

ReStore: Symptom Based Soft Error Detection in Microprocessors

Nicholas J. Wang Sanjay J. Patel
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign

Abstract

Device scaling and large scale integration have led to growing concerns about soft errors in microprocessors. To date, in all but the most demanding applications, implementing parity and ECC for caches and other large, regular SRAM structures have been sufficient to stem the growing soft error tide. This will not be the case for long, and questions remain as to the best way to detect and recover from soft errors in the remainder of the processor — in particular, the less structured execution core.

*In this work, we propose the ReStore architecture, which leverages existing performance enhancing checkpointing hardware to recover from soft error events in a low cost fashion. Error detection in the ReStore architecture is novel: **symptoms** that hint at the presence of soft errors trigger restoration of a previous checkpoint.*

Example symptoms include exceptions, control flow mis-speculations, and cache or translation look-aside buffer misses. Compared to conventional soft error detection via full replication, the ReStore framework incurs little overhead, but sacrifices some amount of error coverage. These attributes make it an ideal means to provide very cost effective error coverage for processor applications that can tolerate a non-zero, but small, soft error failure rate. Our evaluation of an example ReStore implementation exhibits a 2x increase in MTBF (mean time between failures) over a standard pipeline with minimal hardware and performance overheads. The MTBF increases by 20x if ReStore is coupled with protection for certain particularly vulnerable pipeline structures.

1. Introduction

Among the various issues facing the scaling of implementation technologies into the deep submicron regime, cost effective design of reliable processors in the presence of transient errors remains a challenge. Transient errors can arise from multiple sources: external sources such as high-energy particles that cause current pulses in digital circuits, as well as internal sources that include coupling, leakage, power supply noise, and temporal circuit variations.

While transient errors (also known as soft errors) have always to some extent plagued semiconductor-based digital systems, the scaling of devices, operating voltages, and design margins for purposes of performance and functionality raises concerns about the increased susceptibility of future-generation systems to such transient effects. Historically,

transient errors were of concern for those designing high-availability systems or systems used in electronics-hostile environments such as outer space. Because of the confluence of device and voltage scaling, and the increasing complexity of digital systems, the problem of transient errors is forecast to be a problem for all future digital systems. From high-energy neutrons alone, experts estimate that Failures in Time (FITs) for a chip will scale at a minimum with the number of devices (i.e., with Moore's Law).

Known, effective techniques exist for protecting large on-chip SRAM structures such as caches (e.g., parity and ECC), but the question of protecting the unstructured control logic that exists within a modern processor pipeline remains open. The amount of chip area devoted to such general logic is increasing with chip complexity, and therefore the effects of transient errors through combinational logic networks and pipeline latches is of particular concern. Few, and mostly costly, techniques exist for protecting the instruction processing pipeline of a modern high-performance processor.

As a case example, when IBM designed the S/390 Server G5 [27], a high tolerance to soft errors was deemed necessary. Parity and ECC provided error coverage for much of the design, but full duplication was used for the execution pipeline. This decision was made to ensure minimal impact to performance while providing maximal error coverage. This costly approach may be acceptable for a flagship product intended for the high availability market segment. However, it may not be suitable for mainstream use where cost is of greater concern and where applications demand reliable operation but are generally not mission critical. Instead, a lower cost (in terms of die space, design complexity, and power consumption) solution that provides *sufficient* soft error coverage might provide an attractive alternative.

We propose the ReStore processor architecture, which uses a combination of short term on-chip architectural checkpoints with novel symptom-based error detection to provide high degrees of transient error protection at low cost. In our previous work [29], we observed that a very large fraction of simulated transient errors injected into the latches and RAM cells of a modern processor pipeline (using a Verilog processor model) were logically masked before they could adversely affect the executing application. Those injected faults that did corrupt the application often did so *quickly* and *noisily*. That is, such faults cause events that are atypical of normal or steady state operation in addition to generating a data corruption; these atypical events serve as warnings that something

is amiss. We call such events transient error *symptoms*. Examples of such symptoms include exceptions (e.g. memory protection violation, etc) and incorrect control flow (e.g. following the wrong path of a conditional branch). The tendency for these symptoms to occur quickly after a transient, coupled with a checkpointing implementation in hardware to restore clean architectural state, enables a cost effective soft error detection and recovery solution.

In the ReStore Architecture, checkpoints are created by the hardware every n instructions, where n might range from 10 to 1000 instructions, depending on design tradeoffs. As instructions are executed by the processor pipeline, detection logic monitors the application for symptomatic behavior. Detection of a symptom triggers rollback to a checkpoint. If the checkpoint contains uncorrupted data, then the soft error (if any) was successfully detected and recovered. Because symptoms are early alarms, false positives can occur; a symptom detector might falsely trigger a rollback in the absence of an error. If this is the case, the rollback was unnecessary and the processor incurs a slight performance loss.

Errors are detected through time redundancy in the ReStore Architecture. However, instead of providing complete time redundancy, the symptom detectors trigger in situations that are likely to occur in the presence of an error. Thus, the cost of redundancy is only paid when a soft error is likely to be present – in essence, redundancy on demand. While additional hardware is required to implement such a framework, we believe that many of the required components will exist in future microprocessors for the purpose of high performance execution (i.e., speculation). Current microprocessors already maintain checkpoints across 10’s of instructions for purposes of speculation recovery.

We make several contributions in this paper. We describe and evaluate the ReStore processor architecture. We describe the checkpointing process and argue that it is an extension of the checkpointing performed by high-performance processors today. We perform a series of statistical fault injection studies to evaluate the coverage provided by symptom-based checkpoint restoration. In particular, we examine two classes of symptom detectors that we empirically observe to be the most common harbingers of data corruption due to soft errors: exceptions and errant control flow. We find that with these two symptom detectors, we are able to reduce the incidence of soft error-related silent data corruption by 2x over a standard, contemporary high-performance pipeline, with marginal effect on performance and a slight increase in implementation complexity. If the ReStore pipeline is augmented with parity protection on certain pipeline structures, the mean time between data corruptions increases to 7x over a contemporary baseline. In addition, to support implementation, we propose the use of event logs to compare and contrast the events that occur during the original and redundant executions. This mechanism enables error logging and dynamic control of the error coverage/performance tradeoff.

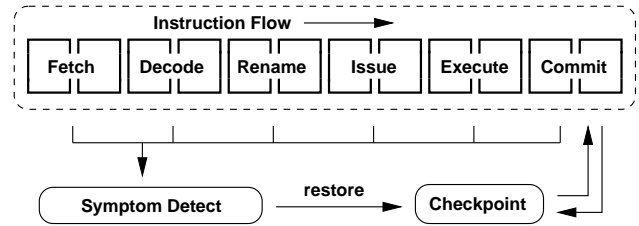


Figure 1. ReStore architecture diagram

2. Processor Architecture

In this section, we present a high level overview of the ReStore processor architecture. ReStore is an augmentation of a modern high performance processor with components for soft error symptom detection, checkpoint storage, and recovery. We argue in this section that the additional mechanisms required for ReStore, in particular the checkpointing mechanism, are straightforward extensions to hardware currently added to today’s processors.

2.1. Microarchitectural overview

Figure 1 presents a conceptual block diagram of a generic processor pipeline, augmented with the ReStore mechanisms. Key features of this architecture include a checkpoint store that stores a snapshot of architectural state from 10–1000 instructions in the past. This state is the “safe” state, and is restored when one of the symptom detectors signals that an error might have occurred. These detectors are distributed throughout the execution pipeline, depending on the symptom they are based on; in the figure, they are conceptually located in the Symptom Detect block.

Logically, a checkpoint is a snapshot of the architectural register file and memory image at an instance in time. At the hardware level, the register file can be checkpointed by saving off its contents, either through explicit copying or by saving the current mapping between architectural registers and physical registers. Various techniques for doing so exist, some of which are used in today’s microprocessors [5]. Depending upon the implementation, the latency of register checkpointing can be minimal (e.g., 1 cycle) with the appropriate hardware resources, with negligible impact on the performance of the processor.

Creating checkpoints for memory state is more involved. The basic idea is to buffer the memory updates executed between each checkpoint. This store buffer can be implemented as a dedicated gated store buffer in the execution core [14], or the L1 cache can be extended to provide this functionality [28]. Clearly, in order to preserve correct execution semantics, load instructions will need to check the store buffer for recent store activity prior to checking the caches.

New checkpoints are taken periodically to maintain a recent recovery point. Furthermore, checkpoints must be taken on external synchronization events. For example, interrupts and synchronizing memory instructions force checkpoints to be taken, in order to maintain correctness.

In order for the ReStore approach to provide resilience to errors, the checkpointed state of the processor needs to be

hardened against data corruption. Because the update frequency of this state is relatively low, it can be protected with ECC for recoverability, or parity for detectability. Furthermore, propagation of corrupt data into the checkpoint store from neighboring components can be minimized through careful design.

The specific design and implementation of the checkpointing mechanisms are beyond the scope of this paper. In this paper, we assume that architects have designed low latency and low overhead checkpointing mechanisms to activate performance enabling speculation. We then leverage their mechanisms for soft error detection and recovery.

2.2. Speculation and checkpointing background

To further our argument that low-level architectural checkpointing is well understood, we examine how architectural checkpointing is used in today’s processors. We also examine the concept of control flow speculation, as it necessitates the use of checkpointing and is a prelude to one of our symptom-based error detection schemes that we discuss in the next section.

Today, almost all processors are pipelined to increase instruction throughput, and most pipelines execute instructions in an out-of-order fashion to extract instruction-level parallelism from the instruction stream. In order to make efficient use of such deeply-pipelined, out-of-order machines, *speculation* is required. And in order to support speculation, low-level architectural checkpointing is required.

Deep pipelines force the instruction fetch unit at the front of the pipeline to speculate and fetch instructions well before it is known whether they are needed or not. In particular, as branch instructions are fetched, their outcome (e.g., taken or not taken) is predicted by the fetcher. Only when the branch executes at the tail end of the pipeline is it known whether the speculation was correct or incorrect. If incorrect, the pipeline is flushed and the fetch unit resumes fetching instructions at the correct memory address.

Because instructions can execute out-of-order, certain instructions after a speculated branch (i.e., those on the wrong execution path) might have executed and modified architectural state, and need to be undone, in some sense. To perform recovery, register file values are restored by recovering architectural to physical register mappings, and memory state is restored by flushing younger store instructions from the store buffer [14]. In a modern processor such as the Pentium 4 [5], the pipeline can flush back to an arbitrary instruction in the pipeline, which amounts to about 10-to-100 instructions of checkpointed state.

The historical trend of processor design has been to speculate over longer periods of time partly due to increased pipeline depth and out-of-order scheduling windows, requiring longer checkpointing intervals. These longer intervals are important to the ReStore architecture, since they enable detection and recovery of soft errors with longer soft error to symptom propagation latencies. Proposed future architectures clearly reflect the trend of more speculation and longer checkpointing intervals [1]. Because these checkpoints are

generated and managed by the processor hardware, they can be created and restored with low latency.

3. Symptom-based Detection

The novelty of the ReStore architecture is the use of symptoms to perform error detection. Error detection has historically been the costliest, most problematic part of error-tolerant processor design. A symptom-based approach relaxes the constraints on the error detection mechanism, allowing it to be approximate.

In this section, we empirically derive two candidate symptoms by performing fault injection studies, and discuss how their detection might be accomplished in a processor pipeline. We also discuss the use of event logs to aid in the implementation of a ReStore processor. Finally, we generalize the notion of symptom-based detection.

3.1. Examining the wake of a soft error

When a soft error occurs in a processor pipeline, one of three possible outcomes will arise:

1. The soft error will be masked, or overwritten before it can cause incorrect behavior. Since the error does not result in a persistent data corruption or program crash, we are not concerned with symptoms in this category.
2. The soft error induces a failure by causing a deadlock or livelock condition with the processor hardware. These conditions are often easily detected by watchdog timers or other liveness checks, and can often be recovered by flushing the pipeline.
3. The soft error propagates into live software visible state (registers or memory state) and is not overwritten. Such errors are called persistent data corruptions, and soft errors that fall into this category are the focus of this work.

As software runs on a processor, it continuously operates upon architectural state — reading values from registers and memory, performing operations on the values, and writing the results back. If the software operates on corrupt values, not only could the data result be incorrect, but the error could result in “side-effects”. For example, exceptions (memory access faults or arithmetic overflow) and incorrect control flow (following the wrong path after a conditional branch) can be caused by corrupt data values feeding into a pointer, arithmetic value, or branch instruction. These events are examples of invalid program behavior — events that should not occur in normal program execution.

In addition, more subtle events like cache and TLB misses can also be caused by soft errors. These events are valid and occur during normal processor operation, since memory caches and translation look-aside buffers are designed to only buffer a subset of all possible entries. These events are infrequent in steady state execution and can indicate the presence of a soft error.

The essential question is: what happens to an executing application shortly after a soft error has corrupted its architectural state? Are there detectable events that can be used

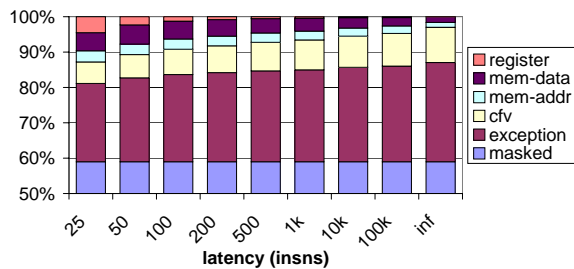


Figure 2. Virtual machine fault injection.

for symptom-based detection? We investigate this question via fault injection campaigns on an instruction set simulator capable of running Alpha ISA binaries. We elected to perform this initial study on a virtual machine to remove any hardware implementation specific effects. In other words, we abstract away the processor implementation by assuming that a soft error has already corrupted architectural state, focusing instead on the propagation of the incorrect architectural state into a soft error symptom. For the fault model in this experiment, we chose a single bit flip in the result of a randomly chosen instruction to emulate the effects of a failure-inducing transient fault. From our previous experience, this fault model is a crude but reasonable approximation of faults injected into a lower-level processor model. The SPEC2000 integer benchmark suite was used as the workload, and each benchmark was subjected to approximately 1000 trials, yielding an overall error margin of less than 0.9% at a 95% confidence level. A summary of the results is provided in Figure 2.

In the figure, each bar represents the number of instructions elapsed between the fault injection to first error symptom (if any). The vertical axis represents all trials, divided into various categories, with the lower categories taking precedence (for example, a trial that fits in both the *exception* and *cfv* categories is placed in the *exception* category). The different categories are described in Table 1. For example, the *masked* category contains data corruptions that did not ultimately affect the executing application.

Category	Observed Error Symptom
masked	The injected fault was masked (did not cause failure)
exception	Instruction Set Architecture defined exception
cfv	Control flow violation - incorrect instruction executed
mem-addr	Address of a memory operation was affected
mem-data	A store instruction wrote incorrect data to memory
register	Only registers were corrupted

Table 1. Figure 2 category descriptions

Across the benchmarks, the average injected fault is masked approximately 59% of the time. This high rate of software level logical masking is due to corruption of results that are dead, transitively dead (data that is only consumed by dead or other transitively dead instructions), or a part of prefetch instructions [20]. We are primarily concerned with the remainder of the trials — the 41% of the injected faults that do cause persistent data corruption.

About 24% of all fault injections (more than half of all failing trials) result in an ISA defined exception within 100

instructions. Most of these are memory access faults (such as attempting to access a virtual page for which the application does not have appropriate permissions), while a small portion consist of arithmetic overflow or memory alignment exceptions. Another 8% of all fault injections result in a incorrect control flow within the same 100 instruction latency. The vast majority of these stem from selecting the incorrect target in a conditional branch instruction (i.e. choosing the taken target instead of falling through or vice versa).

The remaining trials either propagate corrupt values into memory loads and stores or only affect register file values. As the latency allowed for symptom detection is stretched to the entire program length, the coverage provided by the *exception* and *cfv* categories increases steadily — although the majority of the coverage is still obtained with relatively short latency. The *reg* category all but disappears as the corruptions propagate into the other categories.

Thus, nearly 80% of the failure inducing faults injected in this experiment result in an exception or control flow violation within 100 instructions of the fault injection. If one were able to detect these error symptoms and roll back to a checkpoint at least 100 instructions in the past, then 80% of the failure inducing faults would be covered. Coupled with the intrinsic 59% masking level provided by the application, only 9% of injected faults would propagate to a failure.

The level of coverage provided by detecting exceptions and control flow violations might at first seem rather high. We reason that many of the instructions in a typical program are devoted to calculating memory address values and determining control flow [15], so corrupting the result of a random instruction can often result in memory access exceptions or incorrect control flow.

Furthermore, another contributing factor to the frequent occurrence of memory exceptions is that for many workloads, the virtual address space is significantly larger than the memory footprint of the application. This increases the likelihood that a random corruption in a pointer value will result in a pointer to an invalid or unmapped virtual page.

Thus, an architecture with a smaller virtual address space (or correspondingly, a workload with a larger memory footprint) may not exhibit such a large occurrence of exceptions from soft errors. To investigate this effect, we conducted another fault injection campaign where only the bottom 32-bits of each 64-bit result were targeted. The data showed that the *exception* category did indeed become smaller, losing about 25% of its size. The slack was consumed by the *cfv* and *mem-addr* categories, with the *cfv* category picking up the majority. This indicates that control flow based symptoms would play a larger role on a machine with a smaller virtual address space or on programs with larger memory footprints.

3.2. Implementation

3.2.1. Detecting exceptions. Detecting memory access and alignment exceptions as soft error symptoms is straightforward, since all processors must already detect and report any ISA-defined exceptions. In the ReStore architecture, instead of stopping execution and reporting an exception immedi-

ately upon its discovery, the processor first rolls back to a previous checkpoint and re-executes the instructions leading up to the discovered exception. If the exception fails to appear again, then a soft error has been detected and possibly recovered from. Otherwise, either the exception is genuine or a data corruption occurred prior to the checkpoint.

Since there is a performance penalty associated with rolling back to a checkpoint and re-executing a set of instructions, it is important to minimize the number of rollbacks that occur in the absence of a real error. We call such checkpoint restoration causing symptoms *false positive symptoms*, since they falsely identify the presence of a soft error. However, because exceptions are fairly rare during error-free operation, and, perhaps more importantly, program execution cannot continue without first handling any exceptions that arise, there is little reason to not initiate a checkpoint recovery on memory access, alignment or any other exceptions.

3.2.2. Detecting incorrect control flow. Designing a low cost mechanism for detecting invalid control flow is more difficult. Many techniques to monitor control flow have been proposed in the past [16], but they all (to the best of our knowledge) focus on detecting *illegal* control flow. Illegal control flow occurs when the sequence of executed instructions is impossible, owing to a control flow transition either in the absence of a branch instruction or in the presence of a branch instruction but to an illegal target.

Here, not only do we wish to detect illegal control flow, but we also wish to detect legal, but incorrect, control flow. An example of legal but incorrect control flow occurs when a conditional branch chooses the alternate target (taken instead of not taken or vice versa). While such a control flow transition is legal (it is one of the two acceptable choices specified by the compiler), the branch condition is evaluated incorrectly. Control flow watchdogs proposed previously do not detect such invalid control flow.

Instead of designing another hardware or software mechanism to identify the entire set of control flow violations, we chose to leverage already existing hardware in high performance microprocessors. The pipeline flushes described in Section 2 have a dramatic impact on performance, since the latency to refill the pipeline after a flush is proportional to the pipeline’s length. Thus, computer architects have designed highly accurate branch predictors to direct the front of the pipeline, in order to minimize the number of pipeline flushes required. These predictors accurately predict the outcome of conditional branches [13, 17], function returns, and other indirect and direct branches. They are typically correct for well over 95% of branch instances.

We propose to use these highly accurate control flow predictors to aid in the detection of soft errors — if a control mis-speculation is discovered deep in the pipeline, perhaps the branch predictor was indeed correct and the “mis-speculation” was instead the result of a soft error. Control mis-speculations could be used as a symptom of soft errors in this manner. Assuming that the processor implements some form of checkpointing, restoration of a prior checkpoint and

re-execution of the intermediate instructions would result in recovery from any soft errors that had occurred since the checkpoint was taken.

However, to minimize the performance overhead, we wish to minimize the number of checkpoint recoveries that occur. Despite a greater than 95% branch predictor accuracy, the large frequency of branch instructions in typical workloads implies that pipeline flushes are still fairly common. Adding a checkpoint recovery on top of each pipeline flush would be unacceptably costly in terms of performance. To this end, we observe that architects have also devised confidence predictors for conditional branch predictions [2, 7, 12], which assign a level of confidence to each conditional branch prediction. A confidence predictor is similar to a branch predictor; it monitors the branch predictor’s past accuracy in predicting a particular branch and determines a “confidence” for that branch. Instead of performing a checkpoint recovery on each control mis-speculation, we only use the control mis-speculation as a soft error symptom if the mis-speculation was labeled as high confidence by the confidence predictor.

Used within the ReStore framework, the different confidence predictors trade off performance (frequency of checkpoint rollbacks) for soft error coverage (percentage of soft errors detected) or vice versa. In this work, we selected the JRS confidence predictor [12], prioritizing performance over coverage. We note that in ReStore, a control flow violation need not be identified immediately. If a violation initially slips detection, it could still induce soft error symptoms (high confidence branch mispredictions or otherwise) in its wake. If this is the case, the different confidence prediction implementations trade off performance for error detection latency.

3.2.3. Detecting memory instruction address differences. In Figure 2, a portion of failing trials that did not fall into the *exception* or *cfv* categories fit into *mem-addr*. This indicates that the injected fault perturbed the address of a memory access instruction, but not enough to cause an exception in the form of a segmentation fault.

Microprocessors use caches to mitigate long latency accesses to main memory. They do this by keeping data predicted to be useful in the near future close to the processor core. Generally speaking, a large fraction of memory accesses are served by the caches. A cache miss to main memory often indicates a shift in the processor’s workload, and can also indicate the presence of a *mem-addr* soft error. Thus, we chose to evaluate the use of cache misses as error symptoms.

With cache miss symptoms, false positive error symptoms can also yield significant performance overheads similar to those seen with the use of misspeculated control flow symptoms. If necessary, a predictor similar to the branch confidence predictor in the previous section could be used to mitigate the effects of false positives. We did not explore this option in this work.

3.2.4. Event logs. To support the implementation of ReStore, we propose event logs that track and record the events leading up to a symptom. These event logs enable detection of soft

errors during re-execution, which in turn allows for dynamic fine tuning of the ReStore framework. Event logs can also provide strong speculation hints (e.g. branch predictions) and can provide input replication to ensure correctness during re-execution (e.g. Load Value Queues [24]).

By tracking and recording events during both the original and redundant executions, soft errors can be detected and logged. As an example, with control flow based symptoms, the event log might store control instruction outcomes. A soft error is detected if any control instructions produce differing results between the original and redundant executions. In the presence of differing results, an implementation of ReStore may elect to re-execute a third time to verify that the soft error occurred during the original execution.

Being able to detect the presence of soft errors enables dynamic fine tuning of the symptom based mechanism. For example, if a processor encounters a high concentration of false positive control flow symptoms, it may elect to temporarily ignore all symptoms in the interest of minimizing the performance robbing impact of checkpoint recovery. Generally speaking, rollback “thresholds” can be adjusted dynamically to trade off error coverage for performance.

3.3. Generalizing symptom-based error detection

In summary, our symptom based detectors trigger on the following two events: “Did an exception occur?” or “Did we mis-speculate a high confidence branch?”. In either case, a prior checkpoint is restored, which enables soft error detection and recovery.

However, the ReStore architecture is a framework into which other symptom-based detection can be easily integrated. Generally speaking, candidate symptoms can be evaluated on the following metrics: (1) the frequency that failure-causing errors generate the symptom, (2) the typical propagation latency from point of error to the symptom, and (3) the frequency of the symptom in the absence of an error. The first and second metrics evaluate the error coverage provided by the candidate symptom. The third metric provides a measure of the performance impact of incorporating the metric into the ReStore framework, i.e., the incidence of false positives. For example, triggering checkpoint restoration on data cache misses might not be a good detection strategy as while data cache misses are favorable on points 1 and 2, they may not be sufficiently rare enough in the absence of transient faults and may cause undue false positives (Point 3). Investigation of other useful symptoms is a topic for later research.

4. Experimental Results

In this section, we present the results of our experimental evaluation of the ReStore architecture. There are two subsections: an evaluation of the ReStore architecture in terms of error coverage and impact on performance, and an assessment of scaling trends.

4.1. Evaluating the ReStore architecture

We now evaluate the ReStore architecture, using deadlocks, exceptions, high-confidence branch mispredictions,

and cache misses as detectors. First, we will examine a baseline processor augmented with the ReStore mechanisms. Second, we will examine the additional benefit beyond ReStore obtained by incrementally protecting portions of the processor from soft error events. Finally, we evaluate ReStore’s impact on processor performance.

4.1.1. ReStore only. Figure 3 presents the error coverage obtained when all the sequential state in the processor core are subject to soft errors, while Figure 4 presents the same data for only flip-flops. The y-axis in both figures covers all trials that result in failure, corresponding to about 8% of all fault injections in both experiments.

The *deadlock* and *exception* categories respectively indicate deadlocks and exceptions that are successfully detected and recovered under ReStore. The *sdc* and *latent* trials represent injected faults that result in persistently corrupted application data as well as escape detection and recovery by ReStore.

The last two categories are *covered* and *chance*. *Covered* indicates faults that induce an error symptom that is caught by the ReStore configuration indicated on the x-axis. *Chance* indicates trials in which the injected fault does not induce a detected error symptom, however, a symptom was nonetheless identified. Even in the absence of a soft error, *chance* trials would result in a checkpoint recovery. Thus, both the *covered* and *chance* categories represent trials in which ReStore recovered from the injected fault.

Note that a trial could be covered by multiple symptom detectors. These trials are classified into only one of the categories in the following order (from highest precedence to lowest): *deadlock*, *exception*, *covered*, and *chance*.

Along the x-axis, various ReStore configurations are enumerated. The suffix denotes the checkpoint interval in instructions while the prefix denotes the symptoms that are enabled. In all configurations, deadlock and exception symptoms are used, since they suffer little to no performance penalty from false positives. In *all*, the remaining three symptoms are all used: high confidence branch misspeculations and load and store misses to memory. In *ldq*, *stb*, and *br*, load misses, store misses, and branch misspeculations are respectively used in isolation.

We chose to distinguish between load and store misses because we found that they have different advantages. Loads that miss to memory often stall the entire machine until the needed data is returned. Thus, using these idle cycles to recover from a prior checkpoint often has minimal impact to overall performance. On the other hand, store misses are typically handled in the back-end of the processor, off the critical path. This means that there are no idle cycles to overlap the checkpoint recovery with. Fortunately, not only are stores fewer in number compared to loads, but we have observed that stores tend to have higher cache hit rates, leading to fewer false positives.

One advantage of scrutinizing store instructions as opposed to loads is that stores provide a means for corrupt register data to be transferred into memory. In particular, an incor-

rect store address often corrupts data in not just one, but two memory locations, increasing the likelihood that the fault’s effects will persist. Thus, focusing on stores with incorrect addresses tends to direct attention towards those faults that are most likely to result in failure.

The *deadlock* and *exception* categories provide the lion’s share of error coverage in all cases. In the *ldq* and *stb* configurations, with 100 instruction checkpoint intervals, the cache miss symptoms each cover about 20% of the remaining errors after *deadlock* and *exception*. This corresponds to 59% of all errors. The *chance* categories provide a nice coverage boost, but hints at the presence of performance robbing false positives, which we will explore later.

The coverage provided by high confidence branch misprediction symptoms is disappointing: only 5% of the remaining errors at a 100-instruction checkpoint interval. There are three main reasons for this. First, the JRS confidence predictor is conservative in identifying high confidence branches. A perfect confidence predictor would yield nearly twice the error coverage. A more accurate branch predictor would also help. Second, about a third of the control flow violations are of the illegal variety. Examples of illegal control flow include conditional branches to incorrect taken targets or branching behavior from a non-branch instruction. A control flow monitoring watchdog [16] would capture these events. Lastly, some of the trials that fall in the *exception* category also exhibited incorrect control flow. Thus, as previously discussed in Section 3, on a machine with a smaller virtual address space or larger application memory footprint, incorrect control flow symptoms would play a larger role.

Using the cache miss and branch misprediction symptoms together yields very good error coverage (40% of the errors not caught by deadlocks and exceptions). However, as we will see later, the resulting performance hit is likely too significant to make this configuration a reasonable option.

When fault injection is isolated to flip-flop state, error coverage increases sharply at first and levels off quickly when checkpoint intervals grow beyond 100 instructions. Note that 100 instructions is approximately the capacity of the pipeline. The knee at 100 instruction checkpoints is mainly due to the more transient nature of data stored in processor flip-flops. Corrupted data in SRAMs tend to linger before being activated and propagating to a symptom. On the other hand, corrupt data in flip-flops tend to flow along with instructions in the pipeline, leading to shorter symptom detection latencies. At the 100 instruction knee, the error coverage provided by ReStore for flip-flops is significantly greater than that for SRAMs.

4.1.2. ReStore + low hanging fruit. In Figure 5, we examine the trade-off between covering portions of processor state from the effects of soft errors and the error coverage obtained. The x-axis represents the percentage of sequential elements in the processor protected against soft errors, while the y-axis represents the corresponding percentage of failures that are recovered. In essence, this is identifying the *low hanging fruit* of the soft error problem. This data is important if design

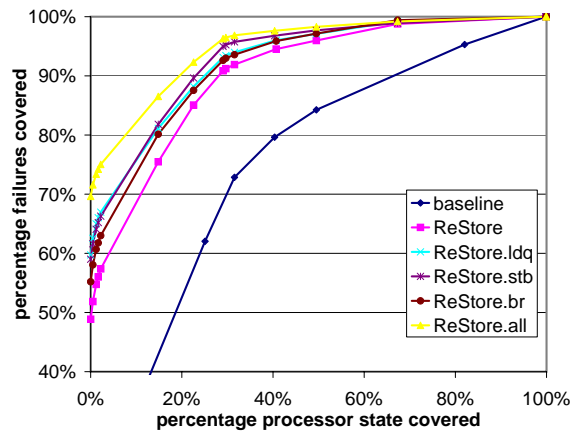


Figure 5. Coverage trade-off between protected processor state and failure coverage

budgets only allow soft error protection for a fraction of a design, perhaps via replication, error correcting codes, radiation hardened circuits, or error trapping latches [18].

To generate this plot, we divided the sequential elements of the processor into disjoint sets of similar logical function. Details about these sets can be found in [29]. Then, each set is analyzed for susceptibility to errors through fault injection, and the sets are sorted by decreasing error susceptibility. The ensuing data points represent the error coverage obtained through incrementally protecting sets of processor state from soft errors.

Six curves are shown, representing six different processor configurations. The first, *baseline*, is the non-ReStore-enhanced processor. At the knee of the curve, protecting 30% of processor states covers 70% of all errors. The remainder of the curves represent various ReStore-enhanced configurations. Note that the ReStore curves all begin with more than 48% error coverage, corresponding to the error coverages observed in Figure 3.

As before, all ReStore configurations include *deadlock* and *exception* symptoms. *ReStore.ldq*, *ReStore.stb*, and *ReStore.br* respectively add load miss, store miss, and branch misspeculation symptoms while *ReStore.all* adds all three. By protecting the most vulnerable 30% of processor state from soft errors, more than 90% of failures are mitigated in each of the ReStore configurations. In particular, with 30% of processor state protected, the *ReStore.stb* configuration covers about 95% of errors (with the store miss symptom covering 45% of the errors uncaught by the *deadlock* and *exception* symptoms). The 95% error coverage translates into a 20x improvement in MTTF (Mean Time to Failure) over the baseline processor.

ReStore.stb by itself contributes 59% error coverage, while covering the most vulnerable 30% of the processor pipeline provides 70% error coverage. Yet, combining the two obtains 95%. The two techniques tend to cover each other’s weaknesses. In particular, some of the most vulnerable portions of the baseline processors are the register alias

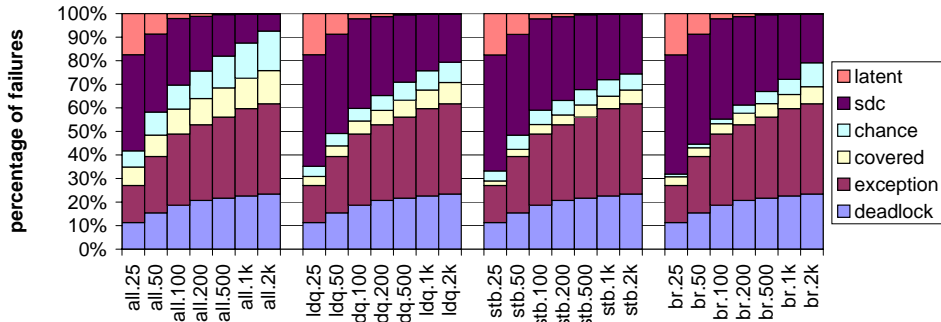


Figure 3. ReStore coverage vs. checkpoint latency for all processor state

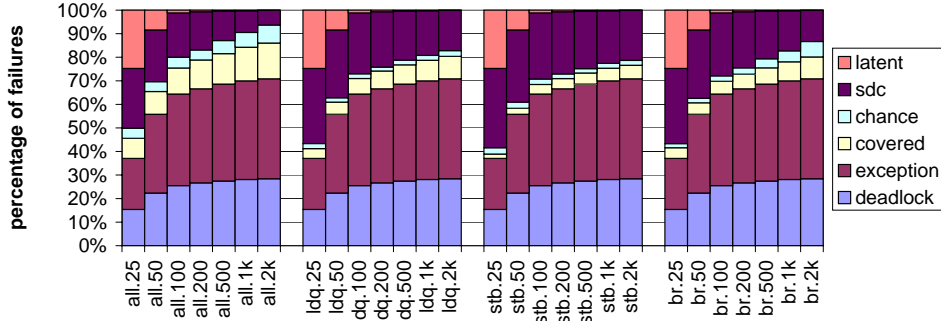


Figure 4. ReStore coverage vs. checkpoint latency for all processor flip-flops

tables, register free lists, and register file, all of which are implemented with SRAMs. On the other hand, ReStore provides better error coverage for flip-flop state, as can be seen by comparing Figures 3 and 4.

4.1.3. Performance impact. In this subsection, we evaluate the performance impact incurred from false positive symptoms. We expect that symptoms originating from watchdog timer saturation and exception events to be infrequent enough to have negligible impact on performance. Thus, we focus on the cost in performance due to checkpoint rollbacks from high confidence branch mispredictions and load and store misses.

We perform this evaluation on a timing model configured to resemble our processor model. In order to support a rollback distance of at least one checkpoint interval, two checkpoints are maintained at all times. Thus, when a rollback is required, the older checkpoint is used to restore architectural state. The average rollback distance is therefore one and a half times the checkpoint interval. During re-execution of a checkpoint interval, a branch outcome event log is used to provide perfect prediction of control flow, eliminating control misspeculations during re-execution.

The results of this experiment are presented in Figure 6. The x-axis plots checkpointing intervals, while the y-axis plots relative performance when compared against a baseline processor without checkpointing. The *imm* prefixed bars indicate the performance impact when rollback occurs immediately upon discovery of a symptom. This has the disadvantage of possibly incurring multiple rollbacks to a single checkpoint if multiple symptoms arise within the same checkpoint interval. As an alternative, we also simulated the performance

impact when checkpoint rollback is delayed until the entire interval is executed. The data from this experiment is represented by the *delayed* prefixed bars.

At shorter checkpoint intervals, the performance hit due to false positive load misses is minimal. This is because most of the penalty of rolling back to a previous checkpoint is overlapped with the miss to memory. As the checkpoint rollback penalty approaches the miss latency (100 cycles), the performance hit increases.

Implementing load miss symptoms also incurs another form of performance overhead. This additional performance overhead comes in the form of an opportunity cost. Run-ahead execution [3, 21] has been proposed as a means to accurately pre-fetch data values and thus increase processor performance. However, run-ahead requires the use of the same idle cycles that ReStore (with load miss symptoms) overlaps much of its checkpoint rollback penalty with. Thus, implementing ReStore may also mean sacrificing the performance gains that would otherwise be obtained by run-ahead. It may be possible to implement a hybrid run-ahead/ReStore scheme to retain most of the benefits of both, and we plan to investigate this in future work.

The performance hit due to store misses is minimal, particularly in the *delayed.stb* configuration (4.5%). Stores occur less often than loads and have lower miss rates. Furthermore, store misses are often clustered together, which proves very beneficial with delayed checkpoint rollback.

The branch misprediction symptom is competitive with the load and store symptoms at lower checkpoint intervals.. Using more accurate branch and confidence [2] predictors

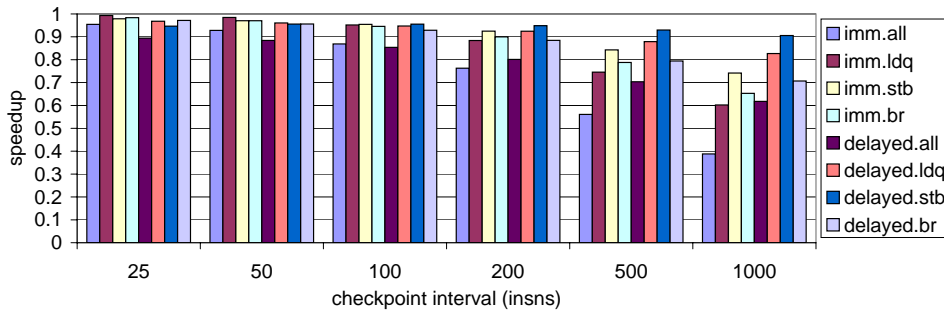


Figure 6. Performance impact of false positive symptoms

would further mitigate the performance loss and also obtain better error coverage.

Blindly using all five error symptoms to trigger checkpoint recovery (*imm.all* and *delayed.all*) yields high performance overheads. Using combinations of the five symptoms might lessen the overheads without sacrificing too much error coverage. We plan to investigate this in future work.

4.2. Scaling trends

In this section, we examine silent data corruption FIT rates as a function of design size for a variety of architectures. FIT stands for Failures in Time, in billions of hours, and is a measure of the reliability of a given design.

To generate the results shown in Figure 7, we assumed a raw FIT of 0.001 per bit [11], which is a widely accepted estimate for per-bit FIT rate in SRAMs. The different processor configurations are the baseline processor without any error detection (*baseline*), with ReStore (*ReStore*, using deadlock, exception, store miss symptoms and 100 instruction checkpoint intervals), with the most vulnerable 30% of processor elements protected from soft errors (*lhf*), and with both techniques (*lhf+ReStore*). The FIT extrapolations are made assuming that the soft error masking rate of the larger designs remains constant as design size is scaled. A reliability goal of 1000 MTBF, or mean time (years) between failures is reflected by the horizontal line at 115 FIT. Any design whose FIT rises above this line fails to meet this goal. Our model consists of approximately 46,000 bits of “interesting” state — state within the pipeline that is particularly problematic to protect. That is, all processor state excluding caches and prediction structures not needed for correctness. This is approximately represented by the first data point in Figure 7. The interesting thing to observe is that the *lhf+ReStore* configuration yields a MTBF comparable to a design about 1/20th the size.

5. Related Work

Gu et al. [8, 9] and Smolens et al. [26] explored failure modes from fault injections into general purpose registers. They also noted a high percentage of failures from exceptions. We investigate error propagation from microarchitectural state to architectural state and eventually exceptions. Furthermore, we evaluate the use of exceptions and other symptoms to aid soft error detection.

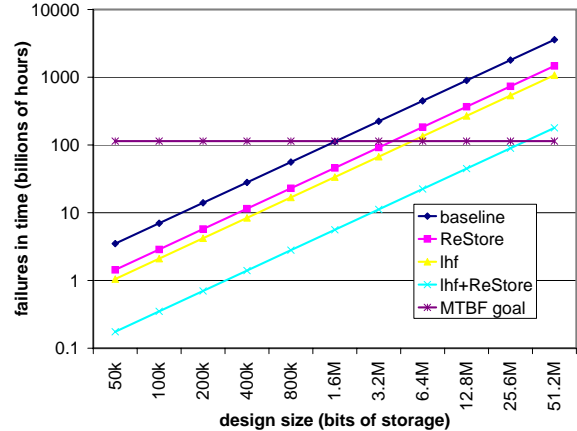


Figure 7. FIT rates with device scaling

Patel et al. [22] explore the error coverage provided by the rePLay framework, which performs checkpoint rollbacks on highly biased branch mispredictions. Like Gu et al., they performed fault injections into general purpose registers. Here, we perform a microarchitectural evaluation in a more general framework.

Previous work has explored using parity, ECC and TMR to provide spatial redundancy in processor cores [6, 10]. Franklin [4] noted different modes of failure throughout the pipeline and proposed mechanisms to guard against them. We propose an alternative, more cost effective approach to soft error detection and recovery.

Other work has introduced other forms of redundancy to mitigate the effects of soft errors [19, 25, 30, 23]. Each relies on “full-time” redundancy, where the cost of redundant execution is paid for on each instruction. Rather than utilizing redundancy to detect and recover from soft errors, Weaver et al. [31] reduce the exposure of instructions to soft errors by squashing instructions upon encountering long latency events like cache misses. Here, we introduce the ReStore architecture to provide efficient “on-demand” time redundancy.

6. Conclusion

In this work, we propose the ReStore architecture which leverages existing performance enhancing hardware for the purpose of soft error detection and recovery. Selected high level events from microarchitectural structures (soft error

symptoms) are used to diagnose the likely presence of failure causing soft errors, initiating checkpoint recoveries for detection. Thus, the ReStore architecture employs on-demand time redundancy, minimizing hardware cost and performance impact. Such an approach sacrifices some amount of error coverage, but would be suitable for environments where reliable operation is desired but not at all costs.

The baseline processor had an intrinsic error masking rate of approximately 92%, indicating that only 8 out of every 100 introduced faults propagate to persistent data corruption. With a 100 instruction checkpoint interval, an example ReStore implementation (using deadlock, exception, and store miss symptoms) detects and recovers from 59% of all failures. Protecting the most vulnerable portions of the baseline processor core and overlaying ReStore extends the mean time between failures by 20x, while incurring a 4.5% performance hit and requiring minimal additional hardware.

References

- [1] H. Akkary, R. Rajwar, and S. T. Srinivasan. Checkpoint processing and recovery: Towards scalable large instruction window processors. In *MICRO-36*, Dec. 2003.
- [2] H. Akkary, S. T. Srinivasan, R. Koltur, Y. Patil, and W. Refaai. Perceptron-based branch confidence estimation. In *HPCA-10*, pages 265–274, Feb. 2004.
- [3] J. Dundas and T. Mudge. Improving data cache performance by pre-executing instructions under a cache miss. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 68–75, July 1997.
- [4] M. Franklin. Incorporating fault tolerance in superscalar processors. In *Proceedings of High Performance Computing*, pages 301–306, Dec. 1996.
- [5] G. Hinton et al. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, Jan. 2001.
- [6] J. Gaisler. A portable and fault-tolerant microprocessor based on the SPARC V8 architecture. In *DSN-2002*, Sept. 2002.
- [7] D. Grunwald, A. Klauser, S. Manne, and A. Pleszkun. Confidence estimation for speculation control. In *ISCA-25*, pages 122–131, June 1998.
- [8] W. Gu, K. Kalbarczyk, and R. K. Iyer. Error sensitivity of the linux kernel executing on powerpc g4 and pentium 4 processors. In *DSN-2004*, June 2004.
- [9] W. Gu, K. Kalbarczyk, R. K. Iyer, and Z. Yang. Characterization of linux kernel behavior under errors. In *DSN-2003*, June 2003.
- [10] H. Ando et al. A 1.3 GHz fifth generation SPARC64 microprocessor. In *Design Automation Conference*, June 2003.
- [11] P. Hazucha and C. Svensson. Impact of CMOS Technology Scaling on the Atmospheric Neutron Soft Error Rate. *IEEE Transactions on Nuclear Science*, 47(6):2586–2594, Dec. 2000.
- [12] E. Jacobsen, E. Rotenberg, and J. E. Smith. Assigning confidence to conditional branch predictions. In *MICRO-29*, pages 142–152, 1996.
- [13] D. A. Jimenez. Fast path-based neural branch prediction. In *MICRO-36*, Dec. 2003.
- [14] A. Klaiber. The technology behind Crusoe processors. Technical report, Transmeta Corporation, Jan. 2000.
- [15] S. S. Lumetta and S. J. Patel. Characterization of essential dynamic instructions. In *SIGMETRICS 2003*, June 2003.
- [16] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors - a survey. *IEEE Transactions on Computers*, 37(2):160–174, Feb. 1988.
- [17] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Western Research Laboratory, June 1993.
- [18] S. Mitra, N. Siefert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *IEEE Computer, Special Issue on Nano-Scale Design and Test*, 38(2):43–52, Feb. 2005.
- [19] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt. Detailed design and evaluation of redundant multithreading alternatives. In *ISCA-29*, pages 99–110, May 2002.
- [20] 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 *MICRO-36*, pages 29–40, Dec. 2003.
- [21] O. Mutlu, J. Stark, C. Wilkerson, and Y. N. Patt. Runahead execution: An effective alternative to large instruction windows. *IEEE Micro*, 23(6):20–25, Nov. 2003.
- [22] S. J. Patel, Z. Kalbarczyk, R. K. Iyer, W. Magda, and N. Nakka. A processor-level framework for high-performance and high-dependability. In *Workshop on Evaluating and Architecting Systems for Dependability*, 2001.
- [23] M. K. Qureshi, O. Mutlu, and Y. N. Patt. Microarchitecture-based introspection: A technique for transient-fault tolerance in microprocessors. In *DSN-2005*, pages 434–443, July 2005.
- [24] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA-27*, June 2000.
- [25] E. Rotenberg. AR-SMT: A microarchitectural approach to fault tolerance in microprocessors. In *FTCS*, June 1999.
- [26] J. C. Smolens, B. T. Gold, J. Kim, B. Falsafi, J. C. Hoe, and A. G. Nowatzky. Fingerprinting: Bounding soft-error detection latency and bandwidth. In *ASPLOS-11*, Oct. 2004.
- [27] L. Spainhower and T. A. Gregg. IBM S/390 parallel enterprise server G5 fault tolerance: A historical perspective. *IBM Journal of Research and Development*, 43(5/6):863–873, 1999.
- [28] J. G. Steffan and T. C. Mowry. The potential for using thread-level data speculation to facilitate automatic parallelization. In *HPCA-4*, Feb. 1998.
- [29] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel. Characterizing the effects of transient faults on a high-performance processor pipeline. In *DSN-2004*, June 2004.
- [30] C. Weaver and T. Austin. A fault tolerant approach to microprocessor design. In *ISCA-29*, May 2002.
- [31] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to reduce the soft error rate of a high-performance microprocessor. In *ISCA-31*, June 2004.