SOS: Using Speculation for Memory Error Detection

Sudhanva Gurumurthi Angshuman Parashar Anand Sivasubramaniam

Department of Computer Science and Engineering
Pennsylvania State University
University Park, PA 16802
e-mail: {gurumurt,parashar,anand}@cse.psu.edu

Abstract

With the problem of transient errors in integrated circuits becoming very critical in future process technologies, it is important to develop effective techniques for detection and possible recovery from such errors. Dense integration, lower supply voltages and higher frequencies make circuits more susceptible to bit flips due to cosmic-ray strikes. Memory cells, such as SRAMs and DRAMs, typically occupy a large portion of the real-estate and are hence particularly vulnerable to these transient faults. While techniques such as parity and ECC are being used to handle such situations, such safeguards can impose significant storage, performance, and power overheads, especially in high-performance designs. This paper investigates a fresh approach to developing transient-fault detection mechanisms. We show that speculative-execution resources inside the processor can be tailored to provide detection of errors that propagate in from memory. Using detailed experiments with different benchmarks, we demonstrate that the branch-predictor in conjunction with a lightweight value-predictor can catch most of the errors that come in to the processor from the memory system. We also show the microarchitectural extensions required for implementing such error-detectors.

1 Introduction

The advent of deep-submicron process technologies in modern processor design poses a big challenge for circuit reliability. The use of smaller transistors necessitates the use of lower operating voltages. Further, the larger die sizes and the smaller transistors produce high integration densities, especially for memory structures like the on-chip caches. These trends coupled with higher clock frequencies make future microprocessors vulnerable to soft-errors [3] caused by alpha-particles from packaging material and and high energy neutrons from cosmic-rays [19]. Although soft-errors might not cause any permanent damage to the circuit, the excess charge that they induce can cause a spurious bit flip. If this change in the logic state is not corrected, it can lead to erroneous computation, possibly leading to catastrophic consequences. Large array-structures, such as caches and main memory, are especially prone to soft errors.

The importance of memory data integrity has lead to the widespread use of informational redundancy mechanisms such as parity and Error Correcting Codes (ECC) [4] to detect and correct errors. Parity can detect single-bit errors and a typical Hamming ECC can detect two erroneous bits and can correct a single bit in error. The Intel Itanium [12] and the IBM POWER4 [1] use parity in their L1 caches and ECC in the lower levels of the hierarchy. Error-correcting codes can impose significant overheads, in terms of storage, performance, and power. In mission-critical high-end systems such as the IBM S/390 G5 [5] and the HP NonStop Himalaya [7], aggressive fault-tolerance techniques such as radiation-hardening and complete hardware duplication are employed. However, given that, for most of the microprocessors, the goal is to meet a desired *reliability budget* rather than complete error-coverage [17], we require scalable mechanisms that can provide adequate protection against soft-errors without siginificantly degrading performance or power.

Another issue of growing concern pertains to transient errors in combinational logic [14]. This has fueled research in verifying integrity of logic in the processor datapath via temporal redundancy [11, 13, 15, 6]. These proposals provide fault-tolerance by redundant execution of the program rather than by the use of duplicate hardware or ECC. Specifically, they define a *Sphere of Replication*, within which fault-detection/recovery are provided by temporal redundancy - executing redundant instruction streams, and verifying that they produce the same output. These proposals assume that memory-hierarchy structures such as the caches and DRAM lie *outside* the Sphere of Replication and that they need to be protected via other means.

In this paper, we propose a new approach to construct scalable error-handling mechanisms for the memory-hierarchy. Our approach is based on one salient trend in microprocessor-design, namely, the use of speculative-execution. High performance processors employ a considerable amount of speculation to predict application behavior and allocate hardware resources in an effective manner based on such prediction. Two such important speculation mechanisms include branch-predictors [18] and value-predictors [8], which can be employed to overcome control and data hazards in the pipeline, thereby effectively improving the par-

allelism in the execution. Speculation can be viewed as a *virtual run-ahead thread* of the program, capturing the control and data-flow aspects of its execution. However, the speculation would need to be resolved to ensure its correctness before committing the executed instructions and changing the architected state of the machine. In this paper, we show that these speculation mechanisms can be adapted to check the integrity of the data coming into the processor from the memory-hierarchy.

The main idea is to use the built-up history of control-flow and data values in the speculative structures as an indication of what is to be expected of the non-erroneous execution. An erroneous datum accessed from memory can possibly cause a deviation (misspeculation) from this historical information, i.e. an erroneous load may cause the value that comes into a register to deviate from that predicted by a value-predictor, or can cause a branch to head to a different target from that predicted by the branch-predictor. This is the key intuition behind our *Sphere of Speculation (SOS)* approach, wherein the on-chip speculative resources can be used to enhance the detection of errors propagated into the datapath. The portions of the processor that are covered by speculative execution can be considered to be within this logical sphere. SOS can be used as a "second-chance" mechanism for detecting errors arising in memory. With the increased error-detection coverage that SOS can provide, it would be possible to use simpler codes (eg. parity instead of ECC) to satisfy the reliability budget. This can provide reduction in the storage-demands and power consumption of the coding schemes.

Note that a misspeculation does not necessarily signify that a soft-error has occurred (*false positive*), nor is this mechanism expected to catch all errors propagated from memory (*false negative*).

Using fault-injection and a set of benchmarks from the SPEC2000 suite, we show that a large fraction of the memory errors that percolate into the processor datapath trigger a value or branch-misprediction before any computation that uses this erroneous value gets stored back into the memory system. We also show how recovery mechanisms can be provisioned when SOS indicates a potential memory error, while alleviating any problems due to the lag between error occurrence and detection.

The rest of the paper is organized as follows. In section 2, we present the rationale behind our SOS scheme and how it could be used in processor designs to achieve fault-tolerance. The simulation infrastructure and system configuration used in experiments are presented in Section 3. We present the results of our study in Section 4 and Section 5 concludes the paper.

2 The Sphere of Speculation (SOS)

Modern processors offer a multiplicity of hardware to exploit parallelism. However, the achievable parallelism is impeded by the presence of control and data dependencies. In order to overcome these limitations, processors make use of speculative execution, whereby the behavior of the program is *predicted* and execution proceeds past these dependencies, even though the requisite information may not be actually available. However, in order to ensure correct execution, the speculation is *resolved* and if found to be incorrect, all the instructions that have been executed as a consequence of the speculation are squashed and execution is re-started from the point where speculation was performed. The microarchitecture guarantees that the speculative instructions do not change the architected (i.e., externally visible) state of the machine until they are resolved to be in the correct execution flow. The speculative mechanisms for overcoming control and data-flow limitations that have been proposed are branch-prediction [18] and value-prediction [8] respectively.

Protection from memory system errors involves two main goals, namely,

- Detecting the error and, if possible, correcting it at the source.
- Preventing the propagation of any newly detected errors. If an error is not detected on time, then its effect can propagate further. There may be different desirables on how much propagation should be allowed. In the scope of our work, *we are trying to avoid the propagation of errors back into the memory system*, i.e. an error read from memory can lead to erroneous computations within the processor datapath, but we are trying to avoid the propagation of any such errors back into the memory system via processor store operations. In effect, we consider the consequences of a computation to be erroneous (based on erroneous data values read) only when the output from the processor datapath produces erroneous values.

The standard approach to meeting the first goal has been by incorporating error-correction codes in the memory structures. However the use of such coding schemes can entail significant storage, performance, and power costs. In this paper, we propose a new approach to allow a second chance at detection of errors that escape any first line of defense provided by informational redundancy. Further, for errors that have been detected, we propose minor hardware extensions to reduce the chances of the error from propagating to the memory system. Our results show that our mechanism can detect a large fraction of the errors that enter the processor datapath. We achieve these goals by exploiting a key property of speculative execution.

Speculation attempts to break potential performance limiters such as branches, data dependencies, etc. by using prediction based on an observed history. Speculation thus attempts to predict the anticipated program behavior. A branch-predictor predicts the control-flow of the program, trying to determine the execution path following a branch instruction. On the other hand, a value-predictor predicts the data-flow of the application, guessing the actual values that an operand would take and propagating it down the dependency chain. The speculative engines in the processor can thus be viewed as a *virtual run-ahead thread* of the actual execution context, capturing the data and control-flow aspects of the execution. This property of speculative execution suggests an interesting possibility from the fault-tolerance viewpoint: *If specula-*

```
1:    r1 <- load @a
2:    r2 <- load @b
3:    if (r1 < r2) jump t
4:    r3 <- r1 - 1
         .
         .
         .
t:    r1 <- r1 + 1
t+1:  jump 3
```

Static code

```
@a = 10, @b = 50

r1 <- load @a
  <Value Prediction: @a = 10>
r2 <- load @b
  <Value Prediction: @b Don't Predict>
  <Resolution: @a = 10;
   Prediction Correct>
if (r1 < r2) jump t
  <Branch Prediction: Branch Taken>
r1 <- r1 + 1
  <Resolution: Branch Taken;
   Prediction Correct>
if (r1 < r2) jump t
    .
    .
    .
```

(a) Dynamic execution sequence

```
@a = 97 (Error), @b = 50

r1 <- load @a
  <Value Prediction: @a = 10>
r2 <- load @b
  <Value Prediction: @b Don't Predict>
  <Resolution: @a = 97;
   Prediction Incorrect>
```

(b) Data-flow error

```
@a = 10, @b = 3 (Error)

r1 <- load @a
  <Value Prediction: @a = 10>
r2 <- load @b
  <Value Prediction: Don't Predict>
  <Resolution: @a = 10;
   Prediction Correct>
if (r1 < r2) jump t
  <Branch Prediction: Branch Taken>
r1 <- r1 + 1
  <Resolution: Branch Not Taken;
   Prediction Incorrect>
```

(c) Control-flow error

Figure 1. The use of speculation to catch erroneous loads with data-flow and control-flow prediction.

tion/prediction by the virtual run-ahead thread can give a good indication of anticipated program behavior, then can we use a deviation from this prediction as an indication of an error?

An error that propagates into the datapath can possibly affect the data values and control-flow of the program. For instance, if the value-predictor anticipates a value that is different from what is actually loaded, it is possible that there could be an error in the value actually loaded (though it is also quite possible that this is really a misprediction rather than an error in the value loaded). Sometimes even if the value-predictor does not flag an error/misprediction, it is possible that the resulting value gets used in some computations, causing some deviations in the control-flow (deviation from branch history) that could be caught by a branch-predictor. Examples of such scenarios are given in Figure 1.

Consequently, speculation can be viewed as providing a logical sphere of fault-detection, which we call the *Sphere of Speculation (SOS)*, (analogous to the Sphere of Replication concept presented in [13] for redundant execution architectures) of memory errors that may percolate into the datapath. The SOS mechanism that we present next is an implementation of this idea, which uses load-value speculation [9] and branch speculation as safeguards against memory errors. In SOS, if the load-value predictor itself flags the error (i.e. it is flagged before committing the load instruction that brings in the erroneous value as shown in Figure 1(b)), then the existing speculation mechanism suffices to squash that instruction (and any subsequently dependent instructions) without having the error affect the architected state (i.e. any changes to the architected register-file, memory, etc. that are externally visible to the application) of the machine. In case the load-value predictor does not catch such an error, it is possible that a subsequent branch prediction invocation may catch it (as in Figure 1(c)). However, the time gap between the loading of the error into the datapath and the flagging of this error by the branch-predictor can cause some instructions to possibly load erroneous values into the register file. Our SOS mechanism tries to adhere to the second goal mentioned earlier in this section wherein we try as far as possible to avoid propagating the errors to memory. The mechanisms for achieving this are discussed in the next subsection.

In the SOS approach, a misprediction in one of the speculative engines, which we refer to as a *SOS event*, could indicate a potential error in the memory-hierarchy. On such an event, we can consult the parity-bits to verify if this truely was an error (instead of proactively signalling a bus-reset on a parity-error). At this point, based on the degree of fault-tolerance required, more aggressive error-detection/correction mechanisms (such as comparing cache-lines in L1 and L2) may be invoked, which is normally not an attractive option due to performance and power reasons.

2.1 Microarchitectural Extensions for Implementing SOS

We now move on to the microarchitectural extensions required to implement SOS-based detection and recovery policies.

Detection: Note that in SOS, an error detection mechanism is triggered at either the load instruction that causes a value-misprediction or by a branch-predictor that anticipates a different control flow. In the former case, the address that needs to be further checked is present in the load instruction itself that is

being speculated on, and the instruction has not been committed to affect the architected state of the machine. However, in the latter case, it is not clear what the address of the loaded erroneous (if at all) value was. It may have been the last loaded value from memory, and possible even one or more of the last few loaded values. In order to check the integrity of those data values, we need to maintain a history of the addresses that were recently accessed by load instructions. It may appear that the entry for the load instruction in the reorder-buffer or the load/store queue could be used for this purpose. However, the load instructions that need to be checked could have already been retired from the processor before the occurrence of the SOS event, and there may not be corresponding entries in the reorder buffer or load/store queues. Therefore, we propose the inclusion of a **Load History Queue (LHQ)**, which is a circular queue (like the reorder-buffer) with head and tail pointers, that maintains a history of the load addresses and the PC of the load instruction in program order.

When a load instruction is encountered, there are five possibilities: (i) the value-predictor is not able to make any prediction (there is not sufficient history built up), (ii) there is no error in the loaded value, and the prediction does not match what is loaded, (iii) there is no error in the loaded value, and the prediction matches what is loaded, (iv) there is an erroneous value, and the prediction does not match this, and (v) there is an erroneous value, and the prediction matches this. Case (v) is a situation that is not caught by our SOS mechanism (i.e. the loaded value is viewed to be correct), though it is expected that this would be a rare occurrence and would probably not significantly affect the detected-error budget. In cases (ii) and (iv), a SOS event is raised right at the load instruction, and an additional error checking mechanism (such the parity) is invoked right at those points to check the integrity of the loaded value. Consequently, it is only in case (i) that our SOS mechanism records the load instruction (the PC and address) in the LHQ.

It is then up to the branch speculation mechanism to catch any errors (on a best-effort basis) that may not have been caught by the value-predictor. On a SOS event due a branch misprediction, we scan the LHQ, starting from the earliest entry, and perform more aggressive memory integrity checks on *all* those addresses. After an address is checked and no error is found, its entry is freed from the LHQ.

Recovery: When an error is detected on a value being loaded by the value-predictor, the existing microarchitectural mechanism that is used to rewind the speculation suffices, i.e. the load instruction and any other dependent instructions using the speculated value are squashed. On the other hand, it takes more extensive support in order to recover from errors that were detected at the time of a branch misprediction. It is possible that the load instruction that accessed the incorrect value has (i) either not yet committed, or (ii) it has committed. In the former case, we can again squash that load (and any dependent instructions). In the case that the erroneous load has committed, it could have impacted the architected state (the ARF, the memory system, etc.) of the machine that is seen by the application code.

Handling the case of erroneous loads that have committed to get back to the original state requires undoing all the effects to the architected machine state since then. While one could think of shadow register-files into which stores are buffered (until "safe" points) to undo the modifications to register files, in this current work we limit ourselves to the goal of only avoiding the propagation of erroneous values to the memory system as stated early in this section, and postpone an investigation of undo-ing the effects on datapath components such as register files for future work.

To limit the propagation of the errors back into the memory system (via a store instruction), we propose a **Safe Store Queue (SSQ)**, which buffers stores to the memory system in program order (similar to the PET buffer that has been proposed for tracking potentially-incorrect instructions in the processor pipeline [17]). In our proposal, LHQ and SSQ operations are closely intertwined with the former indicating what needs to be checked and the latter buffering the store values until all loads prior to this store (in program order) have been verified to be correct. Every time an entry is created in the LHQ, a pointer is created from the LHQ entry to the first free entry in the SSQ, i.e. the stores that would subsequently be inserted in SSQ can be flushed after this load is verified. When an error is detected at a particular address (during a branch misprediction), we throw away all the entries in the SSQ starting from the store pointed to by the corresponding LHQ entry. On the other hand, when a load address is verified to be correct, the SSQ entries pointed to by that LHQ entry (until the next LHQ entry) can be flushed/propagated to memory. The operation of the two queues is illustrated in Figure 2. Whenever the queues are full, it is possible to invoke a SOS event to make sure the queues get emptied before proceeding. In our experimental results, we will show that we do not need large queue sizes (since large sizes become a performance problem or are difficult to implement).

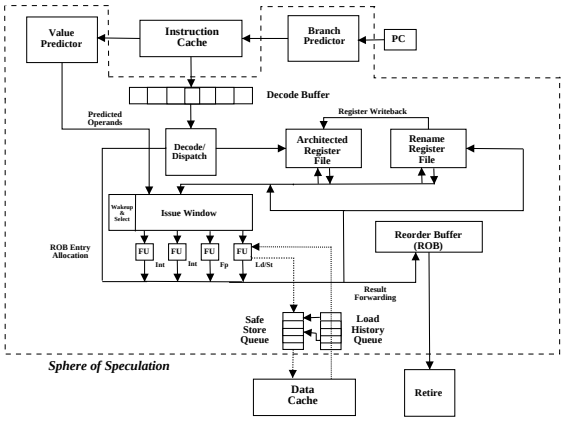


Figure 3. SOS-based processor with Load History Queue and Safe Store Queue. The core is augmented with a load value-predictor to track the data-flow. The SOS is shown in dashed-lines.

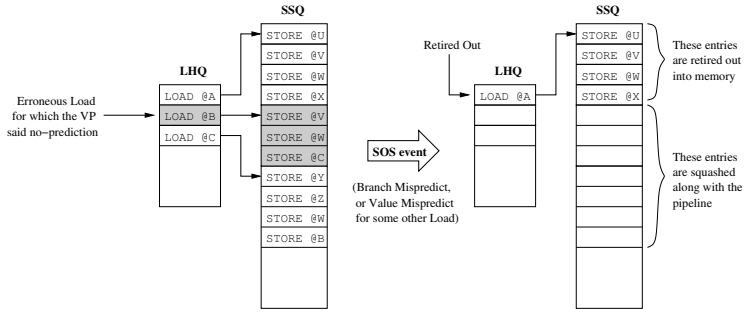


Figure 2. Operation of Load History Queue and Safe Store Queue When Handling an Erroneous Load

We also note below some additional details in the proposed microarchitecture:

- Both the LHQ and the SSQ are updated during the decode/dispatch stage of the processor pipeline in order to ensure that the entries are in program order.
- In the SOS approach, a value misprediction could cause the load instruction to be re-issued to the memory-system (after a correction) if an error is detected, whereas under fault-free conditions, execution would resume from the instruction *following* the load after recovering from misspeculation.
- As mentioned previously, in order for a recovery mechanism to re-start execution, the state of the architected register-file would need to be restored to the state during the execution of the recovered load instruction. It may be possible to develop a technique similar to our LHQ/SSQ mechanism for un-doing register updates, though a detailed investigation is part of our future work.
- The operations on the LHQ and SSQ do not require any full-scans/broadcasts, unlike those for the issue-queue or the load/store queue. This makes their design scale to a larger number of entries.

The processor datapath with the proposed microarchitectural extensions is given in Figure 3. Some possible error scenarios along with the procedure followed by SOS to handle them is shown in Figure 4.

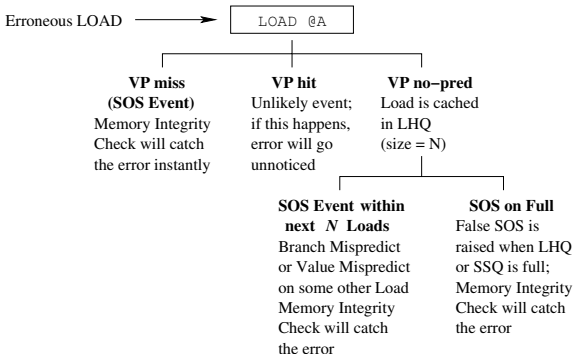


Figure 4. Possible error-scenarios and recovery steps.

3 Experimental Setup

Our experiments were carried out via execution-driven simulation of a set of Spec2000 benchmark applications on SimpleScalar 3.0 [2]. We modified the simulator for performing error-injection experiments. The benchmarks were compiled for the PISA instruction set architecture with the `-O2 -funroll-loops` optimization flags. The reference input set was used for the simulations. We employ the two-level value-predictor proposed by Wang and Franklin [16]. We perform value-prediction for load instructions only [9]. The system configuration that we simulated is given in Table 1.

In our evaluations, each benchmark was first fast-forwarded by 1 billion instructions to skip over the initialization phase. After this, the execution was divided into *windows* of instructions, with each window consisting of 10 million instructions. We simulated each such execution window independently and analyzed several successive windows beyond the initialization phase to get reasonable statistical confidence in the results. We inject an error (a single bit-flip) into the data word that is read upon a load. (Although there exists the possibility that a particle-strike may cause a multi-bit error, the probability of such an error is several orders of magnitude lower than that of single-bit errors [17]. Furthermore, techniques such as bit-interleaving [1] and scrubbing [10] further reduce the likelihood of such errors). In each execution window described above, we introduce one such erroneous data word load and study its ramifications on the execution. Since each window is analyzed individually, an error in one window does not percolate to another (i.e. its effect is studied for only 10 million instructions), and each window of 10 million instructions can be viewed as an independent simulation run.

We use the SimpleScalar simulator, augmented with fault-injection support. This simulation infrastructure, runs two instances of SimpleScalar (with and without error-injection), with each of them (running as separate processes) dumping their execution traces to a shared memory region that is mapped into a control program. This program freezes the simulators periodically (whenever the trace size reaches a limit), and analyzes the two traces to find out how the execution with the error differed from that without. After freeing up this trace space, it resumes the two simulation

Processor Parameters	
Fetch/Decode/Issue/Commit Width	8
Fetch-Queue Size	8
Branch-Predictor Type	Combined predictor; 16K-entry meta-table L2 adaptive predictor with 16K-entry L1 and 16K-entry L2 table, and 11-bit history XORed with address in L2 predictor
RAS Size	64
BTB Size	2K-entry 2-way
Branch-Misprediction Latency	7 cycles
Value-Predictor Type	2-level predictor
Number of VHT Entries	4K
Pattern Size	6
Number of Values per VHT Entry	4
Number of PHT Entries	4K
PHT-Counter Saturation Value	12
PHT-Counter Threshold	3
Increment-Value for	3 (Others: Decrement Correct Prediction by 1)
RUU Size	128
LSQ Size	64
Integer ALUs	4 (1-cycle latency)
Integer Multipliers/Dividers	2 (3,20)
FP ALUs	2 (2)
FP Mult./Div./Sqrt.	1 (4,12,24)
L1 Cache Ports	2
L1 D-Cache	32KB, 2-way with 32B line-size (2)
L1 I-Cache	64KB, 2-way with 32B line-size (2)
L2 Unified Cache	512 KB, 4-way with 64B line-size (12)
TLB Miss-Latency	30 cycles
Memory Latency	112 cycles

Table 1. Simulation parameters. Latencies of ALUs/caches are given in parenthesis. All ALU operations are pipelined except division and square-root.

instances.

4 Results

4.1 Metrics

A comprehensive evaluation of all aspects of SOS, including its implementation and power ramifications, are beyond the scope of this paper. Instead, we focus on two main metrics, namely, *coverage* and *check-fraction*, to examine the potential of this technique.

- **Coverage**: We say that a memory error propagated into the datapath via a load instruction is *covered* if there is a SOS event between the time of the load and the first store instruction that is on the dependency chain of the erroneous load, i.e. SOS invokes a check for this erroneous load before any effect of this load propagates back to the memory system. The check could be done either at the corresponding load itself, or at the time of a branch misprediction as long as there is no erroneous store going back out to memory in the meantime. We define the coverage provided by SOS as

$$Coverage = \frac{Number of Covered Loads}{Total Number of Erroneous Loads}$$

  The coverage gives an indication of how well SOS can detect potential memory errors that have percolated into the datapath and prevent the propagation of new errors back to memory. Note that errors that occur in memory but are never loaded into the processor are not considered to be an error.

- **Check-Fraction**: This is defined as the fraction of load addresses that are subjected to additional memory integrity checks using SOS compared to the baseline case where all the loads may need to be checked aggressively. Specifically, the check-fraction is defined as

$$Check - Fraction = \frac{Number of Detailed Checks}{Number of Memory Loads}$$

  The check-fraction captures the possible performance and power savings (lower the fraction, higher the savings) of SOS compared to more extensive memory integrity checking techniques that may need to perform these checks for each load.

4.2 The Potential of SOS

In Figure 5, we show the coverage and check-fraction using SOS for five applications from the SPEC2000 suite. We break down the coverage into that provided by the value-predictor and that provided by the branch-predictor (when an error has even percolated through the value speculation mechanism of SOS) at a subsequent point before an erroneous store propagates to memory. The data presented is an average over all the execution windows for each application. We find that the two speculation mechanisms put together can provide a fairly good coverage, flagging a closer scrutiny (using a more extensive technique) of 82% of the errors that have propagated into the datapath. In fact, in applications such as *bzip* and *mcf* we get a coverage over 95%. Of the two speculation mechanisms, the value-predictor mechanism is able to provide the bulk of the coverage (72-98% on the average) right at the point of the load. At the same time, we find that the branch-predictor is an useful additional line of

defense against errors that even go undetected by the value-predictor. The sizes of the LHQ and SSQ (described in Section 3) required to achieve this coverage shall be presented shortly.

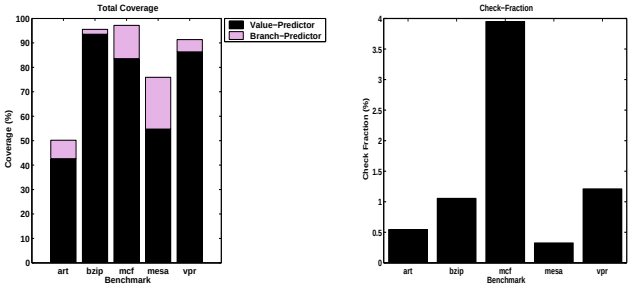
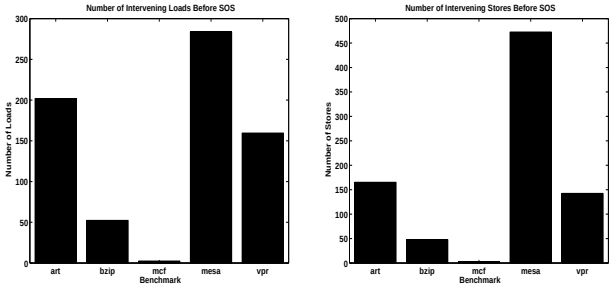


Figure 5. Potential of SOS - Coverage and Check-Fraction

Having seen that SOS can catch many of the memory-errors within the datapath, the next issue to examine is whether the number of detailed checks are reduced. It is quite possible that even though the detailed checks were avoided before the data values were brought in to the processor, SOS could still be raising several events (possibly even false alarms) that can end up performing as many detailed checks as before. The check-fraction in Figure 5 shows that detailed checks are performed for a very small fraction (less than 4% in the worst case, and around 1% or less in the others) of the loaded addresses suggesting that there could be performance (saving latencies of complicated checks at the time of the load) and power savings with the SOS mechanism, while not significantly compromising on the error coverage as our results have just shown.

The implementation of the SOS mechanism mandates the maintenance of the LHQ (to keep track of loads that need to be checked further) and SSQ (to buffer the stores going out to memory in order to undo their effects if needed). It becomes important to study the sizes of these queues for an effective SOS mechanism. If the queues fill up, then the execution may need to stall while detailed checks are performed on the load addresses (even if they may not be needed), and the entries get flushed. If, on the other hand, an option of over-running these queues is chosen, then one may not be able to get good coverage (i.e. some of the loads may not get checked, or some of the erroneous stores in SSQ may propagate to memory). As mentioned earlier, we employ the former option, waiting for empty slots in the queue by performing detailed checks on the entries in the LHQ, rather than compromise on the reliability. Appropriate LHQ/SSQ sizes are necessary in order for this mechanism to not become a performance overhead.

In Figure 6 we plot the average LHQ size and SSQ size during the execution of each benchmark that is needed to provide the error coverage results depicted earlier in Figure 5. As can be seen, we need to maintain only a few hundred entries in each of these queues in order to not stall often for an empty slot in the queues.



(a) Average LHQ Size (b) Average SSQ size

Figure 6. Average sizes of LHQ and SSQ in terms of the number of entries

5 Concluding Remarks

In this paper, we explore the use of speculation to provide transient fault-tolerance for the memory-system. Our SOS mechanism, which uses speculative structures, which are typically implemented for boosting performance, to detect errors in memory. By using points of deviation from previously observed behavior as events to further check the integrity of the data that has been loaded, SOS is able to catch a large fraction of the errors (over 95% in some cases, and 82% on the average) that enter the datapath. At the same time, the number of extensive integrity checks that it performs is significantly lower (less than 5%) than performing such checks on every load, thereby having the potential to enhance performance and reduce power consumption. We illustrated extensions to the processor datapath for limiting the propagation of memory errors between occurrence and detection with our SOS mechanism.

Our current and future work involves refining this technique to improve the error-coverage. Furthermore, we shall investigate whether speculation could be employed to boost the performance of redundant-threading mechanisms, which provide error-detection for the processor logic.

6 Acknowledgements

This research has been supported in part by NSF grants 0103583, 0325056, and 0130143.

References

[1] D. Bossen, J. Tendler, and K. Reick. Power4 System Design for High Reliability. *IEEE Micro*, 22(2):16–24, March 2002.

[2] D. Burger and T. Austin. The SimpleScalar Toolset, Version 3.0. http://www.simplescalar.com.

[3] C. Constantinescu. Trends and Challenges in VLSI Circuit Reliability. *IEEE Micro*, 23(4):14–19, July-August 2003.

[4] C.L. Chen and M.Y. Hsiao. Error-Correcting Codes for Semiconductor Memory Applications: A State-of-the-Art Review. *IBM Journal of Research and Development*, 28(2):124–134, March 1984.

[5] T. S. et al. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, 19(2):12–23, March 1999.

[6] M. Gomaa, C. Scarbrough, T. Vijaykumar, and I. Pomeranz. Transient-Fault Recovery for Chip Multiprocessors. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 98–109, June 2003.

[7] HP NonStop Himalaya. http://nonstop.compaq.com/.

[8] M. Lipasti and J. Shen. Exceeding the Dataflow Limit via Value Prediction. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 226–237, December 1996.

[9] M. Lipasti, C. Wilkerson, and J. Shen. Value Locality and Load Value Prediction. In *Proceedings of the International Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 138–147, October 1996.

[10] S. Mukherjee, J. Emer, T. Fossum, and S. Reinhardt. Cache Scrubbing in Microprocessors: Myth or Necessity? In *Proceedings of the International Symposium on Pacific Rim Dependable Computing (PRDC)*, pages 37–42, March 2004.

[11] Z. Purser, K. Sundaramoorthy, and E. Rotenberg. A Study of Slipstream Processors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 269–280, December 2000.

[12] N. Quach. High Availability and Reliability in the Itanium Processor. *IEEE Micro*, 20(5):61–69, September 2000.

[13] S. Reinhardt and S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 25–36, June 2000.

[14] P. Shivakumar, M. Kistler, S. Keckler, D. Burger, and L. Alvisi. Modeling the Effect of Technology Trends on Soft Error Rate of Combinational Logic. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2002.

[15] T. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-Fault Recovery via Simultaneous Multithreading. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 87–98, May 2002.

[16] K. Wang and M. Franklin. Highly Accurate Data Value Prediction using Hybrid Predictors. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 281–290, December 1997.

[17] C. Weaver, J. Emer, S. Mukherjee, and S. Reinhardt. Techniques to Reduce the Soft Error Rate of High-Performance Microprocessor. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 264–275, June 2004.

[18] T.-Y. Yeh and Y. Patt. Alternative Implementations of Two-Level Adaptive Branch Prediction. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 124–134, May 1992.

[19] J. Zeigler. Terrestrial Cosmic Rays. *IBM Journal of Research and Development*, 40(1):19–39, January 1996.