Opportunistic Transient-Fault Detection

Mohamed A. Gomaa and T. N. Vijaykumar
School of Electrical and Computer Engineering, Purdue University
{gomaa, vijay}@ecn.purdue.edu

Abstract

CMOS scaling increases susceptibility of microprocessors to transient faults. Most current proposals for transient-fault detection use full redundancy to achieve perfect coverage while incurring significant performance degradation. However, most commodity systems do not need or provide perfect coverage. A recent paper explores this leniency to reduce the soft-error rate of the issue queue during L2 misses while incurring minimal performance degradation. Whereas the previous paper reduces soft-error rate without using any redundancy, we target better coverage while incurring similarly-minimal performance degradation by opportunistically using redundancy. We propose two semi-complementary techniques, called partial explicit redundancy (PER) and implicit redundancy through reuse (IRTR), to explore the trade-off between soft-error rate and performance. PER opportunistically exploits low-ILP phases and L2 misses to introduce explicit redundancy with minimal performance degradation. Because PER covers the entire pipeline and exploits not only L2 misses but all low-ILP phases, PER achieves better coverage than the previous work. To achieve coverage in high-ILP phases as well, we propose implicit redundancy through reuse (IRTR). Previous work exploits the phenomenon of instruction reuse to avoid redundant execution while falling back on redundant execution when there is no reuse. IRTR takes reuse to the extreme of performance-coverage trade-off and completely avoids explicit redundancy by exploiting reuse's implicit redundancy within the main thread for fault detection with virtually no performance degradation. Using simulations with SPEC2000, we show that PER and IRTR achieve better trade-off between soft-error rate and performance degradation than the previous schemes.

1 Introduction

CMOS scaling continues to enable faster transistors and lower supply voltage, improving microprocessor performance and reducing per-transistor power. Unfortunately, the downside of scaling is the increased susceptibility of microprocessors to soft errors due to strikes by cosmic particles and radiation from packaging materials. The result is degraded reliability in future commodity microprocessors.

Many techniques [14, 12] provide soft-error detection by redundantly running two copies of the program and comparing their results. These full-redundancy techniques achieve high soft-error coverage at the cost of significant performance degradation. Though perfect coverage would be required in highly-reliable servers and life-protecting custom systems used in hostile environments such as Space missions, the rest of the market does not need perfect coverage. This observation is borne out by the industry practice of employing full-blown redundancy *only* in specialized servers (e.g., Tandem Non-stop [3]), and not in desktops and commodity servers. Except for the select minority of the specialized servers and custom systems, the rest of the systems trade-off performance and reliability to arrive at acceptable design points which provide reasonable, *but not perfect*, fault tolerance. As such, the high performance degradation of the full-redundancy techniques may be unacceptable, given the lack of need for perfect coverage in most commodity systems.

The idea of partial coverage should be understood in the context of technology scaling. Soft-error rates increase exponentially over technology generations [16]. If we assume a technology node where the soft-error rate is so high that it is barely within acceptable levels, then techniques that achieve partial coverage would run out of steam within a few generations. For example, assuming that soft-error rate doubles every generation, a 75% coverage would keep the soft-error rate within acceptable levels only for the next generation beyond which the rate would increase by a factor of 1.5 and exceed acceptable levels. It is clear that in such extreme scenarios, close to full coverage will be needed and partial coverage will not suffice. However, current soft-error rates are nowhere near such prohibitive levels and will not be so for many generations to come even under the exponential trend. As such, in the intermediate generations soft errors will be an increasing but not a prohibitive problem (e.g., soft errors will cause an increasingly non-negligible number of crashes in a data center and lead to substantial financial loss). In the intermediate generations, employing partial coverage to achieve a good reliability-performance trade-off makes sense.

A recent paper [25] explores the reliability-performance trade-off to improve the soft-error rate (SER) of high-performance microprocessors by reducing, *not eliminating*, the architectural vulnerability [8] of the issue queue while incurring minimal performance degradation. The paper observes that the instructions that wait in the issue queue during L2 (long-latency, off-chip) misses are much more susceptible to faults than the instructions that move through the issue queue without stalling. Accordingly, the paper proposes to squash the instructions in the issue queue upon such a miss and to stop processor activity until the miss returns. Because the processor generally ends up stalling during such misses despite out-of-order issue, the squashing and stopping do not significantly degrade performance. Though the performance degradation is small, the technique reduces the vulnerability only of the issue queue and not of the rest of the pipeline (e.g., functional units, rename logic, bypass logic, register file).

Whereas [25] reduces soft-error rate without using any redundancy, we target better coverage while incurring similarly-minimal performance degradation by opportunistically using redundancy. We propose two semi-complementary techniques, called *partial explicit redundancy* (*PER*) and *implicit redundancy through reuse* (*IRTR*), to explore the trade-off between soft-error coverage and performance. For PER, we make the key observation that previous full-redundancy techniques incur performance degradation mostly because the redundant thread competes with the main thread for processor resources. Given the lack of need for perfect coverage, we propose to introduce *explicit* redundancy *only when* the redundant instructions would not compete with the main thread. Whenever the main thread proceeds at full speed and needs all of the resources (e.g., during high-ILP phases), we do not introduce redundancy and there is no soft-error coverage, but there is no performance degradation either. However, when the main thread is stalled and under-utilizes the resources (e.g., during cache misses or low-ILP phases), redundant instructions are executed to provide some soft-error coverage with minimal performance degradation.

Rather than empty the pipeline whenever there is a long-term stall as done in [25], we *opportunistically* exploit the pipeline's idle resources to introduce redundancy. The key advantage of PER over [25] is increased soft-error coverage while maintaining comparable performance degradation. PER's increased coverage comes from increases in both spatial and temporal coverage. Our redundancy scheme increases spatial coverage by covering soft errors in all of the pipeline whereas [25] covers only the issue queue. We increase temporal coverage by exploiting all low-activity durations (due either to cache misses or to inherently low-ILP phases in the program) while [25] leverages only cache misses. Compared to previous redundancy-based schemes [14, 12], our advantage is significantly lower performance degradation while maintaining reasonable coverage.

The extent of our soft-error coverage depends on how much the main thread under-utilizes the pipeline. However, we still wish to achieve some coverage without worsening performance in the complementary cases of when the utilization is high. To that end, we employ our second technique, implicit redundancy through reuse (IRTR). A previous paper [19] discovered the phenomenon of *instruction reuse* (*IR*) where an instruction is re-executed many times with the same input values. The paper exploits IR to avoid re-execution of a reuse instruction by storing the inputs and output of the instruction's instance in the *reuse buffer* and looking up the output for a later instance if the *reuse test* succeeds—i.e., the input values of the two instances match. Misses or failed reuse tests result in updates of the buffer. Avoiding re-execution allows execution resources to be used by other instructions and achieves performance improvement. A recent paper [9] observes that IR can be used to reduce the pressure on execution resources in full-redundancy schemes. The paper proposes dual instruction execution instruction-reuse buffer (DIE-IRB) to avoid executing the redundant thread's later instances by reusing its earlier instance. Upon reuse buffer hits, the buffered output is re-used instead of executing the redundant thread and compared to the main thread for fault detection. Misses or failed reuse tests trigger redundant execution. The buffer is updated after the main and redundant executions are compared.

We take reuse to the extreme of performance-coverage trade-off in IRTR: we make the key observation that this comparison, and consequently the redundant execution, is unnecessary for less-than-perfect coverage. The buffer can be updated using the main thread without any comparison. A subsequent reuse buffer hit accompanied by the checking of the instruction's output against the buffered output suffices for fault detection. The reuse test done for the hit automatically checks the inputs, and the explicit checking covers the output. Thus, reuse detects *implicit* redundancy *within* the main thread and this implicit redundancy allows some coverage without any explicit redundancy and therefore, without worsening performance. Upon misses or failed reuse tests, however, there is no redundant execution against which to compare the main thread, unlike DIE-IRB. Consequently, misses and failed reuse tests result in loss of coverage but there is no performance loss because there is no explicit redundancy. Thus, whereas IRTR exploits the lack of need for perfect coverage to achieve some coverage at virtually no performance loss, DIE-IRB does not and incurs performance loss for coverage that may not be needed. In addition, DIE-IRB slows down the issue queue by adding extra fields whereas IRTR does not need to modify the issue queue in any way. Finally, IRTR complements PER by achieving some coverage without worsening performance even when pipeline utilization is high.

The main contributions of this paper are:

- our two opportunistic techniques, partial explicit redundancy and implicit redundancy through reuse;
- using simulations with SPEC2000, we show that average processor soft-error-rate reduction and performance loss for DIE-IRB adapted to Simultaneous and Redundantly Threaded (SRT) processors [12], squashing on L2 miss, and combined PER and IRTR are 100% and 15%, 4% and 3.5%, 56% and 2%, respectively. Put differently, our techniques can reduce the number of crashes caused by soft errors (described in the third paragraph of this section) by a factor of two at little performance degradation. Our techniques achieve better soft-error rate reduction than the previous schemes at minimal performance degradation.

The rest of the paper is organized as follows. Section 2 discusses redundancy-based schemes—SRT and PER. Section 3 discusses reuse-based schemes—DIE-IRB and IRTR. Section 4 combines PER and IRTR. Section 5 describes methodology and Section 6 presents results. Section 7 discusses related work, and Section 8 concludes.

2 Redundancy-Based Fault Detection

There have been a few techniques to provide transient-fault detection by running two copies of the program and comparing the instructions in the copies. We base our schemes on one of them, namely, Simultaneous and Redundantly Threaded (SRT) processors [12]. Before we explain our schemes, we give a brief background on SRT.

2.1 SRT: A Full-Redundancy scheme

SRT uses SMT's ability to process simultaneously multiple threads and executes both the main thread, called the leading thread, and the redundant thread, called the trailing thread, on the same core. SRT assumes that the memory hierarchy is protected by ECC and that registers are not.

To be able to detect faults, the two threads should not diverge when there are no faults. Though divergence is not a problem for most instructions, duplicating cached loads is problematic because memory locations may be modified by an external agent (e.g., another processor during multiprocessor

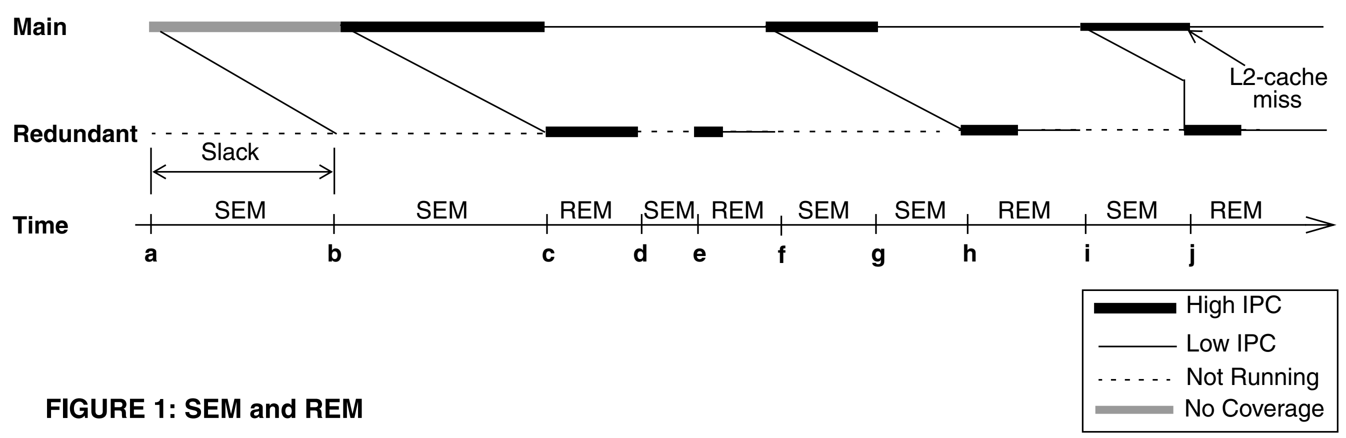

FIGURE 1: SEM and REM

synchronization) between the time the leading thread loads a value and the time the trailing thread tries to load the same value. The two threads may diverge if the loads return different data. SRT allows only the leading thread to access the cache, and uses the Load Value Queue (LVQ) to hold the leading load values. The trailing thread loads from the LVQ instead of repeating the load from the cache. Only load values are obtained non-redundantly, load addresses are computed redundantly in each thread and compared at the time of accessing the LVQ. The LVQ is protected from soft errors by ECC.

SRT allows both threads to update their *respective* copies of register state, but allows only one copy of stores to go the memory hierarchy. SRT compares the two copies of each store before the store is committed to the memory hierarchy. Register writes, however, are committed in the respective threads without any checking. Because an incorrect value caused by a fault propagates through computations and is eventually consumed by a store, checking *only* stores suffices for soft-error detection. SRT uses the store buffer (StB) to hold the leading store address and value until the trailing thread executes the store and the stores are compared.

SRT employs two key optimizations to enhance performance: The leading thread runs ahead of the trailing thread by a long *slack* (e.g., 256 instructions), and provides the leading branch outcomes to the trailing thread through the Branch Outcome Queue (BOQ). The slack *completely* hides the leading thread's cache misses and branch mispredictions from the trailing thread. By the time the trailing thread needs a load value or a branch outcome, the leading thread has already produced it—the load value is available in the LVQ and the branch outcome in the BOQ. Note the difference between the load value and the branch outcome. The trailing thread consumes the load value without redundantly performing the load, whereas the branch outcome is used as a prediction by the trailing thread which redundantly executes the branch and verifies that the outcome is fault-free. Using the leading thread's branch outcomes *entirely* eliminates branch mispredictions in the trailing thread (in the fault-free case), and reduces the number of trailing instructions.

Although SRT provides high soft-error coverage, it incurs significant performance degradation which may be unacceptable especially because of lack of need for perfect coverage, as mentioned in Section 1. This degradation is because (1) even though the redundant thread incurs no branch mispredictions and executes fewer instructions than the main thread, the redundant thread still competes with the leading thread for pipeline resources; and (2) the checking of stores via the StB causes leading-thread stalls whenever the StB fills up.

2.2 Partial Explicit Redundancy

To reduce this degradation, we employ partial explicit redundancy (PER) which allows the main thread to use all the resources it needs during high-IPC phases at the cost of no soft-error coverage. However, when the main thread under-utilizes the available resources, during low-IPC phases or L2-cache misses, the redundant thread runs providing soft-error coverage with minimal performance degradation, as illustrated in Figure 1. We call the mode in which only the main thread executes as the Single Execution Mode (SEM) whereas the mode in which the redundant thread executes simultaneously with main thread as the Redundancy Execution Mode (REM).

PER differs from SRT in that PER's trailing thread has gaps in its execution in that it does not include all the instructions. Other than this difference, PER is similar to SRT in using the LVQ for load values, BOQ for branch outcomes, and StB for holding stores. PER also uses the slack but with a small difference from SRT, as we explain later. Like SRT, PER assumes that the memory hierarchy is ECC-protected but the registers are not.

PER's execution gaps raise three main challenges: First, in PER, unlike SRT, there is a problem checking only stores. Because PER provides only partial redundancy, it may not duplicate the store to which a fault propagates although it may duplicate the instruction where the fault originated, or vice versa. The fault will be caught only if the originating instruction, the store, and all the intervening instructions that propagate the fault are duplicated. Because the probability that all these instructions are duplicated in PER's partial redundancy is low, checking only stores will likely result in low coverage. Therefore, PER checks all instructions in REM. We validate this decision in our results, by showing that PER incurs minimal performance loss while achieving good coverage. Second, when the redundant thread resumes after a gap (e.g., at time *c* in Figure 1), it needs the program state corresponding to the resume point. Finally, we need to determine when to switch from SEM to REM and back.

2.2.1 SEM and REM in PER

The key issue with the first challenge is that checking all instructions requires high-bandwidth buffers to avoid performance degradation. If coverage were the only issue then avoiding the complexity of the buffers may be an option for systems that can get by with low coverage. In PER, however, we need

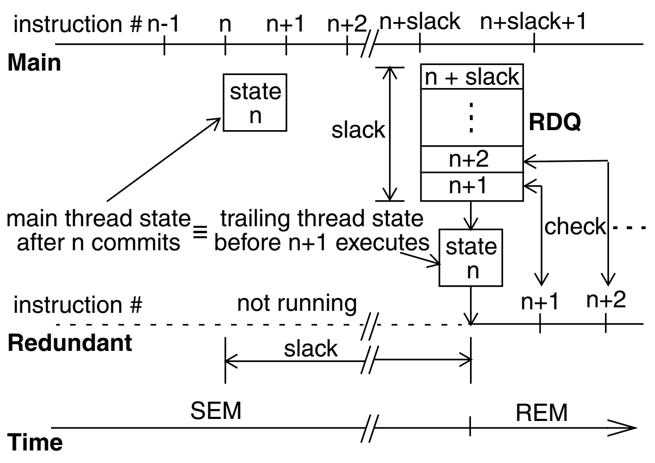
instruction # n-1 n n+1 n+2 n+slack n+slack+1
Main
state n
slack
n + slack
RDQ
n+2
n+1
main thread state after n commits ≡ trailing thread state before n+1 executes
check
state n
instruction #
not running
n+1 n+2
Redundant
slack
SEM
REM
Time

FIGURE 2: RDQ operation

the buffers anyway to handle the second challenge of providing the trailing thread with resume-point state. These buffers represent a performance-complexity trade-off point: On one hand, SRT avoids this complexity because it employs full redundancy and does not have gaps in the redundant thread. On the other, SRT's full redundancy results in performance degradation which our scheme reduces while incurring the complexity. We address the second challenge by using the main thread to provide the redundant thread with the resume-point state. Recall that our technique accepts less-than-perfect coverage to trade-off coverage for performance. Obtaining the state from the main thread does not degrade coverage any more than having gaps in the redundant thread. However, providing the main thread's state to the redundant thread has one slight complication: The slack between the threads implies that at the time the redundant thread resumes, the main thread's state is ahead of the resume point by the amount of the slack (see Figure 2). To obtain the main thread's processor state (registers, condition codes) corresponding to the resume point, we use the register delay queue (RDQ). The main thread inserts into the RDQ all instructions' output state, including register values, branch outcomes, load values and addresses, store values and addresses, as they commit. Essentially, the RDQ subsumes SRT's LVQ, BOQ, and StB. As entries are removed from the RDQ they update the redundant thread's processor state (in SMT, the two threads' processor state are separate). The RDQ is sized to match the slack, so that the redundant thread's state lags behind the main thread's state by the slack amount and matches the resume point (as shown in Figure 2).

In both SEM and REM the RDQ entries update the redundant thread's processor state with one minor difference: In SEM, the entries empty when new entries are inserted by the main thread. In REM, the entries are emptied only after the redundant thread has checked them (see Figure 2). Thus, the RDQ serves dual purposes of addressing the first issue of allowing checking of all instructions and not just store instructions, as well as the second challenge of providing resume-point state. In SEM the RDQ does not affect the main thread's performance because there is no holding up of instructions. In REM the main thread has to stall if the RDQ is full. However, such stalling is rare because the redundant thread is much faster than the main thread due to slack and BOQ optimizations, as discussed in Section 2.1.

Obtaining the resume-point memory state is trivial because PER uses the LVQ for load values. For stores, PER first checks the store copies and then commits the store to the memory hierarchy, like SRT.

There is one important implementation detail remaining. Conceptually, the entries in the RDQ should hold only committed instructions. However, in modern pipelines register values and load values are available only in the register file at the time of commit. As such adding ports to the register file to allow the RDQ to obtain the values would add complexity and latency to the already-belabored register file. Therefore, we adopt the solution proposed in [24] which faces the same problem. [24] uses the register value queue (RVQ) to hold register values as instructions writeback their values. Upon commit, the values can be transferred from the RVQ to the RDQ. The RVQ supports clean-up after branch mispredictions and other squashes.

Though the RDQ can be avoided by eliminating the slack (then the main thread's instructions would directly update the processor state of the redundant thread), the significant performance benefit of the slack makes this option substantially inferior.

2.2.2 Role of slack in SEM and REM

In SEM, the main thread executes similarly to a single-thread superscalar. REM is similar to SRT except for one difference in the slack. SRT *always* tries to maintain the slack between the leading and trailing threads. SRT modifies SMT's ICOUNT [23] to implement the slack. In REM, however, always maintaining the slack would cause the following difficulty. We use the catching up of the redundant thread with the main thread as a trigger to check whether we should switch from REM to SEM, as we explain in the next section. If we were always to maintain the slack in REM, the trigger would never occur and we would not switch to SEM even when redundancy causes performance loss. Therefore, we do not maintain the slack in REM. We allow the redundant thread to catch up with the main thread and consume the slack (i.e., empty the LVQ, RDQ, BOQ, and StB and let the slack reduce to zero). We then suspend the redundant thread till the slack is built up by the main thread (i.e., fill up the queues), as shown in Figure 1 at times *d* and *f*. While consuming the slack, we use the unmodified ICOUNT to fetch the main and redundant threads in REM. At the end of the build-up phase, we determine if execution should continue in REM (at time *e* in Figure 1) or switch to SEM (at time *g* in Figure 1), depending on the conditions discussed in the next section. Thus, in REM we alternate between consuming the slack and building up the slack. In SRT, in contrast, the consumption and subsequent build-up of the slack occurs only naturally and not by design. For instance, a long-latency cache miss in the leading thread results in the trailing thread catching up, followed by the leading thread building up the slack when the miss returns. In steady state (i.e., no cache misses), however, SRT maintains the slack whereas we do not.

2.2.3 Switching between SEM and REM

The final challenge is to decide when to switch between SEM and REM. On one hand, staying in SEM while the main thread is in a low-ILP phase (i.e., processor resources are under-utilized) reduces soft-error coverage. On the other hand, staying in REM while the main thread is in a high-ILP phase degrades performance. Therefore, we need to detect the amount of ILP and adjust accordingly.

We first consider switching from SEM to REM. This switch should occur if the main thread is under-utilizing its resources. Under-utilization occurs under two conditions: low-ILP phase (e.g., at time *c* in Figure 1) and soon after an L2 miss (e.g., at time *j* in Figure 1). Because L2 misses are long, even high-ILP phases suddenly drop to low ILP soon after an L2 miss. Monitoring the long-term ILP of the phase as required by the first condition does not catch this sudden drop. Therefore, we explicitly include the second condition.

We detect the first condition by stipulating that *execute-IPC of the main thread + execute-IPC of the redundant thread is less than the processor issue width*, where execute-IPC includes all the executed instructions (speculative and non-speculative instructions). Because resource usage is more accurately reflected by execute-IPC and not commit-IPC, we use execute-IPC. While in SEM, we determine the main thread's execute-IPC by monitoring the number of instructions executed over a sampling time interval (e.g., 10 cycles). However, determining the redundant thread's execute-IPC is harder because the redundant thread does not run in SEM. Therefore we make an estimate based on the following reasoning: Because the redundant thread does not mispredict (Section 2.1), its execute-IPC equals its commit-IPC, averaged over a long time. Furthermore, because the main thread provides load values and branch outcomes to the redundant thread, the commit-IPC of the redundant thread equals the commit-IPC of the main thread, averaged over a long time. Note that the instantaneous commit-IPC of the redundant thread would be higher than the main threads's commit-IPC because the slack hides the main thread's latencies from the redundant thread. However, here we wish to see the long-term, and not the instantaneous, commit-IPC of the redundant thread. Consequently, we evaluate the inequality: *execute-IPC of the main thread + commit-IPC of the main thread < the processor's issue width,* and switch to REM if the inequality holds (e.g., at time *c* in Figure 1), or continue in SEM otherwise (e.g., between times *a* and *c* in Figure 1).

In addition to the above condition, execution switches from SEM to REM only if the RDQ is full (e.g., at time *c* in Figure 1). It may not be full because execution switched from REM to SEM back to REM before the main thread had a chance to build up the slack—i.e., fill up the RDQ (e.g., at time *f* in Figure 1). This additional condition ensures that the redundant thread has a full slack so that any delays in the main thread are hidden from it (e.g., in Figure 1, execution does not switch to REM at time *f* and switches to REM at time *h* after the slack builds up).

The second condition to trigger switching from SEM to REM is an L2 miss (e.g., at time *j* in Figure 1). Unlike the first condition which requires estimating ILP, the second condition is straightforward because an L2 miss is known from the cache without any estimates. However, there is one difficulty. By the time the cache returns the miss, the pipeline front end and the issue queue are already filled up by instructions that are (transitively) dependent on the missing load, as observed for a general SMT in [22]. As such even if we decide to switch to REM, the redundant thread cannot enter. Therefore, we squash the main thread removing its instructions from the pipeline so that the redundant thread can enter, as proposed in [22] to improve general SMT throughput.

Unlike the first condition above, an L2 miss causes execution switches from SEM to REM even if the RDQ is not full. If we wait for the main thread to build up the slack, the issue queue is likely to fill up undermining the reasoning behind our squashing. Therefore, execution switches to REM immediately after an L2 miss(e.g., at time *j* in Figure 1).

Now we consider switching from REM to SEM. Unfortunately, the inequality for the first condition above cannot be reversed to switch back to SEM. In REM, the execute-IPC of the main thread is affected by the presence of the redundant thread. As long as execution remains in REM, there is no easy way to estimate what the main thread's execute-IPC would be running by itself. Our solution is to introduce a sampling period where the main thread runs by itself and the redundant thread is suspended temporarily—that is, execution temporarily returns to SEM. At the end of the sampling period, PER tests the inequality to decide whether to resume REM or to continue in SEM. A convenient point to start the sampling is when the redundant thread has consumed the slack (i.e., has caught up with the main thread) in REM (e.g., at time *d* in Figure 1). This catching up occurs because the redundant thread's instantaneous commit-IPC is higher than that of the main thread (as explained above). After catching up, the redundant thread waits anyway for the main thread to build up the slack and fill the queues with values. Therefore, temporarily suspending the redundant thread to allow the sampling does not unduly slow down the redundant thread. During the sampling period, the main thread builds up the slack, *in addition to* providing IPC samples that are unaffected by the redundant thread. The built-up slack is used to resume the redundant thread if PER decides to continue execution in REM (e.g. at time *e* in Figure 1). Upon resumption, the redundant thread quickly consumes the slack due to its high instantaneous IPC. Consequently, the temporary suspension hardly impacts the redundant thread or soft-error coverage.

As mentioned in Section 1, [25] also squashes the pipeline on an L2 miss to avoid exposing the instructions waiting for the miss to particle strikes. The paper claims performance degradation to be minimal in an in-order-issue pipeline. Our evaluations, however, are in an out-of-order-issue pipeline which, unlike an in-order-issue pipeline, can overlap some instructions with a miss for high-ILP programs. Because [25] does not resume fetching until the miss returns, such overlap is lost causing performance loss. In contrast, because PER suspends the redundant thread upon slack consumption and allows the main thread to proceed, PER achieves overlap of the main thread's instructions with the miss. Consequently, PER performs better than [25]. In addition, [25] reduces the architectural vulnerability of only the issue queue whereas PER covers all of the pipeline. Also [25] exploits only L2 misses whereas PER exploits low-ILP phases as well. Therefore, our coverage is better. However, the extra coverage comes at a cost of some additional complexity as compared to [25].

PER covers the entire pipeline including the register file in REM. For register values that are redundantly stored in the register file, either via the update of the redundant thread state or through execution in REM, a fault in one of the copies will be detected. However, faults affecting non-redundant register values will go undetected.

3 Reuse-Based Fault Detection

PER's soft-error coverage depends on how much the main thread under-utilizes the pipeline. We employ our second technique, implicit redundancy through reuse (IRTR), to achieve some coverage without worsening performance in the complementary cases of when the utilization is high.

3.1 Background

Dynamic Instruction Reuse (or IR) was originally proposed to improve performance by not re-executing the instructions that have been executed before with the same inputs [19]. In such cases, IR simply uses the previous result stored in a *reuse buffer* (RB). Such re-execution occurs due to branch misprediction squashes which squash not only the control-flow dependent incorrect segment but also the post-if-else-reconvergent segment which is control-flow independent. The independent segment would re-execute with possibly the same inputs. Another example is simply repeated execution of code segments (due to loop or repeated calling of the same function) whose operands change infrequently for a particular user input set.

[19] proposes three implementation schemes for IR—one based on register values (scheme S_v), another on register names (scheme S_n), and the third based on register dependencies (scheme S_{n+d}). We use the value-based implementation because it is the simplest. In this implementation, the RB holds input and output values for instructions to check if their current inputs match the stored inputs and use the stored output upon match. Because the RB is indexed by PC, this scheme checks for reuse among different dynamic instances of the same static instruction. If the RB misses or if the reuse test fails—i.e., the current inputs do not match the inputs stored in the RB entry—then the RB entry is updated with the current inputs and output. For loads there is one complication: because the RB holds the load addresses and values, a store writing to a matching address has to invalidate the matching RB entries. We refer the reader to [19] for descriptions of the other implementation schemes.

A recent paper [9] observes that IR can be used to reduce the pressure on execution resources in a full-redundancy scheme. This paper presents a recovery scheme, whereas our paper targets detection. However, this difference is not important for this discussion because the reuse part is common irrespective of detection or recovery. The paper uses a baseline full-redundancy-based fault-recovery scheme called dual instruction execution (DIE) [11]. Instead of fetching redundantly, DIE duplicates the fetched instruction and inserts them *together* into the issue queue. Consequently, DIE does not exploit the slack and BOQ optimizations. Because of lack of BOQ, DIE redundantly executes *both* correctly-speculated and misspeculated instructions and the pressure on the functional units is far worse in DIE than in SRT. In addition, the lack of slack makes DIE perform poorly. [9] alleviates the functional-unit pressure by using IR so that reused instructions can lookup the results in the RB instead of executing on the functional units. This scheme is called DIE-IRB.

DIE-IRB uses the value-based IR implementation described above in its fault-detection scheme. In DIE-IRB, the main thread does not benefit from IR at all and executes without any changes; only the redundant thread uses IR. If there is a reuse buffer hit, the redundant instruction is removed from the issue queue and not executed. Instead, the buffer value is compared against the main instruction for fault detection. Misses or failed reuse tests trigger redundant execution. The buffer is updated after the main and redundant executions are checked.

Unfortunately, there are some complications in DIE-IRB's implementation. The decision to avoid executing the redundant instruction must occur before the redundant instruction issues. This constraint implies that DIE-IRB needs the redundant instruction's inputs *before* the instruction issues so that the RB can be looked up. However, in modern pipelines, an instruction's input values are available *after* issue (in register read or via bypass). Therefore, the redundant instruction looks up the RB with just the fetch PC in the pipeline front end well before issue stage. The redundant instruction stores the RB entry—i.e. input values and output value stored in the RB—in the issue queue. In modern pipelines, instruction input values are available after issue to avoid making the issue queue large and consequently slow. By holding the RB entry values in its issue queue, DIE-IRB incurs these considerable disadvantages.

Although the RB entry is available, the redundant instruction *still* needs its inputs to decide if there is a match with the RB entry. Only on a match can the output in the RB entry be compared with the main instruction for fault detection. But the main problem is that the redundant instruction does not have its inputs before it issues. Instead, DIE-IRB uses the main instruction's inputs to match against the RB entry, and upon a match the redundant instruction is not executed and the RB entry output is compared with the main instruction's output. Using the main instruction's inputs instead of the redundant instruction's inputs is correct because a fault in the main instruction's inputs would cause the RB entry not to match, the redundant instruction to execute, and the comparison between the main and redundant instructions' outputs to fail; a fault in the main instruction's output would cause the comparison with the RB entry to fail.

In a brief description, the DIE-IRB paper claims that values in the issue queue can be avoided by using a "non data-capture scheduler" where the register read occurs *before* instruction selection. However, such a design has many problems: First, wake-up and select are now in different cycles preventing back-to-back issue of dependent instructions which is crucial to performance. Second, wake-up produces a variable number of ready instructions all of which need to go to the register file whose ports may not match the number of the instructions. Third, in select some instructions may not be selected due to exceeding the number of available functional units in which case the excess has to be held back in a buffer *other than the issue queue* to be selected in the future.

The only simplification in DIE-IRB is that because load values are passed from the main thread to the redundant thread, there is no reuse of load values; though there is reuse of load address calculation. Consequently, DIE-IRB's IR does not need to invalidate the RB on stores.

3.2 Implicit Redundancy Through Reuse

We take IR to the extreme of performance-coverage tradeoff in implicit redundancy through reuse (IRTR) by making the following observation: DIE-IRB' s comparison of the main and redundant executions to update the reuse buffer, upon reuse buffer misses or failed reuse tests, is unnecessary for less-than-perfect coverage. Not performing the comparison leads to loss of coverage in some cases. Because redundant execution is needed only for this comparison, the comparison becoming unnecessary makes the redundant execution unnecessary as well. No redundant execution implies no perfor-

```
(i) add r3, r2, r1     // r1=5, r2=7, RB miss => not checked,
      . . .            r3=r1+r2=12, insert into RB
      . . .
(ii) add r3, r2, r1    // r1=5, r2=7, RB hit & reuse test succeeds
      . . .            and output (r3 = 12) matches the RB entry
      . . .            => no fault
      . . .
(iii) add r3, r2, r1   // r1=5, r2=7, RB hit & reuse test succeeds
      . . .            but fault in the output (r3 = 14) => does not
      . . .            match the RB entry => fault detected
      . . .
(iv) add r3, r2, r1    // r1=5, r2=15, RB hit & reuse test fails =>
      . . .            not checked, r3=r1+r2=20, insert into RB
      . . .
(v) sub r5, r6, r7     // maps to the same RB entry as instruction
      . . .            (iv) which is evicted before being checked
      . . .
(vi) add r3, r2, r1    // r1=5, r2=15, RB miss => not checked,
      . . .            r3=r1+r2=20, insert into RB
```

FIGURE 3: Example of IRTR

mance loss.

In the case of reuse buffer misses or failed reuse tests, IRTR updates the reuse buffer using the main thread without any comparison. The lack of comparison leads to loss of coverage in some but not all cases, as illustrated by Figure 3. Assume that the main thread inserts the first instance of an instruction into the RB without any comparison (e.g., instruction *(i)* in Figure 3). When the second instance probes the RB there are three possibilities: (1) There is an RB hit but outputs differ. If there was a fault in one of the output values— either in the reuse entry or the second instance of the main thread—then this fault would be detected because the reuse test would succeed but the outputs would not match (e.g., instruction *(ii)* shows how no-fault case is covered and instruction *(iii)* shows how faulty case is covered). Thus, in this possibility, there is no loss of coverage. Specifically, a fault even in the first instance is covered though the first instance was put in the RB without checking. (2) There is an RB hit but the reuse test fails because the inputs differ. This case occurs if there is a fault in the input values of either the reuse buffer entry (because it was not checked) or the second instance. This case also occurs even if there is no fault but the second instance uses different input values. This possibility leads to loss of coverage. In Figure 3, if we assume that instruction *(iv)*'s input r2 = 15 is faulty and should have been r2 = 7, then this fault would go undetected. Also, if instruction *(iv)*'s input r2 = 15 is not faulty, then we still lose coverage in that a fault in instruction *(iv)*'s output would go undetected if there is no reuse later. (3) There is an RB miss (i.e., the first instance was evicted from the RB before the second instance probes the RB) (e.g., instruction *(vi)*). This possibility leads to loss of coverage because a fault in the output of the first instance would go undetected.

Thus, reuse detects *implicit* redundancy *within* the main thread and this implicit redundancy allows some coverage without any explicit redundancy and therefore, without worsening performance. Recall that IRTR accepts less-than-perfect coverage to trade-off coverage for performance. In contrast, DIE-IRB does not explore this trade-off and incurs performance loss for perfect coverage which may not be needed.

IRTR also uses the value-based IR implementation but has none of the complications of DIE-IRB. In IRTR, the instructions check the RB *after* execution by which time the input and output values of the instruction are available. Because the RB is checked after execution, there is no need to place values in the issue queue or modify the issue queue in any way. Note that DIE-IRB *always* places the redundant instruction in the issue queue and removes the instruction if the reuse test succeeds. However, in the time duration in which the main instruction waits for its wake-up in the issue queue, the redundant instruction increases the occupancy of the issue queue even though it is removed later. This extra occupancy impacts DIE-IRB's performance especially because DIE-IRB does not use the BOQ and inserts both correctly-speculated and misspeculated redundant instructions. IRTR avoids this extra occupancy by not even placing the redundant instruction in the issue queue if the reuse test succeeds.

In addition, DIE-IRB has to access the RB between fetch and issue whereas IRTR can access the RB any time between execute and commit. Because fetch to issue is much shorter in duration than execute to commit, our RB can be much larger than DIE-IRB's RB to achieve more reuse. Therefore, IRTR uses a 2-way set associative RB, while DIE-IRB used a direct-mapped IRB.

Finally, similar to DIE-IRB, IRTR does not allow the reuse of loads for simplicity.

4 Putting PER and IRTR Together

PER and IRTR are orthogonal and can be implemented separately or together. Though PER and IRTR target complementary cases of low- and high-ILP phases, the techniques are not entirely complementary. For instance, an RB hit in a low-ILP phase is an overlap between PER and IRTR. However, there are many cases where there is no overlap and combining the techniques is beneficial. Because the overlap lessens the benefit of applying PER and IRTR together, we present two optimizations in which PER and IRTR enhance each other when combined. These optimizations mitigate the effect of the overlap. PER improves IRTR's reuse buffer hit rate by avoiding pollution of the buffer with the instructions that are covered by PER; and IRTR improves the utilization of PER's implementation resources by avoiding pollution of the resources with the instructions that are covered by IRTR.

When applied together, instructions that are checked by IRTR need not be executed redundantly in PER. To that end, we slightly modify the RDQ. Upon an RB hit, the main thread instruction marks its RDQ entry. Depending upon whether the mode is REM or SEM the entry has different purposes. In REM, the marked RDQ entry causes the redundant instruction to be discarded and not inserted in the issue queue. The entry updates the redundant thread state as usual.

In SEM, there is no redundant thread but there is an opportunity for IRTR to enhance PER. The marked RDQ entry updates the redundant thread processor state slightly differently than usual (Section 2.2.3). In the absence of IRTR, there is an incentive to keeping instruction values in the RDQ for a long time in anticipation that execution will switch from SEM to REM and the values can be compared to their redundant counterparts. Accordingly, SEM emptied the RDQ to update the redundant thread state only when the RDQ was full. However, in the presence of IRTR, the incentive is gone. A marked RDQ entry implies that the instruction has already been checked by IRTR and there is no need to keep the value in the RDQ. Instead, updating the redundant thread state without waiting for the RDQ to fill has the benefit that newer instruc-

Table 1: System parameters

Architectural parameters	
Instruction issue	6, out-of-order
Branch prediction	8k hybrid of bimodal and gshare, 16-entry RAS, 4-way 1K BTB (15-cycle misprediction penalty)
L1 I- and D-cache	64KB 4-way, 2-cycle
L2 unified cache	1M 8way 12-cycle
RUU/LSQ	128/64 entries
Functional units	6 Integer ALUs, 2 Integer mul/div units, 4 FP ALUs, 2 FP mul/div units
Memory ports	2
Off-chip memory latency	Infinite Capacity, 300 cycles
SRT parameters	
BOQ/LVQ/StB	96/128/20 entries
Slack	256 instructions
PER parameters	
RDQ	300 entries
IRTR parameters	
RB	1024 entries, 2-way

Table 2: Benchmarks

Bench-mark	Commit IPC	Execute IPC	misses per 1000 insts	RB hit %	Redun-dancy %
sixtrack	3.20	3.29	0.51	17	10
mesa	2.60	3.57	0.14	36	6
vortex	2.19	2.52	0.27	51	35
apsi	1.58	1.84	1.95	10	62
crafty	1.57	3.18	0.40	33	42
bzip	1.42	2.74	1.00	40	54
eon	1.40	3.13	0.00	39	53
fma3d	1.14	1.24	10.59	26	65
gcc	1.12	1.14	3.71	32	68
facerec	1.01	1.51	5.14	26	70
applu	0.91	0.91	22.95	10	74
mgrid	0.79	0.80	8.98	7	72
twolf	0.59	1.26	3.58	26	75
swim	0.49	0.49	21.70	2	84
vpr	0.47	0.73	6.29	41	82
art	0.40	0.58	84.44	26	92
equake	0.35	0.36	26.63	33	87
lucas	0.24	0.24	17.52	55	91

tions can come into the RDQ earlier. Then these newer instructions stay in the RDQ longer and have a better chance of being covered by PER.

PER can also enhance IRTR. The idea is to increase the chance of covering those instructions by IRTR that are not covered by PER. Therefore, we have to increase the chance of RB hit for such instructions. One way to improve the hit rate is to avoid polluting the RB with instructions which are likely to be covered by PER. To determine this likelihood, we make the key observation that low-IPC phases in a program tend to be low-IPC when executed again. The same observation is also correct for high-IPC phases. A similar observation is also valid for loads; certain loads tend to be problematic and cause L2 misses. Therefore, when using PER, the probability that a certain phase of a program will be executed in SEM (or REM) mode is high given that it was previously executed in SEM (or REM) mode, respectively. Accordingly, we update the RB only in SEM and not in REM to avoid polluting the RB.

5 Methodology

We modify the Simplescalar 3.2b out-of-order simulator [2] to simulate SRT, [25] which squashes on L2 misses to reduce soft-error rate (we denote it as SL2), PER, and IRTR. Table 1 shows the simulation parameters used throughout our simulations unless otherwise specified.

Estimating soft-error rate (SER) would involve device and circuit details. However, [8] provides an abstract, architectural metric for SER called the architectural vulnerability factor (AVF). AVF of a processor structure is the probability that a fault in that structure will result in a visible error in the final output of a program [8]. Because SER is directly proportional to AVF (SER = AVF * raw soft-error rate), and because our techniques affect only the AVF and not the raw soft-error rate, we show the AVF in our results. We use the methodology described in [8] to compute the AVF of the individual pipeline structures such as the issue queue, ALUs, and the register file. For each structure, we trace the number of bits required for architecturally correct execution (ACE) [8] that go through that structure during execution time. Then we divide that number by the total number of bits that can be held by that structure during that time. All the bits of instructions that are executed redundantly, instructions that are checked using IRTR, dead instructions, and the registers that have redundant copies during redundant execution are considered un-ACE bits because a fault in the bits either can be detected or will not affect the program output.

Extending [8], which computes each individual structure's AVF in isolation, we combine the individual AVFs to produce the overall AVF for the entire processor. Assuming the reasonable assumption that logic density is similar among pipeline structures, the probability of a soft error is directly proportional to the area of the structure. Then, the overall AVF is the weighted sum of each individual structure's AVF weighted by the fraction of the structure's area. We take area estimates from Hot Spot [17] for the structures shown in Table 1 and compute the overall processor AVF.

We use a representative subset of SPEC2000 suite. We skip the number of instructions specified by SimPoint [15] to each benchmark's early single simulation point. We warm-up the caches during the last 1 billion instructions of the skipped instructions if the benchmark is skipped by more than 1 billion instructions. Otherwise, we warm-up the caches during the skipping period. Then, we run the SimPoint for 100 million

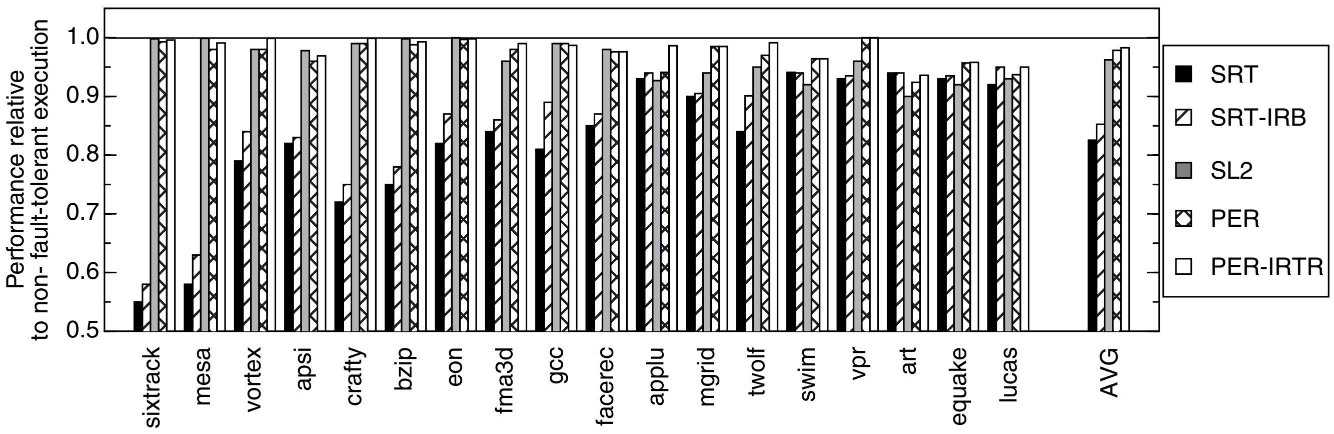

FIGURE 4: Effect of fault tolerance on performance

instructions. Our base case is a non- fault-tolerant execution where the benchmarks execute without redundancy or instruction reuse. Table 2 shows the base-case commit- and execute-IPC (instructions per cycle), the number of L2 misses per 1000 committed instructions, the reuse buffer hit rate (i.e., hit in the RB and reuse test success), and the fraction of committed instructions covered by PER. The benchmarks are ordered in decreasing commit IPC from top to bottom.

6 Experimental Results

We begin our analysis by comparing the performance of the previous work and our techniques. Then we study the trade-off between AVF and performance for the various techniques. Finally, we study the impact of RB size on the performance of IRTR and PER-IRTR. We do not vary the RDQ size because the RDQ size is related to the L2 miss latency and should not be varied independently.

6.1 Performance

In Figure 4, we compare the performance of SRT, SL2, PER, and PER-IRTR (which is PER and IRTR together). We do not compare against DIE-IRB because SRT outperforms DIE-IRB due to SRT's slack fetch and the BOQ optimizations. Instead, we implement SRT-IRB which exploits reuse in SRT to avoid redundant execution whenever there is reuse. Because SRT by itself outperforms DIE-IRB, and SRT-IRB outperforms SRT, our comparison is fair. We do not show IRTR because it does not incur any performance loss and therefore its performance is the same as the base case. The Y axis shows performance relative to the base case (note that the Y axis starts at 0.5). The X axis shows the benchmarks in roughly decreasing commit IPC (shown in Table 2) from left to right, and the average at the far right. The X axis ordering makes it easier to see trends.

SRT degrades performance by 17% on average. Because SRT is a full-redundancy scheme, it significantly degrades high-IPC programs (to the left) due to their high utilization of resources. The low-IPC programs (to the right) incur less degradation. For example, *sixtrack* with commit IPC of 3.20 incurs nearly 45% performance degradation and *lucas* with commit IPC of 0.24 incurs 7% performance degradation. (It may be easier for the reader if the Y axis were percent degradation instead of relative performance. However, in that case, many of the bars for SL2, PER, and PER-IRTR would become invisible.) We see that SRT incurs significant performance degradation to achieve full coverage via full redundancy.

SRT-IRB performs slightly better than SRT, but otherwise closely follows SRT's trends.

Because L2 misses are relatively infrequent and SL2 squashes on L2 misses, SL2 degrades performance only by 3.5% on average. However, there is some degradation for high-miss-rate programs (see Table 2 for miss rates) whereas low-miss-rate programs incur virtually no performance loss. For example, *art, equake, applu*, and *swim* incur 9% performance degradation whereas *sixtrack* and *eon* are virtually unchanged. Though SL2 performs better than SRT, SL2 does not provide the same coverage as SRT.

Our techniques, PER and PER-IRTR, incur about 2% average performance degradation, which is close to that of SL2. In some of the high-miss-rate programs, however, PER and PER-IRTR perform better than SL2 (e.g., *art*, *equake*, and *applu*). Recall that PER and PER-IRTR switch to REM and start executing from both threads immediately after an L2-miss squash, whereas SL2 waits for the miss to return before starting to fetch from the main thread. Allowing the main thread to execute under misses helps these programs.

These results show that (1) our techniques incur minimal performance degradation whereas SRT and SRT-IRB incur significant degradation; (2) our techniques are comparable to SL2 in performance. In the next section, we show the reduction in SER achieved by the techniques. Then, we summarize where the techniques stand in terms of performance-SER trade-off.

6.2 Soft-Error Rate

As mentioned in Section 5, we use AVF as our metric for soft-error rate. We compare the issue queue AVF relative to the base case (whose AVF is normalized to 100%) for all the techniques except SRT and SRT-IRB because their AVF is 0% due to full redundancy. Because SL2 covers only the issue queue, the gains in the issue queue AVF will be overwhelmed by the AVF of the rest of the pipeline, preventing us from seeing SL2's SER improvements. Therefore, we first show the issue queue AVF separately in Figure 5, and then show the overall processor AVF.

SL2 reduces the issue queue AVF by 21% on average. Because SL2 exploits L2 misses, it reduces the AVF more for programs with higher miss rates (see Table 2) (e.g., *art, equake, applu*, and *swim*).

IRTR reduces the issue queue AVF by 29% on average,

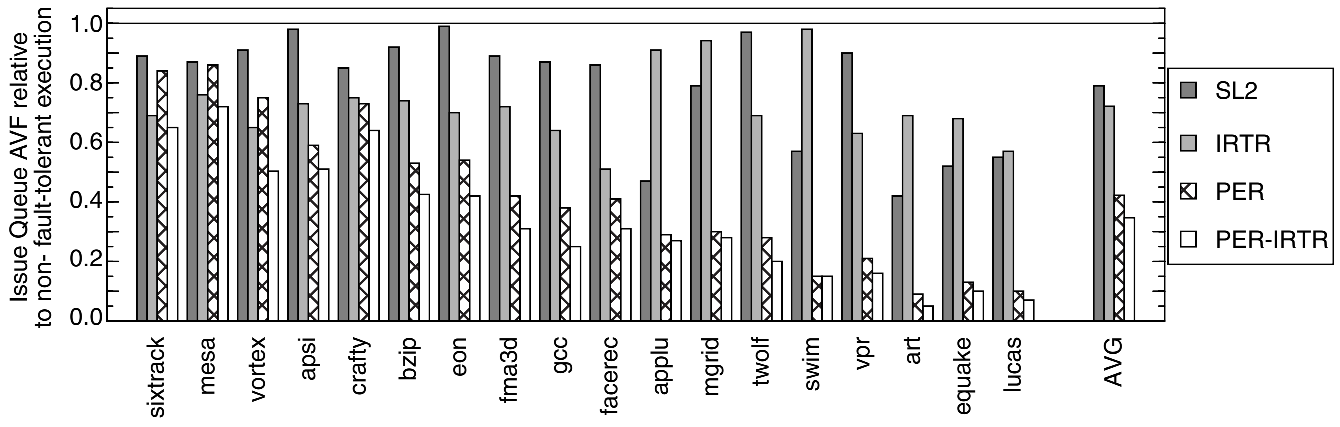

FIGURE 5: Issue queue soft-error rate

which is more than SL2's reduction. The RB-hit-% column in Table 2 shows the reuse buffer hit rate (i.e., hit in the RB and reuse test success) for the programs. The issue queue AVF depends on the time spent by an instruction in the issue queue, and this time is not captured by the RB hit rate. Nevertheless, the AVF reduction correlates well with the RB hit rate. For instance, *vortex* and *lucas* with high RB hit rates achieve large reduction in AVF, and *applu*, *mgrid*, and *swim* with poor RB hit rates achieve little reduction. Because IRTR incurs virtually no degradation, this result illustrates the power of implicit redundancy.

PER reduces the issue queue AVF by 57% on average. The redundancy% column in Table 2 shows the percent of committed instructions covered by PER. The AVF reduction correlates well with this column. As expected, PER works better for low-IPC programs (to the right) (e.g., *vpr*, *art*, *equake*, and *lucas*) and not so well for high-IPC programs (to the left) (e.g., *sixtrack*, *mesa*, and *vortex*).

Finally, PER-IRTR reduces the issue queue AVF by 65% on average. PER-IRTR's AVF reduction = PER's reduction + IRTR's reduction + the two enhancements' reduction - overlap between PER's reduction and IRTR's reduction. Section 4 describes the enhancements and the overlap. PER-IRTR exploits the enhancements described in Section 4 to achieve better coverage than PER or IRTR alone. PER-IRTR achieves better coverage than either PER or IRTR whenever both PER and IRTR achieve reasonably good coverage (e.g., *bzip, fm3d*, *gcc, facerec*, and *lucas*). PER-IRTR does not improve upon the better of PER or IRTR whenever the other (i.e., the worse of PER and IRTR) achieves poor coverage (e.g., *mesa* where PER's coverage is poor, and *swim* where IRTR's coverage is poor). This trend is because when one of PER and IRTR achieves poor coverage then that technique's coverage is subsumed by the other's and our enhancements cannot prevent this subsumption. However, when both PER and IRTR achieve good coverage our enhancements can reduce the overlap between the two.

We compare the overall processor AVF for SL2, IRTR, PER, and PER-IRTR in Figure 6. Because SL2 covers only the issue queue which contributes only 15% to the processor AVF, SL2's processor AVF reduction is about 4% on average. Because our schemes, IRTR, PER, and PER-IRTR, cover the whole pipeline, they achieve better processor AVF reduction of 22%, 44%, and 56%, respectively. Each of IRTR's, PER's and PER-IRTR's coverage of the rest of the pipeline matches that of the issue queue (i.e., if a technique's issue queue coverage is high (low) then its coverage of the rest of the pipeline is high (low)). Consequently, the trends in the issue queue AVF for these techniques in Figure 5 closely match the trends in the overall processor AVF in Figure 6.

Finally, we put these AVF numbers together with the per-

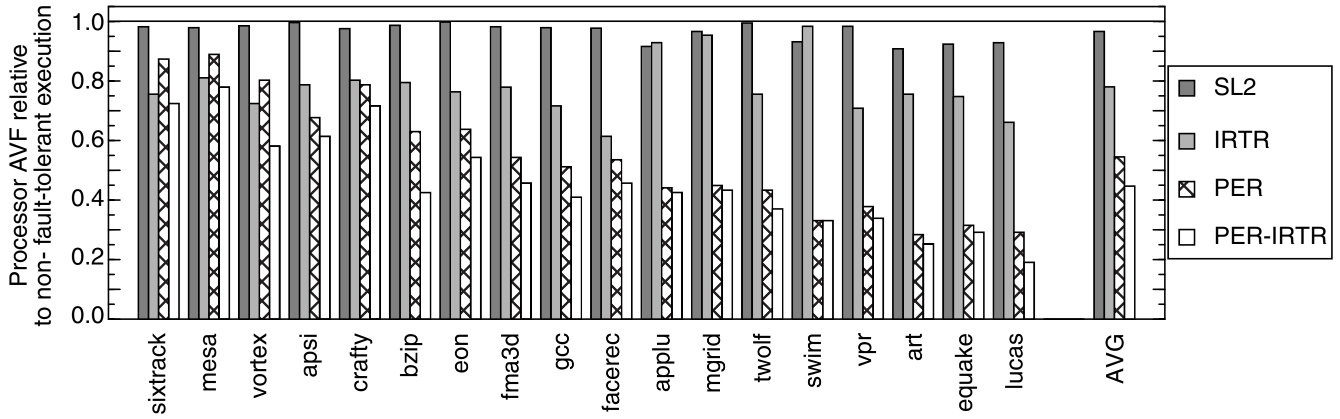

FIGURE 6: Processor soft-error rate

Table 3: SER and performance

	SER reduction %	Performance loss %
SRT	100	17
SRT-IRB	100	15
SL2	4	3.5
IRTR	22	0
PER	44	2
PER-IRTR	56	2

formance degradation numbers from Figure 4 in Table 3. We see that PER and IRTR can reduce the number of crashes caused by soft errors (described in the third paragraph of Section 1) by a factor of two at little performance degradation. The table shows that our techniques achieve better trade-off between performance and soft-error rate than the previous schemes. Thus, our techniques are a better choice than the previous schemes for systems which do not need perfect coverage.

6.3 Effect of RB size

Figure 7 shows the effect of RB size on IRTR and PER-IRTR coverage. We compare the overall processor AVF for RB sizes of 512, 1024, and 2048. We see that for both IRTR and PER-IRTR, the processor AVF improves gradually over these RB sizes, showing that our techniques are robust with respect to the RB size. We also see that increasing the RB size from 512 to 2048 reduces the AVF from 83% to 75% and 50% to 41% for IRTR and PER-IRTR. Thus, our techniques benefit from a larger RB. Although DIE-IRB can also benefit from a larger RB, DIE-IRB has a limit on the RB access time because the RB is accessed between fetch and issue, as described in Section 3.2. Therefore, it is not easy for DIE-IRB to use a large RB. However, IRTR and PER-IRTR do not have this timing limitation.

7 Related Work

Watchdog processors proposes some of the key concepts behind many fault-tolerance schemes [6]. AR-SMT is the first to use SMT to execute two copies of the same program [14]. AR-SMT and its follow-up for chip multiprocessors (CMPs), called Slipstream [21], propose using speculation techniques to allow communication of data values and branch outcomes between the main and redundant threads to accelerate execution. SRT improves on AR-SMT via the two optimizations of slack fetch and checking only stores. DIVA is a fault-tolerant superscalar processor that uses a simple, in-order checker processor to check the execution of the complex out-of-order processor [1]. SRTR extends SRT to provide recovery for SMT [24]. RMT explores design options for fault detection via multithreading, and briefly discusses fault detection using CMPs [7]. CRTR extends RMT to provide fault recovery using CMPs [4]. DIE proposes hardware recovery using superscalar hardware without any SMT support [11]. DIE-IRB exploits instruction reuse to reduce redundant execution [9]. Other work focuses on fault tolerance in functional units [13, 10, 5, 20].

The Compaq NonStop Himalaya [3] and IBM z900 (formerly S/390) [18] provide fault tolerance. The z900 uses the G5 microprocessor which includes replicated, lock-stepped pipelines. The NonStop Himalaya lock-steps off-the-shelf microprocessors and compares the external pins on every cycle. In both systems, when the components disagree, execution is stopped to prevent propagation of faults.

All of the above proposals and products are full-redundancy schemes which incur performance degradation to achieve full coverage. We propose partial redundancy to trade-off coverage for performance in systems where perfect coverage is not needed. [25] also targets such systems but takes a no-redundancy approach to reduce soft-error rate. It squashes the pipeline on L2 misses to avoid particle strikes on the instructions in the issue queue during the long wait for the misses. Our approach achieves better coverage for similar minimal performance degradation. [8] defines AVF to quantify soft-error rate in architectural terms. We use AVF in our evaluations. In addition, we extend DIE-IRB to exploit reuse as implicit redundancy with no additional explicit redundancy.

8 Conclusions

Most current proposals for transient-fault detection use full redundancy to achieve perfect coverage while incurring significant performance degradation. However, most commodity systems do not need or provide perfect coverage. A recent paper explores this leniency to reduce the soft-error rate of the issue queue during L2 misses while incurring minimal performance degradation. Whereas the paper reduces soft-error rate without using any redundancy, we targeted better coverage while incurring similarly-minimal performance degradation by opportunistically using redundancy. We proposed two semi-

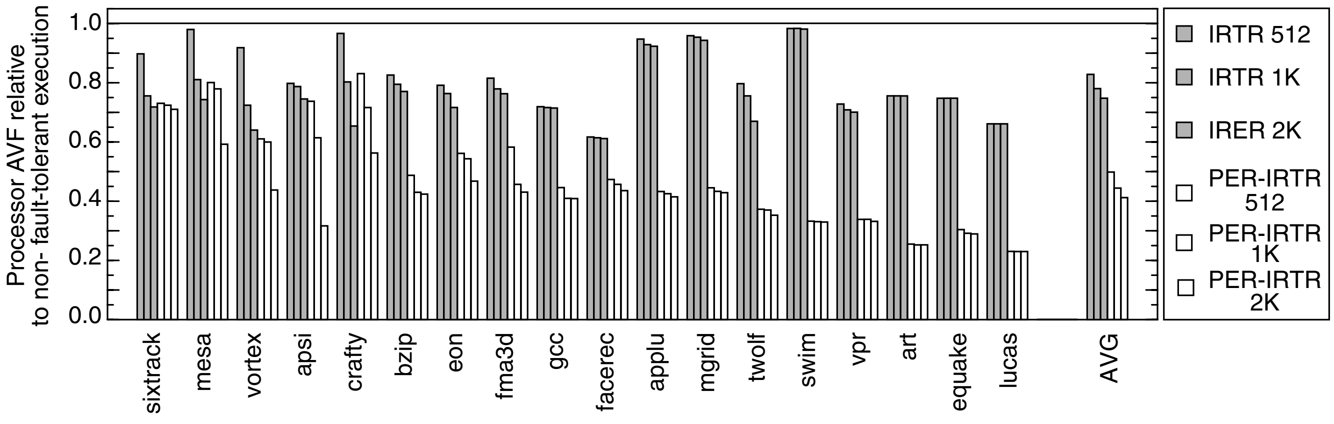

FIGURE 7: Effect of RB size on processor soft-error rate

complementary techniques, called partial explicit redundancy (PER) and implicit redundancy through reuse (IRTR), to explore the trade-off between soft-error rate and performance. PER opportunistically exploits low-ILP phases and L2 misses to introduce explicit redundancy with minimal performance degradation. Because PER covers the entire pipeline and exploits not only L2 misses but all low-ILP phases, PER achieves better coverage. To achieve coverage in high-ILP phases as well, we proposed implicit redundancy through reuse (IRTR). Previous work exploits the phenomenon of instruction reuse to avoid redundant execution while falling back on redundant execution when there is no reuse. IRTR takes reuse to the extreme of performance-coverage trade-off and completely avoids explicit redundancy by exploiting reuse's implicit redundancy within the main thread for fault detection with virtually no performance degradation.

Using simulations with SPEC2000, we showed that average processor soft-error-rate reduction and performance loss for the previous reuse scheme, the previous scheme for reducing the issue queue's soft error rate, and combined PER and IRTR are 100% and 15%, 4% and 3.5%, 56% and 2%, respectively. Put differently, our techniques can reduce the number of crashes caused by soft errors, and the accompanied financial loss, by a factor of two at little performance degradation. Our techniques achieve better trade-off between soft-error rate and performance than the previous schemes.

As transient faults worsen with scaled technologies, techniques like PER and IRTR will become important to achieve good soft-error rates while incurring minimal performance degradation.

Acknowledgments

We thank Ethan Schuchman and the anonymous reviewers for their comments and suggestions. We are also thankful to Purdue University Rosen Center for Advanced Computing for providing valuable computing support.

References

[1] T. M. Austin. DIVA: A reliable substrate for deep-submicron microarchitecture design. In *Proceedings of the 32nd Annual International Symposium on Microarchitecture*, pages 196–207, Nov. 1999.

[2] D. Burger, T. M. Austin, and S. Bennett. Evaluating future microprocessors: The simplescalar tool set. Technical Report CS-TR-1996-1308, University of Wisconsin-Madison, 1996.

[3] Compaq Computer Corporation. *Data integrity for Compaq Non-Stop Himalaya servers*. http://nonstop.compaq.com, 1999.

[4] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, 2003.

[5] J. G. Holm and P. Banerjee. Low cost concurrent error detection in a VLIW architecture using replicated instructions. In *Proceedings of the International Conference on Parallel Processing*, 1992.

[6] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors–A survey. *IEEE Trans. on Computers*, 37(2):160–174, Feb. 1988.

[7] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 99–110, May 2002.

[8] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 29–40, 2003.

[9] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. A complexity-effective approach to ALU bandwidth enhancement for instruction-level temporal redundancy. In *Proceedings of the 31st annual international symposium on Computer architecture*, pages 376–368, 2004.

[10] J. H. Patel and L. Y. Fung. Concurrent error detection on ALU's by recomputing with shifted operands. *IEEE Trans. on Computers*, 31(7):589–595, July 1982.

[11] J. Ray, J. C. Hoe, and B. Falsafi. Dual use of superscalar datapath for transient-fault detection and recovery. In *Proceedings of the 34th annual IEEE/ACM international symposium o n Microarchitecture*, pages 214–224, Dec. 2001.

[12] S. K. Reinhardt and S. S. Mukherjee. Transient-fault detection via simultaneous multithreading. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, pages 25–36, June 2000.

[13] D. A. Reynolds and G. Metze. Fault detection capabilities of alternating logic. *IEEE Trans. on Computers*, 27(12):1093–1098, Dec. 1978.

[14] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *Proceedings of Fault-Tolerant Computing Systems*, 1999.

[15] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 45–57, 2002.

[16] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 389–398. IEEE Computer Society, 2002.

[17] K. Skadron, M. R. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan, and D. Tarjan. Temperature-aware microarchitecture. In *Proceedings of the 30th annual international symposium on Computer architecture*, pages 2–13, 2003.

[18] T. J. Slegel, et. al. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[19] A. Sodani and G. S. Sohi. Dynamic instruction reuse. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 194–205, 1997.

[20] G. S. Sohi, M. Franklin, and K. K. Saluja. A study of time-redundant fault tolerance techniques for high-performance, pipelined computers. In *Digest of papers, 19th International Symposium on Fault-Tolerant Computing*, pages 436–443, 1989.

[21] K. Sundaramoorthy, Z. Purser, and E. Rotenberg. Slipstream processors: Improving both performance and fault-tolerance. In *Proceedings of the Ninth International Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 257–268. Association for Computing Machinery, Nov. 2000.

[22] D. M. Tullsen and J. A. Brown. Handling long-latency loads in a simultaneous multithreading processor. In *Proceedings of the 34th annual ACM/IEEE international symposium on Microarchitecture*, pages 318–327, 2001.

[23] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd annual international symposium on Computer architecture*, pages 191–202, 1996.

[24] T. N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Proceedings of the 29th Annual International Symposium on Computer Architecture*, pages 87–98, May 2002.

[25] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *Proceedings of the 31st annual international symposium on Computer architecture*, page 264, 2004.