch-issue count*. The profiling phase collects statistics for these two heuristics for each load instruction. Dead cycles are defined as cycles during which the processor cannot issue any instructions due to unresolved dependencies. The number of dead cycles is counted for each cache miss; this simple count is proportional to the load instruction's criticality. Fetch-issue count is a more specific measurement of performance and criticality. It counts the number of instructions that are fetched and subsequently issued during a miss to memory. A load instruction with a very low fetch-issue count is critical because there is minimal instruction flow when the instruction misses.

### 4. METHODOLOGY

We used a customized simulator derived from the SimpleScalar suite [9]. The simulator was modified to support software profiling of criticality, fetch halting, and statistical fault injection and detection. Note that in the baseline version of SimpleScalar the IQ and reorder buffer are implemented in a single structure, i.e., the RUU. While splitting the RUU into IQ and ROB components will allow us to obtain more realistic results, the general trends should remain the same. We simulated an 8-wide machine, the configurations for which is presented in Table 1.

For our simulations we selected a subset of the SPEC CPU2000 integer and floating-point benchmarks [10]. We chose a subset that would demonstrate a range of effectiveness for fault reduction<sup>1</sup>

Software profiling was performed on the training input datasets whereas actual results were attained with the reference datasets. Each benchmark was simulated for 100M instructions after fast-

<sup>1</sup>The benchmarks we omitted were either based on Fortran 90 code (for which we don't have a compiler) or showed zero effect from the application of fetch halting, due to negligible cache miss rates.

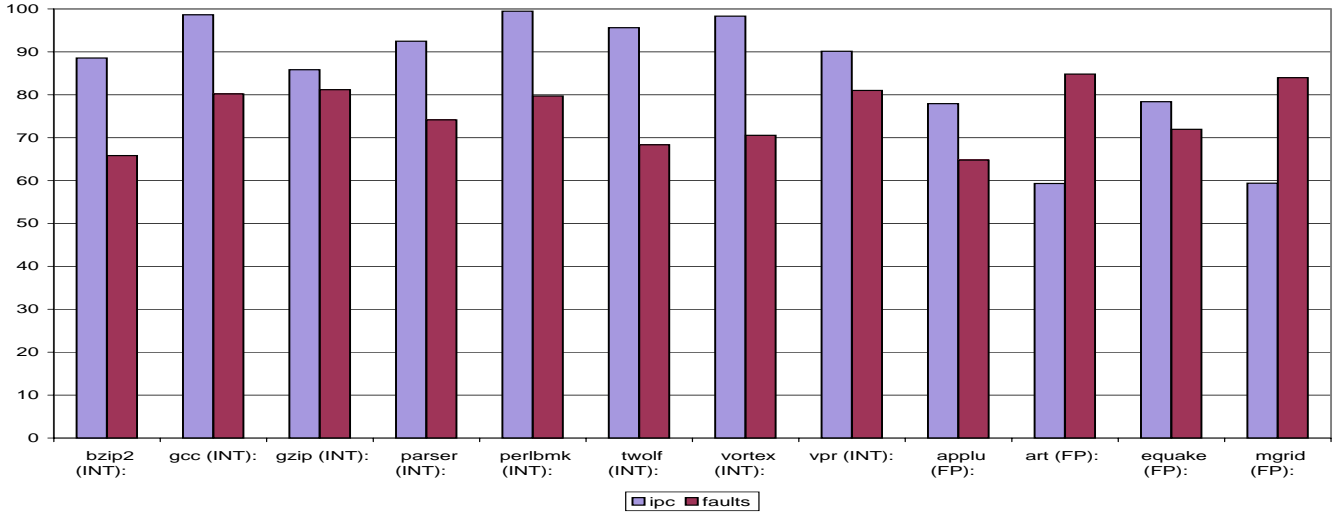


Figure 3: Effects of reduced queue size on performance and faults

Table 1: 8-Wide Simulated processor configuration.

|                  |   |
|------------------|---|
| Inst. Window     | 128-entry RUU, 128-entry LSQ  |
| Machine Width    | 8-wide decode, issue and commit   |
| Fetch Queue      | 32 instructions   |
| Functional Units | 6 ALUs + 2 mult/div<br>4 FP ALUs + 2 mult/div/sqrt<br>3 Load/Store units        |
| L1 Icache        | 16kB 1-way; 32B lines; 1 cycle  |
| L1 Dcache        | 16kB 1-way; 16B line; 1 cycle   |
| L2 Cache         | 128kB 8-way; 32B line;<br>15/20 Icache/Dcache cycles                            |
| Memory           | 64-bit wide;<br>180 cycles first; 2 inter                                       |
| Branch Pred.     | 2k 2-level + 2k bimodal<br>2k meta 2 preds/cycle,<br>3-cycle mispredict penalty |

forwarding through some number of instructions to avoid simulating startup effects. These fast-forward counts were partially influenced by [11] and [12] and were otherwise obtained by simulating through both the train and reference datasets until some fixed breakpoint in the benchmark. This ensured that both profiling and reference runs were simulating the same part of the program. We embed the information obtained from software profiling back into the profiled application by annotating the programs compiled assembly instructions and then reassembling these modified source files.

Statistical Fault Injection was implemented by modeling particle strikes on the RUU where a particle can generate an error with a given probability. For our simulations we set that probability to be .001%

## 5. RESULTS

This section describes the results obtained from three different techniques used to reduce faults. These are fault reduction based on hardware optimization, always halting based on L2 cache miss or either L1 or L2 miss, and selective halting based on instruction criticality. The techniques we propose were applied to an 8-wide

machine described in detail in Table 1.

From our results, we observed that the fault rate is directly proportional to the size of the instruction queue. Therefore, simply reducing its size offers fault prevention. In Figure 3 we show performance and fault reduction results for a 64-entry instruction queue (i.e., RUU from Table 1) relative to the 128-entry basecase. For each bar, the percentage below 100 represents the percentage drop in either performance (measured in terms of instructions per cycle), or observable faults. As expected, this technique does reduce faults; however, although the ROB was cut in half, the largest fault reduction is only 35% (for *applu*). Furthermore, half of the benchmarks suffered a significant reduction in performance. This is especially true with floating point benchmarks. We attribute this to low instruction-level parallelism (ILP) exposed due to the small IQ. Floating point benchmarks suffer the most in terms of performance since these applications tend to have fewer control and data dependencies.

In order to take advantage of high instruction-level parallelism within an application, we need a larger instruction queue. However, since the larger queue implies a higher fault rate, as was apparent from the previous results, we need another approach to reduce the fault rate. To start, we first propose a relatively straightforward approach where we halt instruction fetching every time we encounter a miss to the L2 cache or a miss to either L1 or L2 caches. We call this technique *halt always*. The results for performance, as measured in terms of instructions per cycle (IPC), and fault reduction when applying *halt always* are shown in Figures 4 and 5 respectively. The results show some improvement in performance for the benchmarks that suffered most from reduced instruction queue size. In particular, *applu* and *equake* had their performance improved by 22 and 10% respectively. Unfortunately, this improvement was not apparent for all benchmarks; *mgrid* still exhibits an IPC loss of up to 35%. The *mgrid* benchmark has a relatively high cache miss rate so fetch halting is activated quite often in this case. This somewhat aggressive approach of always halting the instruction fetch unit every time a cache miss is detected may not be appropriate for these types of benchmarks. In particular, we observe that not every cache miss may lead to a pipeline stall, thus causing fetch halting to severely impact performance. This observation leads us to the application of our final approach, *selective fetch halting*.

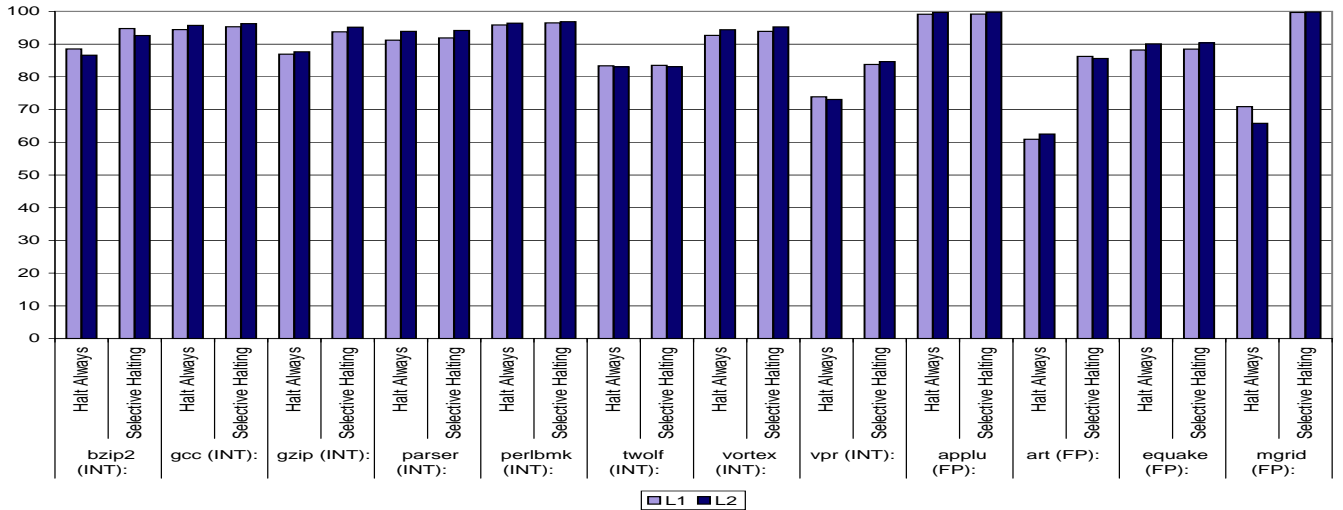


Figure 4: Relative IPC (compared to baseline) for four different runs: always halting on L2 cache miss or either L1 or L2 miss, selective halting on L2 cache miss or either L1 or L2 miss

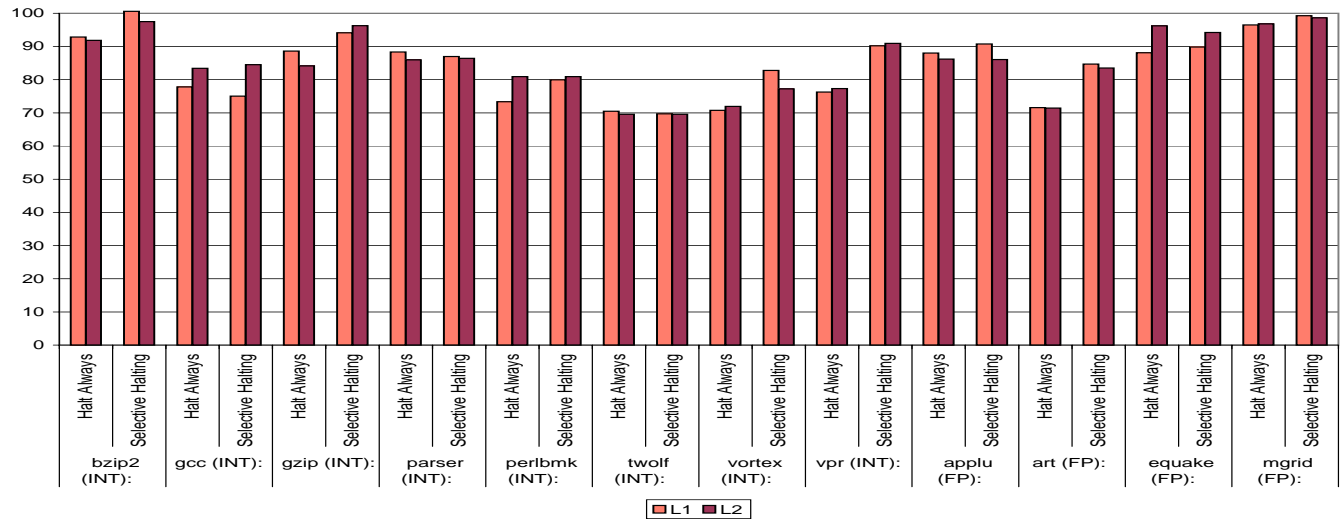


Figure 5: Relative Faults (compared to baseline) for four different runs: always halting on L2 cache miss or either L1 or L2 miss, selective halting on L2 cache miss or either L1 or L2 miss

In order to apply selective fetch halting, we need a mechanism to identify load instructions that actually lead to a pipeline stall (i.e., those that stall the issuing and execution of new instructions when they produce a cache miss). For this purpose, we use the profiled code as described in Section 3. Again, Figures 4 and 5 show the effects selective halting have on IPC and faults. As we can see from the results, nearly all benchmarks have less than a 10% drop in IPC when selective halting is employed.

Although always halting and selective halting help retain performance, they have a mixed effect on fault reduction as compared to hardware optimization (i.e., cutting the instruction queue in half). In the case of *art* we see a greater fault reduction, however when we examine *mgrid* we see nearly no reduction in faults.

The results obtained from our experimental runs suggest that the best mechanism for reducing committed faults largely depends on the particular application. However, many benchmarks show a greater reduction in total committed faults than a reduction in IPC

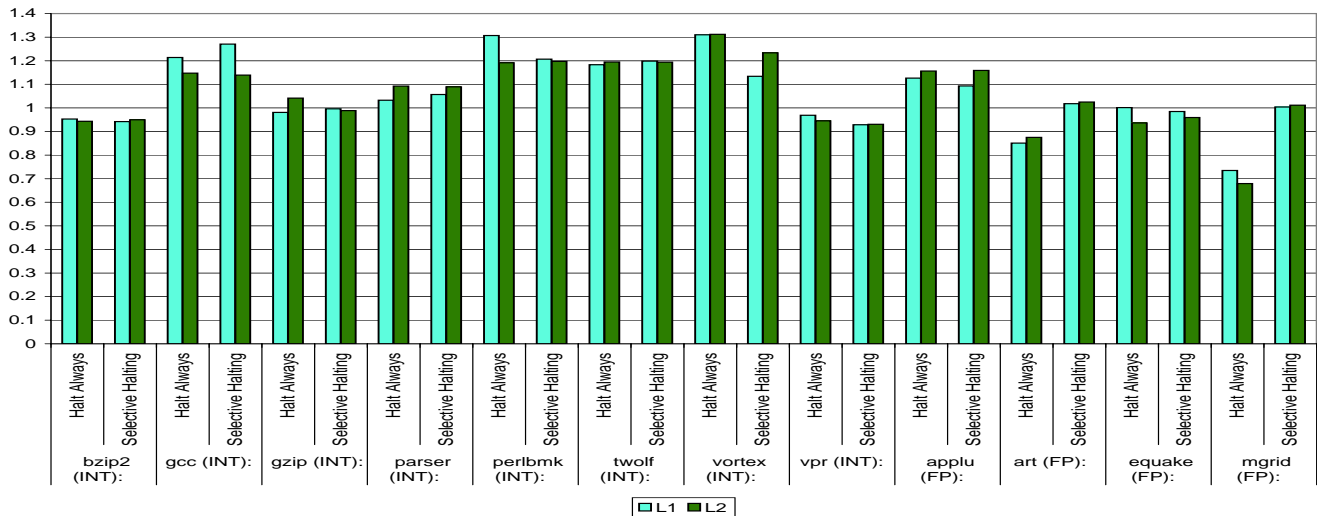
for all three of our fault reduction mechanisms. To measure the value of each of our approaches, we use the following formula:

$$Q_f = \frac{1 - \Delta IPC}{1 - \Delta FaultsCommitted}$$

If the ratio,  $Q_f$ , is greater than 1 then we consider the technique useful. These results are plotted in Figure 6. Though some of the benchmarks show a ratio of less than 1 (e.g., *bzip2* and *vpr*), most prove to be useful when using either *always halting* or *selective halting*.

## 6. CONCLUSIONS AND FUTURE WORK

Reliable computing requires that processors include some means of detecting faults or reducing the rate of soft errors. In this paper, we focus on the latter approach by trying to reduce the time instructions spend in structures vulnerable to single bit faults. In particular, this study targets the instruction queue. We present three



**Figure 6:**  $Q_f$  factor for four different runs: always halting on L2 cache miss or either L1 or L2 miss, selective halting on L2 cache miss or either L1 or L2 miss

techniques that help reduce the time instructions wait in the IQ. The first approach uses fetch halting whenever there is a miss in L1 or L2 cache. We use a measure of the criticality of an instruction in order to selectively halt the fetch unit in our second technique. Finally we scale the size of our IQ to reduce the number of instructions exposed to faults.

We have shown that reducing the size of the instruction queue can have a significant impact on the committed fault rate. On average, the number of faults was reduced by 33%, with a corresponding reduction in IPC of 13%. While this reduction in IPC is not huge, it is significant and it is therefore necessary to consider how best to trade off performance for improvements in fault rates. This issue has led us to explore other techniques that selectively halt the fetch unit in order to prevent instructions dependent on long latency loads from spending many cycles in the instruction queue. We have found that although our approaches of always halting and selective halting the fetch unit whenever a cache miss is detected help maintain performance, they have a mixed effect on fault reduction as compared to hardware optimization. This observation does lead to directions for future work. In particular, we plan to consider alternative metrics for criticality to better guide fetch halting. In addition, it appears that halting on L1 misses may be too aggressive if preserving performance is crucial. Finally, we would like to investigate instruction squashing as an alternative to fetch halting. By squashing and later refetching an instruction from the cache, we may have better control over preventing only stalled instruction from sitting in the instruction queues for long periods of time. In addition, it also provides a means for repairing instruction that have acquired a fault.

## 7. REFERENCES

- [1] N. Mehta, B. Singer, R. I. Bahar, M. Leuchtenburg, and R. Weiss. Fetch halting on critical load misses. In *Proceedings of the IEEE International Conference on Computer Design*, pages 244–249, San Jose, CA, 2004.
- [2] Chris Weaver and Todd Austin. A fault tolerant approach to microprocessor design. In *Proceedings of Dependable Systems and Networks*, July 2001.
- [3] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient fault recovery via simultaneous multithreading. In *Proceedings of the International Symposium on Computer Architecture*, pages 87–98, May 2002.
- [4] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of Fault Tolerant Computing Systems*, pages 84–91, June 1999.
- [5] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, Munich, Germany, 2004.
- [6] B. Fields, S. Rubin, and R. Bodik. Focusing processor policies via critical-path prediction. In *Proceedings of the 28th annual international symposium on Computer architecture*, pages 74 – 85, Goeteborg, Sweden, 2001.
- [7] E. Tune, D. Liang, D. M. Tullsen, and B. Calder. Focusing processor policies via critical-path prediction. In *Proceedings of the Seventh International Symposium on High-Performance Computer Architecture*, 2001.
- [8] B. Fields, R. Bodik, and M. Hill. Slack: Maximizing performance under technological constraints. In *Proceedings of the 29th annual international symposium on Computer architecture*, pages 47 – 58, Anchorage, Alaska, 2001.
- [9] D. Burger and T. Austin. The simplescalar tool set. Technical report, University of Wisconsin, Madison, 1999. Version 3.0.
- [10] John L. Henning. SPEC CPU2000: Measuring cpu performance in the new millennium. *IEEE Computer*, 33(7):28–35, July 2000.
- [11] Suleyman Sair and Mark Charney. Memory behavior of the SPEC2000 benchmark suite. Technical Report RC-21852, IBM Thomas J. Watson Research Center, October 2000.
- [12] Timothy Sherwood, Erez Perelman, Greg Hamerly, and Brad Calder. Automatically characterizing large scale program behavior. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2002.