

ORBIT: Effective Issue Queue Soft-error Vulnerability Mitigation on Simultaneous Multithreaded Architectures using Operand Readiness-based Instruction Dispatch

Xin Fu, Tao Li and José Fortes

Department of ECE, University of Florida
xinfu@ufl.edu, taoli@ece.ufl.edu, fortes@acis.ufl.edu

Abstract

With the advance of semiconductor processing technology, soft errors have become an increasing cause of failures of microprocessors fabricated using smaller and more densely integrated transistors with lower threshold voltages and tighter noise margins. With diminishing performance returns on wider issue superscalar processors, the microprocessor design industry has opted for using simultaneous multithreaded (SMT) architectures in commercial processors to exploit thread-level parallelism (TLP). SMT techniques enhance overall system performance but also introduce greater susceptibility to soft errors - concurrently executing multiple threads exposes many program runtime states to soft-error strikes at any given time. The issue queue (IQ) is a key microarchitecture structure to exploit instruction-level and thread-level parallelism. On SMT processors, the IQ buffers a large number of instructions from multiple threads and is more susceptible to soft-error strikes.

In this paper, we explore the use of operand-readiness-based instruction dispatch (ORBIT) as an effective mechanism to mitigate IQ soft-error vulnerability on SMT processors. We observe that IQ soft-error vulnerability is largely affected by instructions waiting for their source operands. The overall IQ soft-error vulnerability can be effectively reduced by minimizing the number of waiting instructions and their residency cycles in the IQ. We develop six techniques that aim to improve IQ reliability with negligible performance degradation on SMT processors. Moreover, we extend our techniques with prediction methods that can anticipate the readiness of source operands ahead of time. The ORBIT schemes integrated with reliability-awareness and readiness prediction achieve more attractive reliability/performance trade-offs. The best of the proposed schemes (e.g. Predict_DelayACE) reduces IQ vulnerability by 79% with only 1% throughput IPC and 3% harmonic IPC reduction across all studied workloads.

1. Introduction

With the advance of semiconductor processing technology, soft errors have become an increasing cause of failures on microprocessors fabricated using smaller and more densely packed transistors with lower threshold voltages and tighter noise margins [1]. Soft errors, also known as single event upsets (SEUs), are caused by high-energy (more than 1 MeV) neutrons from cosmic radiation, alpha particles emitted by trace uranium and thorium impurities in packaging materials and low-energy cosmic neutron interactions with the isotope boron-10 (^{10}B) in IC materials used to form insulator layers in manufacturing. SEUs corrupt a data bit stored in a device until new data is written to that device. As a result of the continuing scaling of semiconductor technology down to the nano-scale and increasing complexity of microprocessors, soft-error rates of future generations of processors are projected to increase significantly.

Methodologies to tolerate transient faults at lower levels exist, for example, radiation-hardening techniques [2] can be used

to reduce the likelihood of SEUs. However, it is too expensive or impractical to apply these techniques to protect every device in commercial chips that contain billions of transistors. Protection techniques such as parity or ECC are used in memory and cache design. However, the pipeline structures (e.g. issue queue and reorder buffer) are latency-critical and need to handle frequent accesses in a single cycle. These protection techniques can add latency to each access and degrade the performance. For instance, studies in [37] investigated the performance effect of protecting the issue queue (IQ) with ECC and showed that such a modification can result in up to 45% performance degradation. Other approaches [3, 4, 5] use duplicated coarse-grained structures such as functional units, processor cores or hardware contexts to tolerate transient faults. These approaches can result in significant overheads in performance, power, area, and design time if they are used to protect all microprocessor structures. Recently there has been work on modeling and the analysis of the soft-error vulnerability of microarchitecture components [1, 6, 7, 8, 9, 10, 11]. Several studies [6, 9, 12] have observed that a significant fraction of soft errors can be masked at the microarchitecture level, making soft-error tolerance through reliability-aware microarchitecture design a cost-effective approach. Moreover, at this level, program execution characteristics can be exploited to implement application-oriented fault-tolerant mechanisms. These advantages become more attractive for general-purpose processor designs where the realistic goal is to increase robustness (instead of trying to provide total immunity) to soft errors.

With diminishing performance returns on wider issue superscalar processors, the microprocessor design industry has opted for using simultaneous multithreaded (SMT) architectures in commercial processors [13, 14] to exploit thread-level parallelism (TLP). SMT architectures improve the performance of a superscalar processor by dynamically sharing pipeline and microarchitecture resources among multiple, concurrently running threads. SMT techniques enhance overall system performance but also introduce greater susceptibility to soft errors - concurrently executing multiple threads exposes many program runtime states to soft-error strikes at any given time.

In a dynamically scheduled SMT processor, the issue queue (IQ) is a key microarchitecture structure used to exploit instruction-level and thread-level parallelism. The IQ holds decoded instructions from multiple threads until they can be executed. In SMT processors that exploit ILP and TLP, the IQ buffers a large number of instructions, making it more vulnerable to soft-error strikes. Using a reliability-aware architecture simulator, we profiled soft-error vulnerability of several key SMT microarchitecture structures. Results (see Figure 1) are shown in the form of the architecture vulnerability factor (AVF) of a hardware structure which estimates the probability that a transient fault in that hardware structure will result in incorrect program results. Details of our simulation framework, machine configuration, evaluated workloads and metrics are presented in

Section 4. Among the studied microarchitecture structures, the IQ exhibits the highest vulnerability. This suggests that the IQ is a reliability hot-spot in SMT processors.

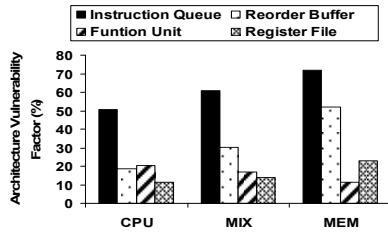


Figure 1. Microarchitecture soft-error vulnerability profile on SMT processor

In this paper, we explore using operand readiness-based instruction dispatch (ORBIT) as an effective mechanism to mitigate IQ soft-error vulnerability on SMT processors. We observe that the IQ soft-error vulnerability is largely affected by instructions waiting for their source operands. The overall IQ soft-error vulnerability can be effectively reduced by minimizing the number of waiting instructions and their residency cycles in the IQ. The main contributions of this paper are as following:

- We propose six techniques that improve IQ reliability with negligible performance degradation in SMT processors. The best scheme (e.g. *Predict_DelayACE*) reduces the IQ vulnerability by 79% with only 1% throughput IPC and 3% harmonic IPC reduction on average across all of the workloads.
- We propose using off-line instruction vulnerability profiling information to further identify reliability critical instructions. We extend our techniques with easy-to-implement prediction methods which can anticipate the ready time of source operands ahead of time. The ORBIT schemes integrated with reliability-awareness and readiness prediction achieve better reliability/performance trade-offs.
- We compare our proposed techniques with existing mechanisms (e.g. *2OP_BLOCK* [15]) that exploit operand availability for SMT performance optimization. Results show that our techniques outperform *2OP_BLOCK* by additionally reducing IQ vulnerability by 15% with similar performance overhead. We further compare the proposed techniques with advanced SMT fetch polices (e.g. FLUSH) which exhibit a superior soft-error mitigation capability, and it shows that our techniques achieve better reliability/performance tradeoffs by improving both IQ reliability by 46% and harmonic IPC by 6%. The vulnerability impact of the techniques on the entire processor core and other core structures is also evaluated.

The rest of this paper is organized as follows. Section 2 provides a background on soft-error vulnerability computation at the microarchitecture level. Section 3 presents our operand readiness based instruction dispatch for IQ soft-error mitigation. Section 4 describes our experimental setup. Section 5 evaluates the proposed techniques in terms of reliability enhancement and performance overhead. We discuss the related work in Section 6 and summarize our work in Section 7.

2. Microarchitecture Soft-error Vulnerability Computation

Several techniques have been proposed to model processor vulnerability to soft error at the microarchitecture level. In [9], Li and Adve estimated reliability using a probabilistic model of the

error generation and propagation process in a processor. In the past, statistic fault injection [12, 16] has also been used to evaluate architectural reliability. In this work, we estimate the reliability of processor microarchitecture using Architectural Vulnerability Factor (AVF) analysis methods introduced in [6, 7]. In this section, we briefly describe the AVF computation methods to provide sufficient background for the rest of the paper.

A hardware structure’s AVF refers to the probability that a transient fault in that hardware structure will result in incorrect program results. The overall hardware structure’s error rate is decided by two factors: the device raw error rate, mainly determined by circuit design and processing technology, and the AVF. The AVF can be used as a metric to estimate how vulnerable the hardware is to soft errors during program execution. To compute AVF, one needs to classify hardware states into bits that store information that can affect the final program output and those that can not. The processor state bits required for architecturally correct execution (ACE) are called ACE bits [6]. In a given cycle, the AVF of a hardware structure is the percentage of ACE bits that the structure holds. The AVF of a hardware structure is derived by averaging the AVFs of the structure across program execution. In reality, it is much more feasible to identify un-ACE bits, the processor state bits that do not affect correct program execution. Examples of locations that contain un-ACE bits include NOPs, idle or invalid states, wrong-path instructions, dynamically dead instructions and data, and cache lines that will not be accessed before eviction. Instructions whose computation results affect the program final outcome are defined as ACE instructions, whereas un-ACE instructions do not affect the program correct execution. Note that un-ACE instructions also contain ACE-bits (e.g. opcode). Through cycle-level simulation, processor microarchitecture states are classified into ACE/un-ACE bits and their residency and resource usage counts are generated. This information is then used to compute the AVF of various hardware structures.

3. Operand Readiness-based Instruction Dispatch (ORBIT) for IQ Soft-error Vulnerability Reduction

In this section, we explore the use of instruction operand-readiness information to effectively mitigate soft-error vulnerability of the IQ on SMT architectures.

3.1. IQ Soft-error Mitigation through Instruction Dispatch

As described in Section 2, microarchitecture soft-error vulnerability is determined by the quantity and residency cycles of the ACE-bits in a structure. In a dynamic-issue, out-of-order execution microprocessor, a dispatched instruction after the decode stage will stay in the IQ until all of its source operands are ready and the appropriate functional unit is available. An instruction IQ residency time can be broken down into cycles during which the instruction is waiting for its source operands and cycles during which the instruction is ready to execute but is waiting for an available function unit. An instruction in the IQ can be classified as either a waiting instruction or a ready instruction, depending on the readiness of its source operands. Both waiting instructions and ready instructions affect the IQ soft-error susceptibility. Figure 2 (a) shows the IQ AVF contributed by waiting instructions and ready instructions across three types of workloads (shown as Table 4) on the studied SMT processor (shown as Table 3). As IQ AVF is determined by the number of vulnerable instructions per cycle and instruction

residency cycles in IQ, Figure 2 (b) and (c) depict the quantity and residency cycles of waiting instructions and ready instructions in the IQ. Since ACE instructions are the major source of ACE bits, Figure 2 (b) and (c) also profile the number of waiting and ready ACE instructions and their residency cycles.

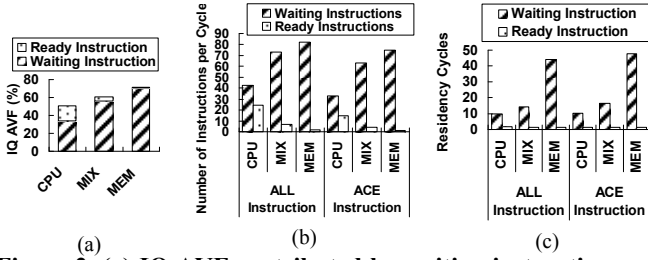


Figure 2. (a) IQ AVF contributed by waiting instructions and ready instructions, profiles of (b) the quantity and (c) residency cycles of ready instructions and waiting instructions in a 96 entries IQ. The ICOUNT fetch policy is used.

As Figure 2 (a) shows, on an average, waiting instructions contribute to 86% of the total IQ AVF. Figure 2 (b) and (c) help to explain the high AVF contribution from waiting instructions. As can be seen, waiting instruction residency time in the IQ ranges from 10 to 48 cycles, whereas ready instructions usually spend 1.5 cycles in the IQ on average. This suggests that an instruction can spend a significant fraction (91% on average) of its IQ residency cycles waiting for source operands that are being produced by other instructions. Previous studies [15] have also observed that instructions usually spend most of their IQ residency cycles waiting for their operands to be ready. Nevertheless, no attempt has been made to exploit operand readiness in reliability optimizations. Figure 2 shows that, at every cycle, the number (61 on average) of waiting instructions also overwhelms that (9 on average) of ready instructions. Especially on MEM workloads that exhibit higher cache miss rates, most instructions are congested in the IQ waiting for ready operands - both the quantity and residency cycles of waiting instructions greatly surpass those of ready instructions. As a result, waiting instructions contribute to 98% of the total IQ AVF. Furthermore, as Figure 2 (b) and (c) show, waiting ACE instructions also play a much more important role than ready ACE instructions in determining the IQ AVF due to their higher quantity and longer IQ residency. In short, in order to mitigate IQ AVF, we should focus on the waiting instructions. IQ residency cycles can be minimized if instructions are dispatched into the IQ with ready operands; meanwhile, the number of waiting instructions is also reduced because when instructions are dispatched they are ready-to-execute directly. Therefore, dispatching instructions into the IQ only when their operands are ready can effectively control both the quantity and residency of waiting instruction in the IQ and reduce IQ AVF significantly. If the ready instructions can be dispatched to the IQ in a timely fashion for execution, the performance impact will be negligible.

In typical processors, resources (a ROB entry, an IQ entry, a LSQ entry and so on) are allocated at the dispatch stage, and instructions are dispatched simultaneously to those resources. To effectively reduce instruction waiting cycles in the IQ, we propose the baseline ORBIT scheme (*DelayALL*) which delays the dispatch of not-ready-to-execute instructions into the IQ. Therefore, instruction dispatch completes in two steps: resource allocation and instruction dispatch into other structures perform normally without any delay; instructions will be dispatched into

the IQ later when they become ready-to-execute. Note that the allocated IQ entry will be reserved until the instruction finally moves into the IQ. Figure 3 describes the ORBIT architecture overview. In order to obtain their operands readiness when the instructions are sitting in the ROB, a multi-banked, multi-ported array is built to record the register files' readiness state. The bit array is updated during write back stage. The ROB can be logically partitioned into several segments to allow parallel accesses to the multiple banks of the array which hold the same copies of information. A simple AND gate is added in each ROB entry to determine the readiness of an instruction. Note that in *DelayALL* scheme, younger instructions can still be dispatched if their source operands are ready and this does not affect the correctness of program execution since instructions are still committed in order.

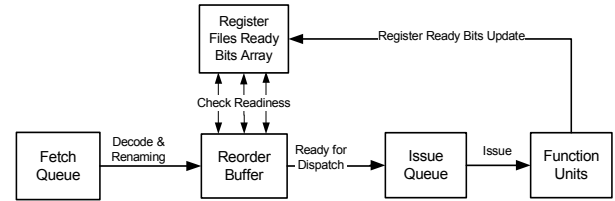


Figure 3. ORBIT architecture overview

3.2. ACE Instruction-aware ORBIT

The primary goal of ORBIT is to reduce the instruction waiting cycles in the IQ. Compared with un-ACE instructions, ACE instructions are the major source of ACE-bits since bits in an ACE instruction are ACE while an un-ACE instruction only contains a small portion of ACE-bits (e.g. opcode). Based on this observation, we propose applying operand readiness based instruction dispatch only to ACE instructions (*DelayACE*). Compared with *DelayALL*, the *DelayACE* can achieve better performance since it does not block the dispatch of un-ACE instructions (31% instructions in SPEC CPU2000 workloads are un-ACE) which are not critical to reliability.

The *DelayACE* scheme relies heavily on the capability of identifying instructions that are vulnerable to soft error strikes. However, in general, a retired instruction cannot be classified as ACE or un-ACE until a large number of its subsequent instructions have graduated. In [6], a post-graduate instruction analysis window with a size of 40,000 instructions is used for vulnerability classification. The SlicK table was proposed in [33] to dynamically identify ACE instructions by taking advantage of the time lag between the leading and trailing threads. This method, however, it is not adaptable to our study. To perform just-in-time vulnerability identification at a per-instruction level, we propose performing instruction vulnerability characterization offline and extending the ISA to encode the 1-bit vulnerability tag (e.g. ACE or un-ACE). When an instruction is decoded, its vulnerability tag will be checked to determine its vulnerability.

The profiling aims to identify un-ACE instructions (e.g. prefetching, dynamically dead instructions and NOPS). Identifying NOPS and prefetching is straightforward. We implemented a post-graduation instruction analysis window to identify dynamically dead instructions. Instructions that can not be proven to be un-ACE will be conservatively classified as ACE. The profiling statically classifies each instruction PC as ACE or un-ACE. A PC is classified as ACE if any of its dynamic instances is identified as an ACE instruction. Instructions executed along mispredicted paths are not used for this

classification. By doing this, we make our classification independent of branch predictor implementation and the non-deterministic inter-thread branch aliasing. The profiling method is conservative since it may predict some eventually squashed instructions as ACE. Moreover, the same instruction is not always ACE or un-ACE during the entire program execution. For example, an instruction within a loop may be un-ACE during the first several iterations, but become ACE in the last iteration if only the last iteration’s computation result is consumed by other instructions, or vice versa. We use the term false-positive to describe the case in which un-ACE instructions are incorrectly identified as ACE, and false-negative to describe the opposite case. As discussed above, using the instruction PC to identify vulnerable instructions can avoid false-negatives, namely, no ACE instruction is mispredicted. However, our method can cause false-positives. Table 1 shows the accuracy of using the instruction PC to identify ACE instructions from committed instructions across 18 SPEC CPU2000 benchmarks. As can be seen, the identification accuracy in most applications is around 98% and the average accuracy is 94%. This indicates that false-positive matches happen infrequently.

Table 1. Accuracy of using PC to identify ACE instructions (Committed instruction only)

Benchmark	Accuracy	Benchmark	Accuracy	Benchmark	Accuracy
<i>applu</i>	99.8%	<i>galgel</i>	98.8%	<i>mgird</i>	99.9%
<i>bzip</i>	87.8%	<i>gap</i>	95.9%	<i>perlbnk</i>	99.9%
<i>craftv</i>	89.4%	<i>gcc</i>	96.5%	<i>swim</i>	99.8%
<i>eon</i>	87.6%	<i>lucas</i>	99.2%	<i>twolf</i>	95.8%
<i>equake</i>	99.1%	<i>mcf</i>	96.1%	<i>vpr</i>	81.8%
<i>facerec</i>	93.7%	<i>mesa</i>	74.9%	<i>wupwise</i>	97.5%
AVG	93.7%				

The PC-based vulnerability classification can incorrectly predict a number of squashed instructions as ACE if their PCs are tagged as ACE. When squashed instructions are considered, the prediction still achieves reasonably high accuracy (as shown in Table 2). Since the branch misprediction rate of each individual benchmark varies across different workload combinations, we present accuracy (average statistics) results of different workload mixes (shown as Table 4). As can be seen, prediction accuracy is higher as workload mixes become more memory bound. This is because the higher cache miss rate on memory-bounded workloads constrains the depth of speculative execution on individual threads and reduces the percentage of squashed instructions.

Table 2. Accuracy of using PC to identify ACE instructions (All executed instructions)

Workloads	CPU	MIX	MEM	AVG
Accuracy	79.3%	80.3%	85.4%	81.7%

3.3. Combine ORBIT with Prediction

The schemes we propose in Section 3.2 can degrade performance since they delay the dispatch of instructions (ACE instructions in *DelayACE*) until their operands are ready. Alternatively, we can dispatch instructions slightly before their operands are ready. By doing so, instructions will become ready-to-execute soon after entering the IQ. Putting more instructions into the IQ one cycle ahead of time can help improve performance by effectively exploiting ILP. In this subsection, we propose a scheme called *PredictALL*, which allows instructions with non-ready operands to be dispatched just before the predicted operand ready cycle.

The efficiency of the *PredictALL* scheme relies on several key design parameters. First, the ready time of instruction source operands should be predicted. Otherwise, instructions can not be dispatched ahead of their ready time. Second, once the ready time is predicted, the scheme should be able to dispatch instructions ahead of time while minimizing their residency cycles in the IQ. In this paper, we opt for dispatching instructions only one cycle before they become ready to execute because this is the required latency to place instructions into the IQ. By doing this, instructions will be immediately available for issuing and execution at the time they enter the IQ. Finally, since predicting an operand’s ready time usually involves forecasting the completion time of the instructions which produce the source operands, it is crucial to decide when such a prediction should be made. The prediction of instruction completion time can be made at several pipeline stages, such as fetch, renaming, dispatch and issue. Predictions are less accurate when made earlier since there are more unpredictable variations (e.g. resource conflicts) between the prediction time and the actual completion time. In this study, we predict an instruction’s destination register ready time when it is entering into the IQ. Note that if an instruction’s predicted operand readiness time is longer than the actual readiness time, ORBIT will dispatch the instruction based on its actual readiness time.

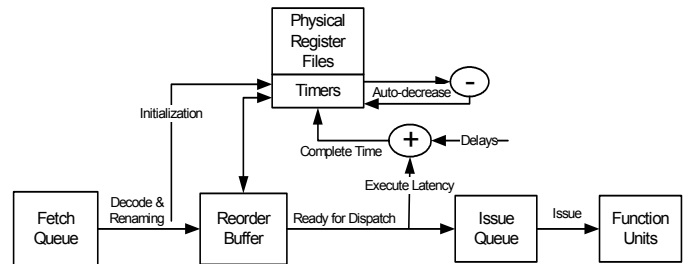


Figure 4. ORBIT with prediction

Figure 4 illustrates the overall architecture of ORBIT with prediction. As can be seen, to implement *PredictALL*, we need a timer for each physical register to count down the remaining cycles during which a source operand will not be available. The register ready bit array in *DelayALL* therefore can be replaced by the multi-ported, multi-banked timer array, and the ROB obtains the predicted operands readiness of the instruction via the parallelized access to it. The timer decreases its value by one in each cycle and it returns ready-to-dispatch when it reaches one since instructions are dispatched one cycle before they become ready to execute. The timer is initialized when the corresponding register is allocated in renaming stage and will not count down until it is set. When an instruction is dispatched into the IQ, its predicted completion time will be recorded into the timer associated with its destination register. The timer is set to the sum of dispatch-to-issue delay, issue-to-execute delay and function unit latency (which can be obtained from an instruction’s opcode).

Since prediction is made at the dispatch stage, function unit conflicts at the issue stage should be considered. To track function unit utilization, we set up a counter for each function unit class. In each cycle, the counter is increased by one if the associated function unit class has at least one function unit available. We then divide the counter of each function unit class by the total number of execution cycles. The results are the percentage of execution time without function unit conflicts. We

found that function unit conflicts occur infrequently in the studied SMT processor and its impact to the instruction completion time prediction is negligible. Therefore, we did not consider function unit conflicts in our prediction. Issue latency is another factor that can affect prediction accuracy. A ready instruction in the IQ can not be issued immediately if the number of instructions scheduled for issue exceeds processor issue bandwidth. In our prediction scheme we set the instruction issue latency to zero cycles by assuming that the processor always has sufficient issue bandwidth.

Due to cache misses, accurate prediction of the memory reference instructions' completion time is challenging. Since store instructions have no data consumers, we do not predict their completion time. Therefore we only need to predict operand readiness for load instructions. The load latency, depending on cache hit/miss at different levels, varies from zero cycles (e.g. a hit in load store queue) to hundreds of cycles (e.g. a L2 cache miss). This information can not be accurately determined until the effective address is calculated in the execution stage and the cache access is performed. In order to obtain the load latency for instructions' pre-schedule in the IQ, [17, 18] proposed several complicated predictors to predict the effective address and cache hit/miss at the instruction fetch stage. To implement those predictors, multi-level last-value tables, prediction engine and complex operations on the predictors are required. Note that in our proposed techniques, predicting a cache miss is not required since forecasting load instruction completion time can be deferred to one cycle ahead of the dispatch of load dependent instructions. As a result, our prediction techniques do not require the last-value table [17] and the cache miss prediction engine [18]. Figure 5 illustrates the load instruction completion time prediction in detail. When a load instruction is dispatched, we temporarily predict that its destination register will be ready after a long latency which is equal to the largest number of cycles the timer can be set to. The predicted completion time is updated after the load instruction's effective address is calculated and the cache access is performed. Note that bus competition is not taken into consideration for prediction simplicity, and our load instruction complete time prediction does not consider scenarios such as TLB misses and page faults. In these cases, the faulty instruction will be terminated and re-executed after the exception is handled.

In this study, we also examine an alternative design called *Predict_non_load*, which skips completion time prediction for all load instructions. As a result, instructions that directly depend on loads can not be dispatched to the IQ ahead of time, but prediction resumes for the following dependent instructions (except loads). Figure 6 illustrates the difference between *Predict_non_load* and *PredictALL* with a code segment example, and the data dependence among instructions is also presented. Note that node 9 performs a store operation and its completion time is not predicted in either of the designs, and other instructions' readiness is predicted in *PredictALL*. In *Predict_non_load*, however, the readiness of instruction 3 and 6 are not forecasted since they are load instructions. Correspondingly, instruction 4 and 7 are not dispatched until they are ready-to-execute due to the direct data dependence on instruction 3 and 6. Additionally, readiness prediction on instruction 5 and 10 are resumed in the *Predict_non_load* scheme since they are indirectly dependent on loads 3 and 6. Since un-ACE instructions contribute fewer ACE-bits than ACE instructions, we further extend the *PredictALL* and *Predict_non_load* to *PredictALL_DelayACE* and

Predict_non_load_DelayACE. Similar to *DelayACE*, both *PredictALL_DelayACE* and *Predict_non_load_DelayACE* don't apply the operand readiness based dispatch to un-ACE instructions.

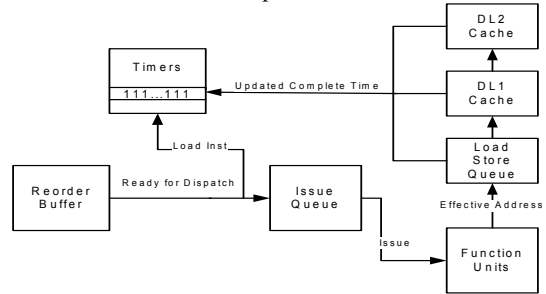


Figure 5. Predict the load instruction completion time

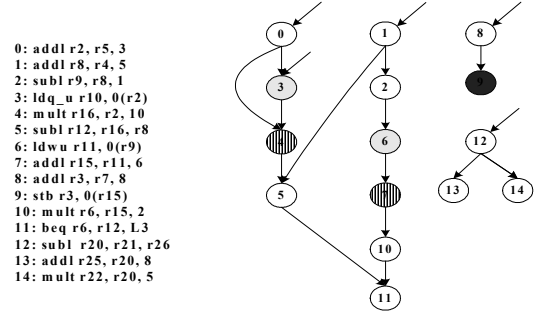


Figure 6. Readiness prediction example on *PredictALL* and *Predict_non_load*

To improve the performance of SMT processors, [15] proposed *2OP_BLOCK*, which blocks instructions with two non-ready operands and their corresponding threads in the dispatch stage until one of the sources becomes ready. The goal of *2OP_BLOCK* is to improve IQ utilization by suspending threads with long latency instructions and allocating more IQ entries to threads exhibiting high ILP. *2OP_BLOCK* can reduce IQ AVF since instructions with long waiting cycles are blocked at the dispatch stage. *2OP_BLOCK* resumes instruction dispatch once one source operand of the instruction is ready. Therefore, the instruction still spends IQ residency cycles waiting for other source operands. Another difference between *2OP_BLOCK* and our techniques is that by using out-of-order dispatch, we block not-ready instructions but don't stall the corresponding thread. We implement *2OP_BLOCK* and examine its efficiency on soft-error vulnerability mitigation. In addition, we explore *2OP_BLOCKACE* which applies *2OP_BLOCK* to only ACE instructions.

4. Experimental Setup

To evaluate the reliability and performance impact of the proposed techniques, we have developed a reliability-aware SMT simulation framework. Our framework is built on a heavily modified and extended M-Sim simulator [19] which models a detailed, execution driven simultaneous multithreading processor. Table 3 shows the baseline machine configuration we use in this study. We use ICOUNT [20] which assigns the highest priority to the thread that has the fewest in-flight instructions as the baseline fetch policy.

The SMT workloads in our experiments are comprised of SPEC CPU 2000 integer and floating point benchmarks. We create a set of SMT workloads with individual thread characteristics ranging from computation intensive to memory access intensive (see Table 4). The CPU and MEM workloads

consist of programs only from the CPU intensive and memory intensive workloads respectively. Half of the programs in a SMT workload with mixed behavior (MIX) are selected from the CPU intensive group and the rest are selected from the MEM intensive group. To ensure that our experimental results are not biased by a specific set of threads; we build 3 groups for each type of SMT workload and report average statistics wherever possible. We use the Simpoint tool [21] to pick the most representative simulation point for each benchmark and each benchmark is fast-forwarded to its representative point before detailed multithreaded simulation takes place. The simulations are terminated once the committed instructions from any thread reach 400 million.

Table 3. Simulated machine configuration

Parameter	Configuration
Processor Width	8-wide fetch/issue/commit
Baseline Fetch	ICOUNT
Issue Queue	96
ITLB	128 entries, 4-way, 200 cycle miss
Branch Predictor	2K entries Gshare, 10-bit global history per thread
BTB	2K entries, 4-way
Return Address	32 entries RAS per thread
L1 Instruction	32K, 2-way, 32 Byte/line, 2 ports, 1 cycle access
ROB Size	96 entries per thread
Load/ Store Queue	48 entries per thread
Integer ALU	8 I-ALU, 4 I-MUL/DIV, 4 Load/Store
FP ALU	8 FP-ALU, 4FP-MUL/DIV/SQRT
DTLB	256 entries, 4-way, 200 cycle miss
L1 Data Cache	64KB, 4-way, 64 Byte/line, 2 ports, 1 cycle access
L2 Cache	unified 2MB, 4-way, 128 Byte/line, 12 cycle access
Memory Access	64 bit wide, 200 cycles access latency

The AVF is used as a baseline metric to estimate how susceptible a microarchitecture structure is to soft-error strikes. In our experiments, although ACE-ness is classified at instruction-level, the AVF computation is performed at bit-level. We use throughput IPC, which qualifies the throughput improvement, and harmonic mean of weighted IPC [22], which qualifies both performance improvement and fairness, to evaluate the performance impact of various techniques.

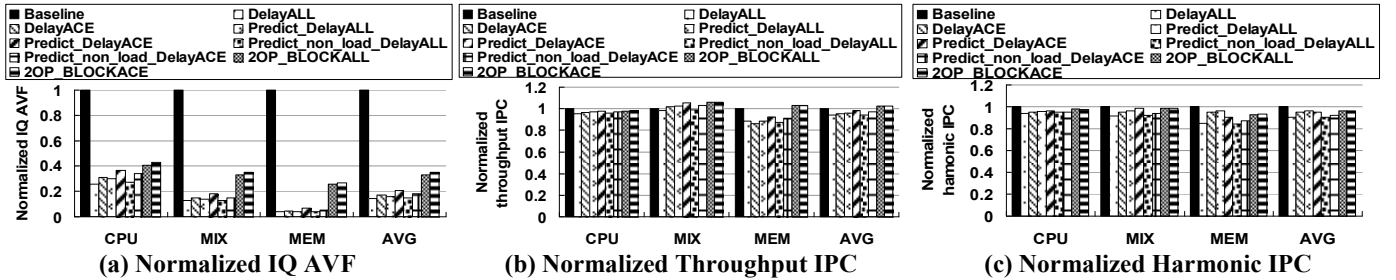


Figure 7. Normalized IQ AVF (a), throughput IPC (b) and harmonic IPC (fetch policy: ICOUNT)

Predict_DelayALL yields higher performance than both *DelayALL* and *DelayACE*. On average, it decreases throughput IPC by 4% and harmonic IPC by 4%. The IQ AVF reduction is 84%, which is smaller than that on *DelayALL* since statistically each instruction could reside for one more cycle in the IQ. Ideally, *Predict_DelayALL* should have no performance penalty since instructions are always permitted to dispatch one cycle before they are ready. However, the maximum number of dispatched instructions can not exceed the processor dispatch bandwidth. In a scenario that multiple predicted ready to execute instructions compete for limited dispatch bandwidth, dispatch congestion occurs and prevents the dispatch of these instructions on time. This will eventually affect the number of instructions that the processor can issue. The performance penalty of *Predict_DelayALL*

Table 4. The studied SMT workloads

Thread Type	Benchmarks	
CPU	Group A	<i>bzip2, eon, gcc, perlbnk</i>
	Group B	<i>gap, facerec, crafty, mesa</i>
	Group C	<i>gcc, perlbnk, facerec, crafty</i>
MIX	Group A	<i>gcc, mcf, vpr, perlbnk</i>
	Group B	<i>mcf, mesa, crafty, equake</i>
	Group C	<i>vpr, facerec, swim, gap</i>
MEM	Group A	<i>mcf, equake, vpr, swim</i>
	Group B	<i>lucas, galgel, mcf, vpr</i>
	Group C	<i>equake, swim, twolf, galgel</i>

5. Evaluation

In this section, we evaluate the efficiency of the proposed techniques on IQ soft-error vulnerability mitigation and assess their performance impact.

5.1. Reliability and Performance Impact

Figure 7 a-c shows IQ AVF, throughput IPC and harmonic IPC yielded on the proposed techniques. The results are normalized to a baseline case without optimization. As Figure 7 (a) shows, on average, the *DelayALL* scheme reduces IQ AVF by about 86%. On MEM workloads where a high frequency of L2 cache misses cause more long latency instructions, the IQ AVF is reduced by 96%. This suggests that the long residency cycles of these instructions are effectively reduced by ORBIT. Figure 7 (b) and (c) show that *DelayALL* decreases throughput and harmonic IPC by 6% and 10% respectively. *DelayACE* achieves an 83% IQ AVF reduction which is slightly smaller than that on *DelayALL* since *DelayACE* does not block the dispatch of un-ACE instructions and un-ACE instruction residency cycles also contribute to the IQ AVF as un-ACE instructions still contain a small number of ACE bits. On average, *DelayACE* results in better performance by showing a 5% reduction in both throughput and harmonic IPC.

is more noticeable on MEM workloads due to burst increased pressure on the dispatch bandwidth caused by L2 misses. Compared with *Predict_DelayALL*, on average across all of the workloads, *Predict_DelayACE* further improves performance by showing 1% throughput IPC and 3% harmonic IPC reduction. The IQ AVF reduction on *Predict_DelayACE* is 79%. Figure 7 (b) and (c) show compared to *Predict_DelayALL*, *Predict_non_load_DelayALL* yields lower throughput and harmonic IPC. In *Predict_non_load_DelayALL*, the dispatch of direct consumers of load instructions are delayed until their operands are available as the completion time of load instructions is not predicted. As a result, *Predict_non_load_DelayALL* also shows lower IQ AVF. A similar observation holds when comparing *Predict_non_load_DelayACE* with *Predict_DelayACE*.

Figure 7 also shows the IQ AVF and performance yielded on *2OP_BLOCKALL* and *2OP_BLOCKACE*. The *2OP_BLOCK* techniques increase performance by 2% and reduce IQ AVF by 65%. Note that [15] reports a higher performance improvement (9% and 5% increase in throughput IPC and harmonic IPC) on *2OP_BLOCK* on an IQ with a size of 64. The modeled SMT processor in our study has as a 96-entry IQ. As discussed in [15], the benefit of *2OP_BLOCK* reduces with increased IQ size. On CPU and MIX workloads, *Predict_DelayACE* and *Predict_non_load_delayACE* show similar performance characteristics as *2OP_BLOCK* while *Predict_DelayACE* and *Predict_non_load_delayACE* exhibit superior capability in mitigating IQ AVF (e.g. on MIX workloads, IQ AVF is further decreased by 18% on *PredictACE* and *Predict_non_load_delayACE*).

5.2. A Comparison with Different Fetch Policies

To improve performance in modern SMT processors, various fetch policies have been proposed in the past. For example, STALL [23] blocks instruction fetch from offending threads when experiencing a L2 cache miss. As an extension of STALL, FLUSH [23] not only stalls those threads but also squashes instructions from them. DG [24] and PDG [24] respond to cache misses by assigning a lower priority to offending threads. These fetch policies are built on the ICOUNT scheme. Prior work [11] analyzed the impact of SMT fetch policies on microarchitecture soft-error vulnerability and showed that compared with ICOUNT, the advanced fetch policies exhibit a superior capability of soft-error mitigation. In this subsection, we compare the reliability and performance characteristics of the proposed techniques

implemented with ICOUNT with those on advanced fetch policies.

Figure 8 shows the results on IQ AVF (a), throughput IPC (b) and harmonic IPC (c). Due to space limitations, we show results using the worst case (*DelayALL*), two best cases (*Predict_DelayACE* and *Predict_no_load_DelayACE*), and the average statistics across all 8 schemes. As Figure 8 (a) depicts, the advanced fetch policies improve IQ reliability to some extent, especially FLUSH which reduces IQ AVF significantly on MEM workloads due to the frequently triggered flush operation. Compared to advanced policies, ORBIT schemes achieve a greater IQ AVF reduction. Note that all of these fetch policies aim to improve IQ throughput by disallowing instructions to occupy an IQ entry for too long. However, instructions are permitted to dispatch with not ready operands and cycles spent waiting on source operands are unavoidable. Throughput IPC comparison (Figure 8-b) shows that FLUSH and STALL can boost the overall performance of SMT execution while our designs (e.g. *Predict_DelayACE* and *Predict_no_load_DelayACE*) yield negligible performance degradation. Nevertheless, FLUSH and STALL suffer the fairness problem as shown in Figure 8 (c) since they blindly enforce flush/stall operations on all of the instructions from the offending threads and are biased to high ILP threads. Contrary to this, ORBIT techniques only yield a negligible loss in harmonic IPC. Although DG and PDG yield higher performance than ORBIT schemes, the much higher IQ vulnerability reduction gained from ORBIT schemes outweigh the slight performance difference.

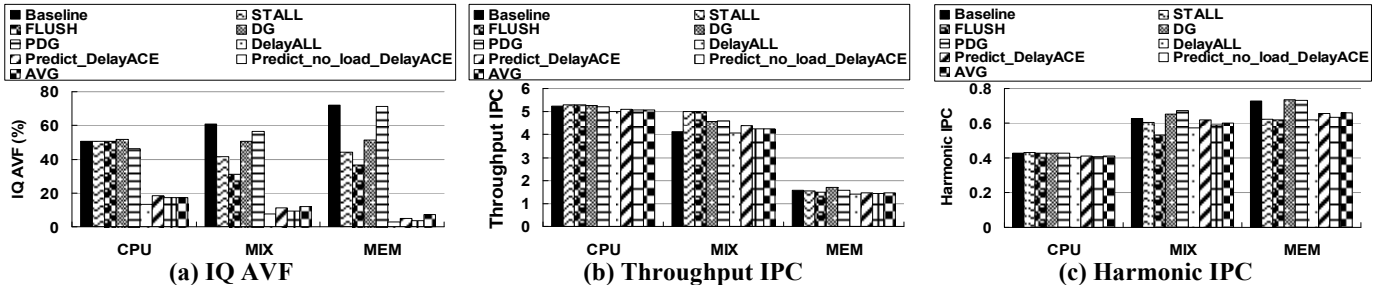


Figure 8. A Comparison with different fetch policies

In our study, we use an IQ with a size of 96-entries. To investigate the applicability of the ORBIT schemes on different IQ configurations, we consider IQ sizes ranging from 32-entries to 128-entries. Experimental results show that the ORBIT techniques yield significant IQ AVF reduction on IQ with different sizes.

5.4. Reliability Impact on the Entire Processor Core

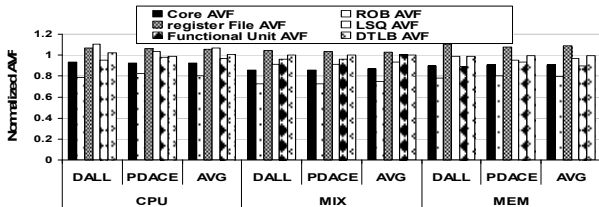


Figure 9. The Impact of proposed techniques on core and other microarchitecture structures' vulnerability (DALL: *DelayALL*, PDACE: *Predict_DelayACE*)

As described above, the proposed techniques exhibit strong ability in improving IQ reliability. It is important to evaluate their impact on the reliability of entire processor core and other

microarchitecture structures. Figure 9 shows the AVFs of processor core and major microarchitecture structures across the three types of workloads. Results are normalized to the baseline case without any optimization. We present results of the worst case (*DelayALL*) and the best case (*Predict_DelayACE*), and the averaged case through all the proposed techniques.

As Figure 9 shows, on average, our techniques reduce core AVF by 10%. The impact of the techniques on other microarchitecture structures is trivial except ROB (AVF reduces 22%). Because our techniques delay instructions' dispatch in each thread, the default fetch policy (ICOUNT) will fetch fewer instructions when the thread has a number of delayed instructions in the pipeline. As a result, it prevents larger number of vulnerable bits that will exhibit long residency cycles from entering into the thread's private ROB. On the other hand, ICOUNT may bring vulnerable bits for other threads with fewer instructions in the pipeline into ROB, however, their residency time in ROB tends to be short. As a result, the overall ROB AVF decreases.

6. Related Work

There is a growing amount of work aimed at characterizing soft error behavior at the microarchitecture level [6, 7, 8, 10, 12, 16]. Sridharan et al. [36] examined the vulnerability contribution of instructions that are in-flight during long-stall instructions. To reduce IQ AVF, Weaver et al. [1] proposed selectively squashing instructions when long delays are encountered. They examined cache miss triggers and squashing actions to remove existing instructions from the IQ. Joseph et al. [15] proposed *ZOP_BLOCK* to block instructions with 2 non-ready operands and the corresponding thread at dispatch stage to improve the performance, but its impact on reliability is unknown. Jones et al. [34] proposed a software assisted approach to reduce the IQ power consumption by limiting the number of instructions dispatched and resident in the IQ. Karkhanis [35] gained IQ energy reduction by avoiding the fetch of mis-speculated instructions and reduce the number of instructions in IQ. Our work is unique in its joint consideration of performance and reliability of IQ design on SMT architectures. In [25, 26, 27, 28, 29], researchers proposed various dependence-based IQ designs for instructions being issued on data-flow instead of program order to reduce the complexity of issue logic and access latency to large IQ. The instructions' ready order in the IQ is predicted. Nevertheless, the predictions do not foretell the exact instruction ready time as we did in this study. Moreover, to implement those designs, either a total new IQ structure (e.g. IQ is split into several FIFO buffers) and issue logic or a complicated prescheduled issue buffer is required, whereas our designs are much simpler. In [30, 31, 32], instructions' completion time is predicted for instructions' pre-schedule in instruction window and therefore, improve the performance. To achieve the target, prediction has to be done at an early pipeline stage (e.g. decode and renaming stage), and as a result, these mechanisms rely on implementing complicated prediction tables to maintain the required prediction accuracy. While in our techniques, a much simpler predictor was implemented since we only need to perform prediction during the dispatch stage.

7. Conclusions

Based on the observation that instructions' long waiting-for-ready time significantly increases IQ soft-error vulnerability, we have developed six novel microarchitecture techniques to improve the issue queue reliability while maintaining processor performance. We use instruction operand readiness prediction to reduce the performance penalty of ORBIT schemes. The proposed *Predict_DelayACE* scheme reduces IQ vulnerability by 79% with 1% throughput IPC and 3% harmonic IPC degradation on average across all three types of workloads. Moreover, it also reduces processor core vulnerability by 10% on average. The ORBIT schemes outperform the advanced SMT fetch policies when reliability, throughput and fairness are all considered.

Acknowledgements

This research is partially supported by NSF grant 0720476, SRC grant 2007-RJ-1651G, and Microsoft Research Trustworthy Computing Award 14707. José Fortes is also funded by the BellSouth Foundation.

References

[1] C. Weaver et al., Techniques to Reduce the Soft Error Rate of a High Performance Microprocessor, ISCA, 2004.

- [2] M. Choudhury et al., Design Optimization for Single-Event Upset Robustness using Simultaneous dual-VDD and Sizing Techniques, ICCAD, 2006
- [3] Y. Yeh, Triple-triple Redundant 777 Primary Flight Computer, In Proceedings of the IEEE Aerospace Applications Conference, 1996
- [4] S. K. Reinhardt et al., Transient Fault Detection via Simultaneous Multithreading, ISCA, 2000
- [5] H. Zhou, A Case for Fault Tolerance and Performance Enhancement using Chip Multi-Processors, IEEE Computer Architecture Letters, Sept. 2005
- [6] S. S. Mukherjee et al., A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor, MICRO, 2003.
- [7] A. Biswas et al., Computing Architectural Vulnerability Factors for Address-Based Structures, ISCA, 2005.
- [8] H. Asadi et al., Balancing Performance and Reliability in the Memory Hierarchy, ISPASS, 2005.
- [9] X. Li et al., SoftArch: An Architecture-Level Tool for Modeling and Analyzing Soft Errors, DSN, 2005.
- [10] X. Fu et al., Characterizing Microarchitecture Soft Error Vulnerability Phase Behavior, MASCOTS, 2006.
- [11] W. Zhang, An Analysis of Microarchitecture Vulnerability to Soft Errors on Simultaneous Multithreaded Architectures, ISPASS, 2007.
- [12] N. J. Wang et al., Characterizing the Effects of Transient Faults on a High Performance Processor Pipeline, DSN, 2004.
- [13] J. Borkenhagen et al., A Multithreaded PowerPC Processor for Commercial Servers, IBM Journal of Research and Development, 2000.
- [14] D. Marr et al., Hyper-Threading Technology Architecture and Microarchitecture, Intel Technology Journal, 2002.
- [15] J. Sharkey et al., Efficient Instruction Schedulers for SMT Processors, HPCA, 2006.
- [16] S. Kim et al., Soft Error Sensitivity Characterization of Microprocessor Dependability Enhancement Strategy, DSN, 2002.
- [17] K. Wang et al., Highly Accurate Data Value Prediction using Hybrid Predictors, MICRO, 1997.
- [18] G. Memik et al., Just Say No: Benefits of Early Cache Miss Determination, HPCA, 2003.
- [19] J. Sharkey, M-Sim: A Flexible, Multi-threaded Simulation Environment, Tech. Report CS-TR-05-DPI, SUNY Binghamton, 2005.
- [20] D. Tullsen et al., Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor, ISCA, 1996.
- [21] T. Sherwood et al., Automatically Characterizing Large Scale Program Behavior, ASPLOS, 2002.
- [22] K. Lou et al., Balancing Throughput and Fairness in SMT processors, ISPASS, 2001.
- [23] D. Tullsen et al., Handling Long-latency Loads in a Simultaneous Multithreading Processor, MICRO, 2001.
- [24] A. El-Moursy et al., Front-end Policies for Improved Issue Efficiency in SMT Processors, HPCA, 2003.
- [25] R. Canal et al., A Low-complexity Issue Logic, ICS, 2000.
- [26] R. Canal et al., Reducing the Complexity of the Issue Logic, ICS, 2001.
- [27] S. Palacharla et al., Complexity-Effective Superscalar Processors, ISCA, 1997.
- [28] P. Michaud et al., Data-flow Prescheduling for Large Instruction Windows in Out-of-order Processors, HPCA, 2001.
- [29] S.E. Raasch et al., A Scalable Instruction Queue Design Using Dependence Chains, ISCA, 2002.
- [30] D. Ernst et al., Cyclone: A Broadcast-Free Dynamic Instruction Scheduler with Selective Replay, ISCA, 2003.
- [31] Y. Liu et al., Scaling the Issue Window with Look-Ahead Latency Prediction, ICS, 2004.
- [32] T. E. Ehrhart et al., Reducing the Scheduling Critical Cycle using Wakeup Prediction, HPCA, 2004.
- [33] A. Parashar et al., Slick:Slice-based locality exploitation for efficient redundant multithreading, ASPLOS, 2006.
- [34] T.M. Jones et al., Software Directed Issue Queue Power Reduction, HPCA, 2005
- [35] T. Karkhanis et al., Saving Energy with Just In Time Instruction Delivery, ISLPED, 2002.
- [36] V. Sridharan et al., Reliability in the Shadow of Long-Stall Instructions, SELSE-3 Workshop, 2007
- [37] V. Stojanovic et al., A Cost-Effective Implementation of an ECC-protected Instruction Queue for Out-of-Order Microprocessors, DAC, 2006