Optimizing Issue Queue Reliability to Soft Errors on Simultaneous Multithreaded Architectures

Xin Fu, Wangyuan Zhang, Tao Li and José Fortes
Department of ECE, University of Florida
xinfu@ufl.edu, zhangwy@ufl.edu, taoli@ece.ufl.edu, fortes@acis.ufl.edu

Abstract

The issue queue (IQ) is a key microarchitecture structure for exploiting instruction-level and thread-level parallelism in dynamically scheduled simultaneous multithreaded (SMT) processors. However, exploiting more parallelism yields high susceptibility to transient faults on a conventional IQ. With the rapidly increasing soft error rates, the IQ is likely to be a reliability hot-spot on SMT processors fabricated with advanced technology nodes using smaller and denser transistors with lower threshold voltages and tighter noise margins. In this paper, we explore microarchitecture techniques to optimize IQ reliability to soft error on SMT architectures. We propose to use off-line instruction vulnerability profiling to identify reliability critical instructions. The gathered information is then used to guide reliability-aware instruction scheduling and resource allocation in multithreaded execution environments. We evaluate the efficiency of the proposed schemes across various SMT workload mixes. Extensive simulation results show that, on average, our microarchitecture level soft error mitigation techniques can significantly reduce IQ vulnerability by 42% with 1% performance improvement. To maintain runtime IQ reliability for pre-defined thresholds, we propose dynamic vulnerability management (DVM) mechanisms. Experimental results show that our DVM techniques can effectively achieve desired reliability/performance tradeoffs.

1. Introduction

Soft errors have become an increasing cause of failures on microprocessors fabricated with advanced technology nodes using smaller and denser transistors with lower threshold voltages and tighter noise margins. 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. The trends of scaling-down semiconductor technology to the nano-scale device and ever-increasing complexity of microprocessors suggest that soft error rates of future generations of processors are projected to increase significantly [1].

Due to the diminishing performance gains from instruction level parallelism (ILP) on wide-issue superscalar processors, simultaneous multithreaded (SMT) architectures [2] have been proposed and used in commercial processors [3] to exploit thread-level parallelism (TLP). The SMT architectures improve the performance of a superscalar processor by dynamically sharing pipeline and microarchitecture resources among multiple, concurrently running threads. In a dynamically scheduled SMT processor, the issue queue (IQ) is a key microarchitecture structure for extracting parallelism. The IQ holds decoded instructions from multiple threads until their operands and appropriate function units are available. In SMT processors that aggressively exploit ILP and TLP, the IQ buffers and exposes a large number of instructions to neutron or alpha particle strikes.

Using a reliability-aware architecture simulator, we characterize soft error vulnerability of several key microarchitecture structures in a SMT processor. Results are shown in the form of the architecture vulnerability factor (AVF) [4] of a hardware structure which estimates the probability that a transient fault in that hardware structure will produce incorrect program results. Details of our simulation framework, machine configuration, evaluated workloads and metrics are presented in Section 3. Figure 1 indicates that among the microarchitecture structures studied, the IQ exhibits the highest vulnerability. This indicates that the IQ is likely to be a reliability hot-spot in SMT processors fabricated using advanced technology nodes. In this paper, we explore reliability-aware microarchitecture optimizations to mitigate IQ soft error vulnerability on SMT architectures. We focus our study on the IQ which shows the highest vulnerability, and it is a necessary first step towards protecting entire chip with multiple components to which similar principles might also be applicable.

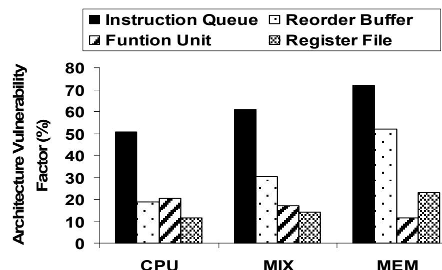


Figure 1. Microarchitecture soft-error vulnerability profile on SMT processor

The main contributions of this paper are as following:

- We propose to use off-line instruction vulnerability profiling information to identify reliability critical instructions, and prioritize their scheduling. The vulnerability-aware instruction scheduling shows the potential to reduce the residency cycles of vulnerable bits in the IQ.
- We apply dynamic resource allocation to the IQ to prevent excessive vulnerable bits from entering the IQ. Combining with vulnerability-aware instruction scheduling, the proposed schemes achieve significant IQ reliability enhancement.
- Similar to its performance and power domain profiles, a program's reliability domain characteristics exhibit time varying behavior: the runtime IQ vulnerability varies significantly during program execution. We propose dynamic vulnerability management (DVM) mechanisms to

maintain runtime IQ reliability for the pre-defined thresholds. We show that a conventional IQ enhanced with DVM can ensure that its runtime vulnerability meets the pre-set reliability targets.

The remainder of this paper is organized as follows. Section 2 presents our reliability-aware microarchitecture optimizations. Section 3 describes our experimental setup. Section 4 evaluates the reliability efficiency and performance overhead of the proposed schemes. Section 5 proposes dynamic vulnerability management mechanisms. Section 6 discusses related work and section 7 concludes the paper.

2. IQ Reliability Optimizations for SMT Processors

In this section, we propose microarchitecture techniques to reduce the SMT processor's IQ vulnerability to soft errors. We first reveal opportunities for applying vulnerable-instruction-aware issue and study its implications on IQ vulnerability mitigation. We then propose several optimizations that can efficiently optimize reliability, performance and their tradeoffs.

2.1. Vulnerable InStruction Aware (VISA) Issue

The processor state bits required for architecturally correct execution (ACE) are called ACE bits [4] and those bits that do not affect the correctness of program execution are called un-ACE bits. 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. ACE instructions are those whose computation results affect the program final output, while un-ACE instructions do not affect the final results. In [4], Mukherjee et al. reported 55% of un-ACE instructions in SPEC2000 programs compiled with IA64 ISA. Note that un-ACE instructions also contain ACE-bits (e.g. opcode). In contrast to ACE instructions, un-ACE instructions contain fewer vulnerable bits. The soft error vulnerability of a hardware structure is determined by the percentage of ACE bits that the structure holds and the residency cycles of ACE bits in that structure. Since ACE instructions are the major contributor of ACE bits, the IQ soft error vulnerability can be mitigated by reducing the residency cycles and quantity of ACE instructions in the IQ.

To reduce ACE instruction residency cycles in the IQ, we propose Vulnerable InStruction Aware (VISA) issue that gives the ACE instructions higher priority than the un-ACE instructions. Therefore, once there is a ready ACE instruction, it can bypass all the ready-to-execute un-ACE instructions. If there are several ready ACE instructions, they will be issued in the program order. Note that the ready un-ACE instructions cannot be issued until all the ready-to-execute ACE instructions have been issued. If the number of ready ACE instructions is less than the number of available issue slots, the ready un-ACE instructions can also be issued in their program order.

Workload IQ utilization characteristics (e.g. the average number of ready instructions per cycle and the fraction of ACE instructions in ready instructions) can significantly affect the efficiency of applying VISA issue for IQ vulnerability reduction. For example, if the average number of ready instructions per cycle is smaller than the issue bandwidth, it is unnecessary to use the VISA issue policy since all ready ACE and un-ACE instructions can be issued within the same cycle. If the average fraction of ACE instructions per cycle is small (or there is no ACE instruction in the ready state at all), the benefit of applying VISA issue policy will be marginal. To answer the above questions, we set up experiments to characterize a SMT processor's IQ utilization from both performance and reliability perspectives. In each cycle, we count the number of instructions stayed in the ready queue (a collection of IQ entries that holds ready-to-execute instructions) and break them down into ACE and un-ACE instructions. Note that the ready queue length at a certain cycle is an accumulative effect of former cycles, it doesn't represent the number of newly generated ready instructions at that cycle. We then plot the histograms of ready queue size with the corresponding ACE instruction percentage. Figure 2 shows results for the 4-context CPU workload. The Y-axis on the left represents the probabilistic-based ready queue length distribution and the Y-axis on the right represents the corresponding ACE instruction percentage.

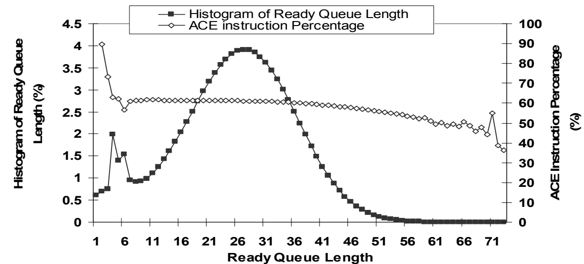


Figure 2. The histograms of ready queue length and ACE instruction percentage on a 96-entry IQ. SMT processor issue width = 8, the experimented workloads: 4-context CPU (*bzip,eon,gcc,perlbmk*)

As can be seen, the maximal ready queue length is 73. This indicates that on SMT processors, exploiting TLP increases the number of ready-to-execute instructions in the IQ. Interestingly, a hill is shown in the ready queue length distribution and its peak value is around 26. Moreover, only 10% of execution cycles have a ready queue length less than 9. Since the issue width of the SMT processor is 8, there are abundant ready-to-execute instructions that can be chosen by the issue logic. The ACE instruction percentage plot shows that on average, 60% of the ready instructions are ACE instructions. The fraction of ACE instructions becomes higher when the ready queue is short. The above observations indicate that there are good opportunities to reduce IQ vulnerability by giving ACE instructions a higher issue priority.

The VISA-based issue policy heavily relies on the capability of identifying the ACE-ness of each instruction in the IQ. However, in general, a retired instruction cannot be classified as ACE or un-ACE until a large amount of its following instructions have graduated. In [4], a post-graduate instruction analysis window with a size of 40,000 instructions is used for vulnerability classification. To perform the just-in-time vulnerability identification at per-instruction level, we propose to perform instruction vulnerability characterization offline and extend ISA (Alpha ISA is applied in our work) to encode the 1-bit ACE-ness tag (e.g. ACE or un-ACE). When an instruction is decoded, its ACE-ness tag will be checked to determine its vulnerability. The offline profiling statically classifies each instruction's program counter (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 the 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 can predict some finally 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 in the first several iterations, but becomes ACE at the last iteration if only the last iteration's computation result is consumed by other instructions, and vise versa. We refer false-positive to the case that un-ACE instructions are incorrectly identified as ACE, whereas false-negative is the opposite case. As described above, using instruction PC to identify vulnerable instructions can avoid false-negative, namely, no ACE instruction is mispredicted. However, our method can cause false-positive. Table 1 shows the accuracy of using instruction PC to identify ACE instructions from committed instructions across different SPEC CPU2000 benchmarks. As can be seen, the identification accuracy in most applications is around 98% and the average accuracy is 93%. This indicates the false-positive matches happen infrequently. Our PC-based ACE-ness classification can incorrectly predict a number of squashed instructions as ACE if their PCs are tagged as ACE, however, the prediction still achieves good accuracy (83% on average) when squashed instructions are considered.

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

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

2.2. Exploring VISA Issue Based Optimizations

The vulnerability-aware instruction scheduling shows the potential to reduce the residency cycles of vulnerable bits in the IQ. In this section, we further apply dynamic resource allocation to the IQ to prevent excessive vulnerable bits from entering the IQ. Combining with vulnerability-aware instruction scheduling, the proposed schemes achieve significant IQ reliability enhancement by effectively reducing the number of vulnerable bits and their residency in the IQ.

(1) Optimization 1: Dynamic IQ Resource Allocation

Using VISA-based issue reduces ACE instructions residency cycles but it also increases the overall IQ utilization. Because ACE instructions generally exhibit longer data dependence chains than unACE instructions, more ILP can be exploited by issuing ACE instructions earlier. As a result, more instructions can be dispatched into IQ. From the reliability perspective, as the number of instructions in IQ increases which results in more ACE bits moving to the IQ, the IQ becomes more vulnerable to soft error strikes. If IQ resource allocation can be dynamically controlled to maintain performance while minimizing soft error vulnerability, a better performance/reliability tradeoff can be achieved. In this section, we explore reliability-aware IQ resource allocation to control the quantity of ACE bits that the processor can bring into the IQ. Incorporated with VISA issue policy, the proposed techniques can effectively mitigate IQ vulnerability to soft error.

When an instruction is in the dispatch stage, the processor checks whether there is a free entry in the IQ. If the IQ is fully occupied, the instruction has to wait for an available entry. To reduce IQ vulnerability, we setup a threshold to cap IQ utilization. The instruction dispatch logic compares current IQ utilization with this threshold. If the current IQ utilization is higher than the threshold, the processor will not allocate new IQ entries for instructions even there are idle entries in the IQ. As a result, the processor can not dispatch any instruction to the IQ until the IQ utilization drop below the threshold. To identify the optimal threshold, a large number of off-line simulations need to be performed for each SMT workload. In this paper, we propose an on-line mechanism that learns and adapts the IQ utilization threshold dynamically. We sample workload execution characteristics within fixed-size intervals and then use the gathered statistics to setup IQ resource utilization appropriately for the next execution interval. Figure 3 shows the dynamic IQ resource allocation algorithm. IQL stands for the number of allocated IQ entries.

$$0 \le IPC \le 2,\ IQL = min\left\{\left(RQL + \frac{1}{6} * IQ_SIZE\right), \frac{1}{3} * IQ_SIZE\right\}$$

$$2 < IPC \le 4,\ IQL = min\left\{\left(RQL + \frac{1}{3} * IQ_SIZE\right), \frac{1}{2} * IQ_SIZE\right\}$$

$$4 < IPC \le 6,\ IQL = min\left\{\left(RQL + \frac{1}{2} * IQ_SIZE\right), \frac{2}{3} * IQ_SIZE\right\}$$

$$6 < IPC \le 8,\ IQL = min\left\{\left(RQL + \frac{2}{3} * IQ_SIZE\right), IQ_SIZE\right\}$$

Figure 3. Dynamic IQ resource allocation based on IPC, ready queue length and total IQ size

In Figure 3, RQL is the ready queue length and IQ_SIZE is the total size of the IQ. As can be seen, we adapt IQ resource allocation strategies using workload IPC. This is because IQ utilization can highly correlate with IPC: high-IPC workloads require more IQ entries to exploit ILP. Since processor performance is sensitive to the size of ready queue, we incorporate it in our decision making policies. To give vulnerability reduction a priority, we use static caps which are proportional to the total size of the IQ. Since the maximal commit bandwidth of the studied SMT processor is 8, we partition the IPC into 4 non-overlapping regions (our experimental results show that 4 regions outperform other number of regions) and set up the ratios using the low and high IPCs of each region. Alternatively, these ratios can be dynamically setup using the actual IPC. We experiment with dynamic ratio setup using linear models that correlates with IPC. Simulation results show that both static and dynamic ratios show similar efficiency. We use static ratios in this paper due to their simplicity. The interval size is another important parameter in our design. If it is too large, the technique will not be adaptive enough to the resource requirements. If it is too small, the technique will be too sensitive to the changes of workload behavior. After experimenting with various interval sizes, we choose an interval size of 10K cycles in our design.

(2) Optimization 2: Handling L2 Cache Misses

As the number of L2 cache miss increases, the ready queue becomes shorter and there will be more instructions sitting in the waiting queue (a collection of IQ entries that holds not-ready-to-execute instructions) until the cache misses are solved. Due to the

clogged IQ, processors exhibit low IPC performance. After the cache misses are solved, the ready queue size and IPC increases rapidly. However, as shown in Figure 3 when the IPC and the ready queue length are both low, the number of allocated IQ entries will be small. This will limit the number of instructions that can be dispatched to the waiting queue. After L2 misses are solved, the number of instructions that become ready-to-execute will be much less than the scenario where no IQ resource control is performed. Therefore, the optimization shown in Figure 3 will result in noticeable performance degradation if there are frequent L2 cache misses.

$$L2_cache_miss \le T_{cache_miss} \quad \begin{cases} 0 \le IPC \le 2, \; IQL = min\left[\left(RQL + \frac{1}{6} * IQ_SIZE\right), \frac{1}{3} * IQ_SIZE\right] \\ 2 < IPC \le 4, \; IQL = min\left[\left(RQL + \frac{1}{3} * IQ_SIZE\right), \frac{1}{2} * IQ_SIZE\right] \\ 4 < IPC \le 6, \; IQL = min\left[\left(RQL + \frac{1}{2} * IQ_SIZE\right), \frac{2}{3} * IQ_SIZE\right] \\ 6 < IPC \le 8, \; IQL = min\left[\left(RQL + \frac{2}{3} * IQ_SIZE\right), IQ_SIZE\right] \end{cases}$$

$$L2_cache_miss > T_{cache_miss}, \quad enable_FLUSH_policy$$

Figure 4. L2-cache-miss sensitive IQ resource allocation

Recall that our goal is to mitigate IQ soft error vulnerability with negligible performance penalty. To achieve this, we propose a L2-cache-miss sensitive IQ resource allocation strategy. As Figure 4 shows, when the L2 cache miss frequency is below a threshold (T_{cache_miss}), the dynamic IQ resource allocation mechanism described in Figure 3 is used to perform reliability optimization. On the other hand, when the L2 cache miss frequency exceeds the threshold, FLUSH fetch policy [8] is used for vulnerability mitigation. On SMT processors, FLUSH stalls threads that cause L2 cache misses by flushing their instructions from the pipelines. The de-allocated pipeline resource can be efficiently used by the non-offending threads to boost their performance. Note that the cache miss threshold T_{cache_miss} is an important parameter in our design. Using different SMT workload mixes, we performed a sensitivity analysis and choose 16 as the L2 cache miss threshold.

3. Experimental Setup

To evaluate the reliability and performance impact of the proposed techniques, we use a reliability-aware SMT simulation framework developed in [5]. It is built on a heavily modified and extended M-Sim simulator [6] which models a detailed, execution driven simultaneous multithreading processor. We use the computed AVF as a metric to quantify hardware soft error susceptibility. In our experiments, although ACE-ness is classified at instruction-level, the AVF computation is performed at bit-level. Table 2 lists microarchitecture parameters of the SMT processor we considered in this study. The ICOUNT [7], which assign the highest priority to the thread that has the fewest in-flight instructions is used as the default fetch policy. In this work, we further examine the efficiency of the proposed IQ reliability optimizations using a set of advanced fetch policies such as STALL [8], DG [9], PDG [9] and FLUSH [8].

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 3). The CPU and MEM workloads consist of programs all from the computation 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. The results we presented in this paper are average statistics of each benchmark category. We use the Simpoint tool [10] 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 total number of simulated instructions reaches 400 million.

Table 2. 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

Table 3. The Studied SMT Workloads

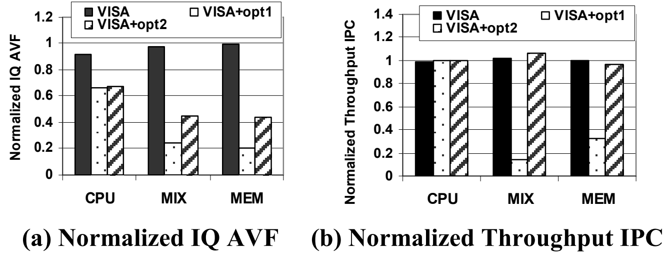
Thread Type		Benchmarks
CPU	Group A	*bzip2, eon, gcc, perlbmk*
	Group B	*gap, facerec, crafty, mesa*
	Group C	*gcc, perlbmk, facerec, crafty*
MIX	Group A	*gcc, mcf, vpr, perlbmk*
	Group B	*mcf, mesa, crafty, equake*
	Group C	*vpr, facerec, swim, gap*
MEM	Group A	*mcf, equake, vpr, swim*
	Group B	*lucas, galgel, mcf, vpr*
	Group C	*equake, swim, twolf, galgel*

4. Evaluation

In this section, we evaluate the efficiency of VISA-based issue and optimizations across various SMT workload mixes. Figure 5 presents the average IQ AVF and the throughput IPC yielded on each type of SMT workloads (CPU, MIX and MEM). ICOUNT is used as the default fetch policy. The IQ AVFs and throughput IPCs are normalized to the baseline case without any optimization.

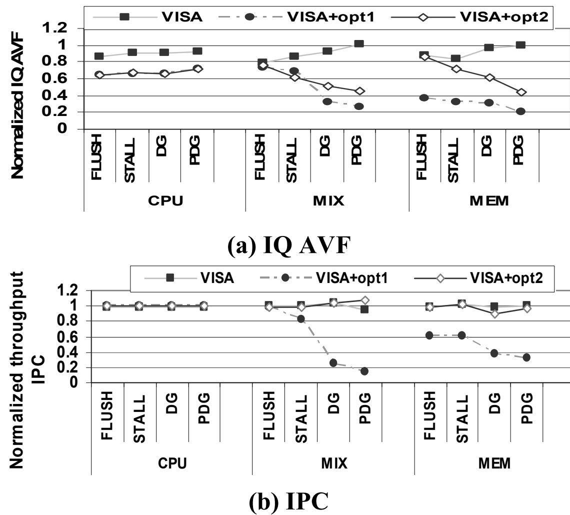
As we expected, by issuing ACE instructions first, the IQ AVF is reduced moderately (5% on average), and throughput IPC is close to the baseline case (1% improvement on average). This is due to the higher IQ utilization. The IQ AVF is further reduced by applying dynamic IQ resource allocation (VISA+opt1). Using CPU workloads as an example, IQ AVF is reduced by about 34% while maintaining the same IPC. This suggests that VISA+opt1 can be used to effectively control the IQ utilization and more aggressively reduce vulnerability without performance penalty on computation intensive workloads. Conversely, on MIX and MEM workloads, VISA+opt1 reduces both throughput IPC and IQ AVF noticeably. This indicates that this scheme is not well suited to handling memory intensive workloads which introduce resource contention more frequently. The results become more promising, however, if we use the number of L2 cache misses to trigger

FLUSH. When we apply VISA+opt2, the throughput IPC is improved 1% than the baseline case on the average, with a 48% reduction in IQ AVF. The IQ AVF reduction on MIX and MEM workloads (56%) is higher than that on CPU workloads (33%) because the baseline IQ AVF is lower on CPU workloads which encounter fewer resource clogs that extend ACE instruction residency. On MEM workloads, VISA+opt2 yields slightly lower IPC than the baseline case. This is caused by the FLUSH fetch policy. FLUSH continues to fetch for at least one thread even if all other threads are stalled and their corresponding pipeline resources have been de-allocated. Thus, the active thread's instructions will occupy the entire pipeline. On MEM workloads, the performance of the active threads can not be improved much by increasing the pipeline resources due to their inherently low ILPs. Worse, the IQ can be occupied by active threads with even lower IPCs than the stalled threads. Interestingly, the normalized IPC yielded by VISA+opt2 on MIX workloads is higher than that on the baseline case. Due to the mix of computation intensive and memory intensive programs, FLUSH is triggered less frequently. Further, when FLUSH is triggered, the probability that the active threads will be low IPC is less than that on the MEM workloads. If the active threads are computation intensive, the throughput IPC will increase. Therefore, the normalized IPC on MIX workloads shows less predictable behavior.



(a) Normalized IQ AVF (b) Normalized Throughput IPC

Figure 5. Normalized IQ AVF (a) and throughput IPC (b) (fetch policy: ICOUNT)



(a) IQ AVF

(b) IPC

Figure 6. Normalized IQ AVF (a) and IPC (b) using different fetch policies

VISA-based issue and optimizations can be integrated into any SMT fetch policy. Next we show the results achieved by using FLUSH, STALL, DG and PDG as the default fetch policy. Figures 6 (a) and (b) show the average IQ AVF and IPC in each workload category when the above fetch policies are used. The results are normalized to the baseline cases of the corresponding fetch policies. As is shown in the figures, even when advanced fetch policies are used, our approaches are still able to provide an impressive IQ AVF reduction of 36% with only a 1% performance penalty on the average. On MIX and MEM workloads, the IQ AVF reduction is less significant using the FLUSH policy than when using the other fetch policies. This is because the FLUSH baseline case is already proficient at handling resource congestion and its IQ AVF is already much lower than the baseline cases of the other fetch policies. On CPU workloads, however, the differences in the advanced fetch policies do not affect the IQ AVF reduction since there are less cache misses and thus the advanced fetch polices have less opportunity to take effect. When VISA+opt2 is employed, IPC increases on MIX workloads but decreases on MEM workloads when comparing DG and PDG baseline cases. The trend is similar to that of using ICOUNT as the default fetch policy (see Figure 5 (b)).

5. Dynamic Vulnerability Management (DVM) for IQ

The techniques we proposed in Section 2 aim to reduce IQ vulnerability. Nevertheless, there is no guarantee that the IQ vulnerability can be maintained within a given threshold. Managing microarchitecture runtime vulnerability is important since workload execution exhibits time varying behavior [10]. In this section, we further explore dynamic vulnerability management (DVM) schemes which are capable of keeping IQ vulnerability to a pre-set threshold.

5.1. The DVM Mechanism

A primary goal of DVM is to maintain IQ vulnerability to within a pre-defined reliability target during the entire program execution. Meanwhile, DVM should minimize its impact on the performance of program execution. To achieve both objectives, we proposed a dynamic mechanism (shown as Figure 7) to perform runtime IQ vulnerability management.

```
1.  DVM_IQ
2.  {
3.     ACE bits counter updating();
4.
5.     For every contexts
6.     {
7.        if current context has L2 cache misses
8.         then stall dispatching instructions for current context
9.     }
10.
11.    Every (sample_interval/5) cycles
12.    {
13.       if online IQ_AVF > trigger threshold
14.        then wq_ratio = wq_ratio/2
15.         else  wq_ratio = wq_ratio+1
16.    }
17.
18.    if (ratio of waiting instruction # to ready instruction # > wq_ratio)
19.    then stall dispatching instructions for all threads
20.
21.    if (all threads stall)
22.    if (online IQ_AVF < trigger threshold)
23.    then resume dispatching for contexts with most un-ACE instructions
24.
25. }
```

Figure 7. IQ DVM Pseudo Code

The first step in implementing DVM is to design trigger and response mechanisms. Our proposed scheme uses several microarchitecture and workload characteristics as trigger mechanism. Among them, the online IQ AVF is used to estimate runtime microarchitecture vulnerability. We use an ACE-bit

counter which accumulates the number of ACE bits that reside in the IQ every cycle during a sampled interval. The total number of ACE-bits in the IQ can be obtained using the instruction-level ACE/un-ACE classification described in Section 2.1. In this paper, we use 10K cycles as the size of each sampled interval. We then estimate the online AVF by dividing the ACE-bit counter value by the number of cycles and the total number of bits in the IQ. The estimated AVF is compared against a trigger threshold to determine whether it is necessary to enable a response mechanism. In addition to monitoring the online IQ AVF, we further optimize the trigger mechanism to achieve the desired reliability and performance tradeoffs. First, a L2 cache miss will immediately enable the response mechanism. This is because upon a L2 cache miss, the ACE bits contributed by instructions which have a dependence on the cache miss will remain in the IQ for at least hundreds of cycles. The increased ACE-bit resident cycles increase the IQ AVF significantly. Throttling the instruction dispatching to the IQ upon a L2 cache miss can effectively reduce IQ vulnerability. Additionally, we sample the IQ AVF at a finer granularity (e.g. five times within each interval) and compare the sampled AVF with the trigger threshold. If the IQ AVF exceeds the trigger threshold, a parameter *wq_ratio*, which specifies the ratio of number of waiting instructions to that of ready instructions in the IQ, is updated. The purpose of setting this parameter is to maintain the performance by allowing an appropriate fraction of waiting instructions in the IQ to exploit ILP. During program execution, the presence of few ready instructions in the IQ reflects the fact of limited ILP. Aggressively dispatching instructions to the IQ leads to higher IQ vulnerability while only having a small contribution to performance. By maintaining a desired ratio between the waiting instructions and the ready instructions, vulnerability can be reduced at negligible performance cost. The *wq_ratio* update is triggered by the estimated IQ AVF. In our DVM design, *wq_ratio* is adapted through slow increases and rapid decreases in order to ensure a quick response to a vulnerability emergency. If all threads stall due to L2 cache misses, the SMT processor can not make any progress and performance degradation will be significant. To avoid this situation, we restore instruction dispatching for threads that have the fewest ACE instructions in the fetch queue whenever the online AVF drops below the trigger threshold. The rationale is that in contrast to ACE instructions, un-ACE instructions contain many less vulnerable bits. Therefore, bringing un-ACE instructions into the IQ has little impact on reliability yet increases the ability of exploiting the inherent ILP in programs.

Another key design parameter for DVM is the trigger threshold. Trigger threshold is defined as the gate value at which the response mechanism starts to take effect. Since maintaining a reliability threshold is the final goal of DVM, the trigger threshold should be set appropriately to the final reliability target. If the trigger threshold is set too close to the reliability threshold, it will be too late to enable the response mechanism for keeping the vulnerability under control. On the contrary, if the trigger threshold is too far from the reliability threshold, the response mechanism will invoke prematurely, resulting in considerable performance degradation. After experimenting with a wide range of possible design choices, we set the trigger threshold to 90% of the reliability threshold. Since obtaining the ratio of waiting instructions to ready instructions (line 18 of Figure 7) involves an integer division, we perform one computation every 50 cycles.

5.2. An Evaluation of DVM

The pre-set threshold is an important factor that affects the efficiency of a DVM scheme. To evaluate our DVM design, we use various vulnerability thresholds. To determine the range of the vulnerability thresholds, we measure the maximum IQ AVF (Max_{IQ_AVF}) that programs yield during execution. We choose our reliability target from the range [0.3* Max_{IQ_AVF} , 0.7* Max_{IQ_AVF}] since this range provides a good representation of realistic vulnerability goals. If the target upper bound is set too high, the vulnerability reduction is of little significance. Conversely, if the lower bound is set too low, the usability of the system can be lost (e.g. the extreme case where the AVF target is set to 0 and no instructions are issued and no work is completed). We find that the above range provides a good tradeoff region between AVF values and realistic performance overheads. To quantify how well the proposed mechanism maintains runtime microarchitecture reliability, we use the percentage of vulnerability emergencies (PVE) metric, which counts the percentage of execution intervals in which IQ AVF exceeds the pre-set reliability target. We use IPC as a metric to quantify the performance impact of our DVM mechanism. We also report harmonic IPC [39] which takes fairness into consideration.

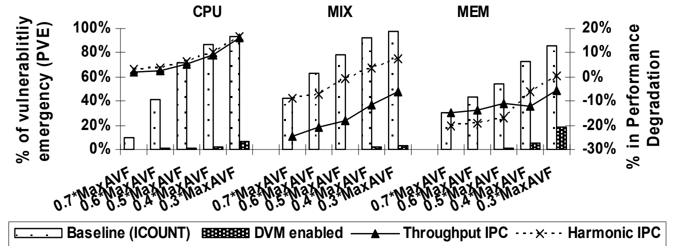


Figure 8. DVM efficiency and its impact on performance (fetch policy: ICOUNT)

Figure 8 shows the average results for the three types of workloads. We can see that the DVM mechanism we proposed is able to eliminate the majority of vulnerability emergency. For example, in the case in which the target reliability level is set to 0.5* Max_{IQ_AVF}, the PVEs for CPU, MIX and MEM workloads decrease from 72% to 1%, 79% to 1% and 55% to 1% respectively. Even if a very aggressive threshold (i.e., 0.3* Max_{IQ_AVF}) is used, on average, our DVM scheme is still successful in achieving the reliability target for 90% of the entire execution. Note that PVEs in MEM is high (19%) when DVM is triggered. A closer look at these intervals shows that the AVF of most of these intervals just slightly surpass the threshold. Experiments show that 11% of the 19% of intervals that experience vulnerability emergencies have AVFs that surpass the threshold by a maximum of 2%. As shown in Figure 8, the performance overhead increases as the reliability demand increases. This trend indicates that an aggressive target reliability level requires the invocation of the control mechanism more frequently which incurs a cost in performance. Setting an aggressive target threshold allows for a more reliable system, but this comes at the cost of performance. Although the trend is similar for all types of workloads, the absolute amount of performance overhead primarily depends on the workload types. For CPU workloads, DVM incurs some slowdown in throughput. Since CPU workloads are characterized by both high ILP existing within each individual thread and high TLP across all threads, performance loss is incurred during the period in which the DVM

response mechanism is throttling the instruction dispatched to the IQ for reliability control purpose. Therefore, an insufficient amount of instructions in the IQ restricts the pipeline's ability to explore both ILP and TLP, leading to a performance drop. However, with a mild reliability demand, such as 0.5* Max_{IQ_AVF}, our DVM technique achieves performance loss of less than 7% for CPU workloads. It is interesting to find that throughput IPCs increase on MIX and MEM workloads (represented by negative values in Figure 8) when the DVM scheme is applied. In the DVM response mechanism, instruction dispatching is stalled once a L2 cache miss occurs. Preventing fetching instructions from offending threads is beneficial for allocating IQ entries for other threads. However, throughput gains are gradually offset by penalties associated with increased reliability demands. Differing from MEM workloads, it is interesting to note that the degradation in harmonic IPC for MIX workloads is considerably larger than that of MEM workloads. As was discussed earlier, although stall dispatching benefits throughput IPC, the downside of this gain is the fairness problem. In contrasting to throughput, fairness is reduced due to a biasing toward CPU-bounded threads in MIX workloads, resulting in a more noticeable degradation in harmonic IPC. For MEM workloads, each thread makes progress evenly because the performance characteristics are similar across all threads. Therefore, the harmonic IPC trend is almost identical to that of throughput IPC.

Since FLUSH shows superior microarchitecture soft error mitigation capability than other fetch policies [5], it is no necessary to explore DVM if FLUSH can manage the dynamic reliability of IQ. Figure 9 shows simulation results when FLUSH is chosen as the baseline fetch policy. The results show that that DVM works well even if FLUSH policy is active concurrently.

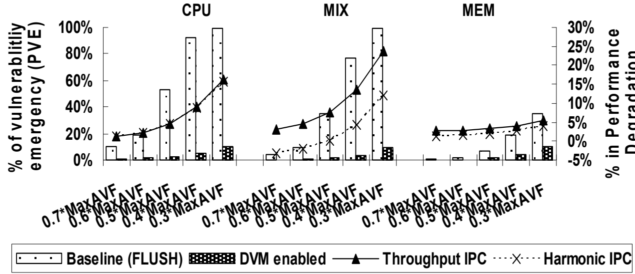


Figure 9. DVM efficiency and its impact on performance (fetch policy: FLUSH)

Figure 10 compares DVM with other reliability-aware optimizations. The first three bars presents the PVEs results using the reliability optimizations proposed in Section 2. DVM (static) is a variant of our DVM scheme in which the ratio is set statically. The first three bars show a very high PVE, which implies that the mechanism that we proposed earlier is incapable of managing runtime microarchitecture vulnerability. As for the static ratio approach, there is a broad range of values that can be selected as the ratio. To find an optimized value, we measure the average ratio in the dynamic approach and set the static ratio to this average. Results from the static approach show that it is capable of managing runtime reliability to a degree. On the other hand, our DVM relies on dynamically adapting the ratio to keep an optimal number of waiting instructions for both reducing AVF and maintaining performance. As a result, compared to the static method, the dynamic approach always outperforms the static, which is a result of its ability to adapt at runtime.

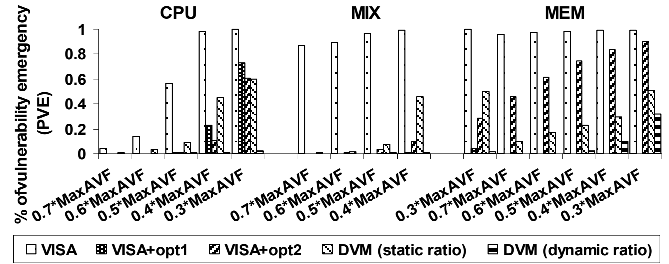


Figure 10. A comparison of DVM and other reliability optimization schemes

6. Related Work

In the past, there has been extensive research on instruction scheduling. To avoid the large issue queue size which degrades the clock cycle time, Lebeck et al. [11] proposed the waiting instruction buffer to hold instructions dependent on a long latency operation and reinsert them into a small size IQ when the operation completes. Various forms of dependence-based IQ design were proposed in [12, 15, 16], they generally sorted instructions following data-flow order in the partitioned IQ based on dependence chains, instructions thereby is issued in-order which reduces the complexity of issue logic and improves the clock frequency for large IQ window size. In [17], checkpoint processing and recovery microarchitecture is proposed to implement large instruction window processor without large cycle-critical buffers. Srinivasan et al. [18] found out critical load instructions by building a critical table. Tune et al. [19] and Fields et al. [20] independently proposed the prediction of critical path instructions which provides the opportunity of optimizing processor performance. Seng et al. [21] proposed to issue critical instruction in-order and slot them to fast function units to reduce power consumption. In our work, instruction is prioritized and scheduled based on their reliability criticality.

As power consumption has become an important consideration in processor design, researchers have also studied low power IQ design. Ponomarev et al. [22] proposed to dynamically adjust the IQ size based on the interval sampling of its occupancy. Folegnani et al. [23] partitioned issue queue into blocks, and disabled them while the IPC monitoring mechanism reports they have no contribution to IPC. Jones et al. [24] saved power via software assistance which dynamically resizes IQ based on compiler analysis. In [25], Brooks et al. explored the tradeoffs between several mechanisms for responding to periods of thermal trauma. With appropriate dynamic thermal management, the CPU can be designed for a much lower maximum power rating with minimal performance impact for typical applications. In [26], a novel issue queue is proposed to exploit the varying dynamic scheduling requirement of basic blocks to lower the power dissipation and complexity of the dynamic issue hardware. In [27], a circuit design of an issue queue is presented for a superscalar processor that leverages transmission gate insertion to provide dynamic low-cost configurability of size and speed. Karkhanis et al. [28] proposed to save energy by delivering instructions just-in-time, it inhibits instruction fetching when there are a certain number of in-flight instructions in the pipeline. Buyuktosunoglu et al. [29] combined fetch gating and issue queue adaptation to match the instruction window size with the application ILP characteristics and achieved energy saving. In our study, the dynamic resource allocation in

IQ is dependent on several important reliability-relevant feedback metrics (e.g. RQL) which are not applied in previous studies, and the methodology to implement the resource allocation is different from prior work. Additionally, DVM works on SMT architectures with several novel futures including wq_ratio adaptation, reliability-aware instruction dispatching and on-line AVF estimation.

There is a growing amount of work aimed at characterizing soft error behavior at the microarchitecture level [5, 30, 31, 32, 33, 34, 35]. 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 hurt the performance. Weaver et al. [36] proposed to selectively squash instructions when long delays are encountered to reduce IQ soft error vulnerability. Madan et al [13] proposed several mechanisms to save power in redundant multithreading (RMT), later in [14], they leveraged 3D technologies to tolerate soft errors. Walcott et al. [37] used a set of processor metrics to predict structure AVF, which is then applied to trigger RMT for structure's reliability maintenance. Soundararajan et al. [38] proposed dispatching stall and partial redundant thread to control the ROB AVF under certain threshold at cycle level. Our work is unique in its joint consideration of performance and reliability of IQ design on multithread environment without using redundant execution.

7. Conclusions

We have presented novel microarchitecture techniques designed to reduce the IQ vulnerability to soft errors on SMT processors. The key observation is that the number of vulnerable instructions that are ready to execute on SMT processors is much higher than that on superscalar processors. The IQ vulnerability to soft error can be reduced by assigning vulnerable instructions a higher issue priority. This has the effect of reducing the number of ACE-bit resident cycles and thus the vulnerability of the IQ. We further apply reliability-aware dynamic resource allocation to the IQ to prevent excessive vulnerable bits from entering the IQ. Results from the implementation and evaluation of the proposed techniques show 42% reliability improvement in the IQ with 1% performance improvement. To maintain runtime IQ reliability for pre-defined thresholds, we propose dynamic vulnerability management (DVM) mechanisms. The proposed DVM scheme can effectively achieve reliability/performance tradeoffs. For example, when reliability threshold 0.5* Max_{IQ_AVF} is set, it reduces percentage of vulnerability emergencies (PVE) to less than 1% for all types of SMT workloads with a negligible performance cost in most cases. In this paper we focus on the IQ, however we believe our technique could be extended to other microarchitecture structures.

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] S. Mitra, N. Seifert, M. Zhang, Q. Shi and K. S. Kim, Robust System Design with Built-In Soft-Error Resilience, Computer, Vol. 38, No. 2, page 43-52, Feb. 2005.
[2] D. Tullsen, S. Eggers and H. Levy, Simultaneous Multithreading: Maximizing On-Chip Parallelism, ISCA, 1995.
[3] D. Marr, F. Binns, D. Hill, G. Hinton, D. Koufaty, J. Miller and M. Upton, Hyper-Threading Technology Architecture and Microarchitecture, Intel Technology Journal, 6(1), Feb. 2002.
[4] 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, MICRO, 2003.
[5] W. Zhang, X. Fu, T. Li and J. Fortes, An Analysis of Microarchitecture Vulnerability to Soft Errors on Simultaneous Multithreaded Architectures, ISPASS, 2007.
[6] J. Sharkey, D. Ponomarev, Efficient Instruction Schedulers for SMT processors, In Proceedings of the International Symposium on High Performance Computer Architecture, 2006.
[7] D. Tullsen, S. Eggers, J. Emer, H. Levy, J. Lo and R. Stamm, Exploiting Choice: Instruction Fetch and Issue on an Implementable Simultaneous Multithreading Processor, ISCA, 1996.
[8] D. Tullsen and J. Brown, Handling Long-latency Loads in a Simultaneous Multithreading Processor, MICRO, 2001.
[9] A. El-Moursy and D. H. Albonesi, Front-end Policies for Improved Issue Efficiency in SMT Processors, HPCA, 2003.
[10] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, Automatically Characterizing Large Scale Program Behavior, ASPLOS, 2002.
[11] A. R. Lebeck, J. Koppanalil, T. Li, J. Patwardhan and E. Rotenberg, A Large, Fast Instruction Window for Tolerating Cache Misses, ISCA, 2002.
[12] S. Raasch, N. Binkert and S. Reinhardt, A Scalable Instruction Queue Design using Dependence Chains, ISCA, 2002.
[13] Niti Madan, Rajeev Balasubramonian, Power Efficient Approaches to Redundant Multithreading, IEEE Transactions on Parallel and Distributed Systems, 2007.
[14] Niti Madan, Rajeev Balasubramonian, Leveraging 3D Technology for Improved Reliability, MICRO, 2007.
[15] S. Palacharla, N. P. Jouppi, J. E. Smith, Complexity-Effective Superscalar Processors, ISCA, 1997.
[16] P. Michaud and A. Seznec, Data-flow Prescheduling for Large Instruction Windows in Out-of-order Processors, HPCA, 2001.
[17] H. Akkary, R. Rajwar, S. T. Srinivasan, Checkpoint Processing and Recovery: Towards Scalable Large Instruction Window Processors, MICRO, 2003.
[18] S. T. Srinivasan, R. D. Ju, A. R. Lebeck, C. Wilkerson, Locality vs. Criticality, ISCA, 2001.
[19] B. Fields, S. Rubin, R. Bodik, Focusing Processor Policies via Critical-Path Prediction, ISCA, 2001.
[20] E. Tune, D. Liang, D. M. Tullsen, B. Calder, Dynamic Prediction of Critical Path Instructions, HPCA, 2001.
[21] J. S. Seng, E. S. Tune, D. M. Tullsen, Reducing Power with Dynamic Critical Path Information, MICRO, 2001
[22] D. Ponomarev, G. Kucuk and K. Ghose, Reducing Power Requirements of Instruction Scheduling through Dynamic Allocation of Multiple Datapath Resources, MICRO, 2001.
[23] D. Folegnani and A. Gonz´alez, Energy-Effective Issue Logic, ISCA, 2001.
[24] T. M. Jones, M. F. P. O'Boyle, J. Abella and A. Gonzalez, Software Directed Issue Queue Power Reduction, HPCA, 2005.
[25] D. Brooks and M. Martonosi, Dynamic Thermal Management for High-Performance Microprocessors, HPCA, 2001.
[26] M. G. Valluri, Lizy K. John and K. S. McKinley, Low-power, Low-complexity Instruction Issue using Compiler Assistance, ICS, 2005.
[27] A. Buyuktosunoglu, S. Schuster, D. Brooks, P. Bose, P. W. Cook and D. H. Albonesi, An Adaptive Issue Queue for Reduced Power at High Performance, PACS, 2000.
[28] T. Karkhanis, J. E. Smith, P. Bose, Saving Energy with Just In Time Instruction Delivery, ISLPED, 2002.
[29] A. Buyuktosunoglu, T. Karkhanis, D. H. Albonesi, P. Bose, Energy Efficient Co-Adaptive Instruction Fetch and Issue, ISCA, 2003.
[30] A. Biswas, R. Cheveresan, J. Emer, S. S. Mukherjee, P. B. Racunas and R. Rangan, Computing Architectural Vulnerability Factors for Address-Based Structures, ISCA, 2005.
[31] N. J. Wang, J. Quek, T. M. Rafacz and S. J. Patel, Characterizing the Effects of Transient Faults on a High Performance Processor Pipeline, DSN, 2004.
[32] H. Asadi, V. Sridharan, M. B. Tahoori, D. Kaeli, Balancing Performance and Reliability in the Memory Hierarchy, ISPASS, 2005.
[33] X. Li, S. V. Adve, P. Bose, J.A. Rivers, SoftArch: An Architecture-Level Tool for Modeling and Analyzing Soft Errors, DSN, 2005.
[34] X. Fu, J. Poe, T. Li and J. Fortes, Characterizing Microarchitecture Soft Error Vulnerability Phase Behavior, MASCOTS, 2006.
[35] S. Kim and A. K. Somani, Soft Error Sensitivity Characterization of Microprocessor Dependability Enhancement Strategy, DSN, 2002.
[36] C. Weaver, J. Emer, S. S. Mukherjee and S. K. Reinhardt, Techniques to Reduce the Soft Error Rate of a High Performance Microprocessor, ISCA, 2004.
[37] K. R. Walcott, G. Humphreys, S. Gurumurthi, Dynamic Prediction of Architectural Vulnerability from Microarchitectural State, ISCA, 2007.
[38] N. Soundararajan, A. Parashar, A. Sivasubramaniam, Mechanisms for Bounding Vulnerabilities of Processor Structures, ISCA, 2007.
[39] K. Lou, J. Gummaraju, M. Franklin, Balancing Throughput and Fairness in SMT Processors, In Proceedings of the International Symposium on Performance Analysis of Systems and Software, 2001.