Transient Fault Prediction Based on Anomalies in Processor Events

Satish Narayanasamy[†] Ayse K. Coskun [†] Brad Calder[†‡]

[†]University of California, San Diego [‡]Microsoft

Abstract

Future microprocessors will be highly susceptible to transient errors as the sizes of transistors decrease due to CMOS scaling. Prior techniques advocated full scale structural or temporal redundancy to achieve fault tolerance. Though they can provide complete fault coverage, they incur significant hardware and/or performance cost. It is desirable to have mechanisms that can provide partial but sufficiently high fault coverage with negligible cost.

To meet this goal, we propose leveraging speculative structures that already exist in modern processors. The proposed mechanism is based on the insight that when a fault occurs, it is likely that the incorrect execution would result in *abnormally* higher or lower number of mispredictions (branch mispredictions, L2 misses, store set mispredictions) than a correct execution. We design a simple transient fault predictor that detects the anomalous behavior in the outcomes of the speculative structures to predict transient faults.

1. Introduction

As the scaling of CMOS technology continues, the reduced voltage and noise margins, high integration levels and high clock speeds will increase the susceptibility of transistors to transient faults (also known as soft errors) due to cosmic rays and electrical noise [6].

Most of the transient fault tolerance techniques that exist today either resort to structural redundancy [17, 2] where the same program is executed on two different processors and their outputs are compared; or resort to temporal redundancy [13, 15] where the same program is again executed twice but on the same processor time multiplexing for the available resources. While these techniques provide very high fault coverage, structural redundancy increases the cost of the system while temporal redundancy causes severe performance overhead. Redundancy either in the form of structural or temporal is required only in the highly specialized servers and mission critical applications that require complete protection against transient faults. But for the rest of the desktop and commodity servers, processor manufacturers tend to trade off performance/hardware cost with reliability. For such systems that do not require complete fault tolerance, it is desirable to provide partial fault coverage with very low hardware cost and performance overhead, which is the goal of this paper.

To provide partial fault coverage at low cost, we propose leveraging the architectural structures that are commonly used in today's processors to predict the occurrence of a transient fault. We can recover from a predicted transient fault by replaying from a past checkpoint. Our prediction mechanism is based on the following insight. Whenever a program's behavior changes, most often it affects the number of mispredictions in the speculative structures and the number of cache misses. Thus, if a transient fault affects the correctness of a program's execution, it is very likely to cause anomalies in the behavior of processor structures.

Wang and Patel [19] proposed a similar anomaly based approach called ReStore to predict transient faults. However, ReStore treats every high confidence branch misprediction as an anomaly to predict a transient fault. We improve this basic transient fault predictor in several ways.

We show that it is not performance efficient to predict a transient fault on every high confidence misprediction, which would lead to a large number of unnecessary rollback and replays. Instead of using single misprediction event in isolation, we propose to examine an *interval* of program execution. A transient fault is predicted only if the absolute difference between the number of mispredictions seen in the interval and the "expected" number of mispredictions is greater than a *threshold* value. We determine the expected number of mispredictions using a simple history based predictor that is based on the number of mispredictions observed in the past few intervals. We also propose a mechanism to adaptively vary the threshold to bound performance overhead. We build the transient fault predictor based on the outcomes of caches and branch predictors. In addition, unlike prior approaches, we consider using the outcomes of highly accurate store set predictors [5] to predict transient faults that cause errors in the memory address stream. Also, in our scheme we effectively combine the decisions made by various predictors.

The result is an efficient transient fault predictor that does not require structural duplication and provides better performance-reliability design points. For example, we can detect 34% of the silent data corruption faults for a performance overhead of less than 10%.

2. Fault Model and Methodology

We now describe our simulation methodology and the model we use to simulate the effects of transient faults. This section also describes the fault recovery mechanism that we used.

2.1 Simulation Methodology

We used the SimpleScalar/Alpha 3.0 tool set [4] to quantify the fault coverage and the performance overhead of our technique, and ran the integer benchmarks from the SPEC2000 suite. We simulated a sample interval of 100 million instruction for each benchmark which was chosen using Simpoint [16]. Also, the processor structures (caches, branch predictors, store set predictors) were warmed up by simulating them for 1 million instructions before the simulation interval. This is important to accurately analyze the faults that we inject at the beginning of the interval. To get deterministic re-execution of the program across various simulation runs, we used EIO (External I/O) checkpoint traces in SimpleScalar, which consist of the input values from the external system. The configuration of our out-of-order processor model is presented in Table 1.

2.2 Fault Model

The fault model we assume is the single event upset (SEU) model, in which only a single bit can get corrupted at a time. We assume that the caches are protected through ECC. Our goal is to detect and recover from the transient faults that occur in the processor pipeline.

When a transient fault occurs in the processor pipeline, it can get *masked* (that is, outcome of the program execution is not affected). If a transient fault is not masked, then it propagates to the registers and/or the program counter before it can corrupt the memory subsystem. Our focus is to detect these unmasked faults. Therefore, in

Simulation Configuration	
Fetch Width	8 inst
Issue Width	4 inst
Func Units	4-ialu, 1-imult, 2-mem, 3fpalu, 1-fpmult
Reorder buf	RUU: 64, LSQ: 64
L1D	64KB, 2 way, 64B Block, 3 cycle latency
L1I	64KB, 2 way, 64B Block, 3 cycle latency
L2 Unified	2MB, 16 way, 64B Block, 20 cycle latency
DTLB	128 entry, 30 cycle miss penalty
ITLB	64 entry, 30 cycle miss penalty
Memory	275 cycle latency
Branch Pred	16K meta chooser between gshare (8K entry) and bimodal table (8k entry); 16 Return Address Stack; 1024 BTB (4-way); 12 cycle misprediction penalty
Store Sets	Last Fetched Store Table (128) Store Set ID Table (4096)

Table 1. Simulation model.

order to model a transient fault, we randomly choose an instruction in the 100 million simulated sample interval and toggle the value in instruction's output register after executing the instruction. In our experiments, we analyze 1000 random faults. Such a method is expected to have an overall error margin of less than 0.9% at a 95% confidence level [19]. A similar approach was used by earlier works such as SWIFT [15].

Like in [20], we declare that a fault is masked if the addresses and outputs of store instructions for the 10,000 instructions executed after the fault injection matches with that of the correct execution. This would also require that the branch decisions match as well. For the faults we injected, 13% of faults are masked and about 11.7% of faults lead to an exception. Masked faults are not a cause of concern, and the faults causing exceptions can be easily caught and recovered. Therefore, in the results that we report in the later sections we focus on the 75.3% of faults, which cause data corruption.

2.3 Fault Recovery Model

When a transient fault is predicted, the execution is rolled back to a previous checkpoint and replayed from thereon. Akkary et al. [1] proposed a hardware checkpointing mechanism to improve performance by increasing the instruction window size. We assume similar checkpointing support. In our approach, recovery is initiated after a fault prediction event. The cost of recovery (cycles spent in roll-back and re-execution) is often overlapped with misprediction penalties and stalls due to cache misses, etc.

We need to know if a recovery was due to a false prediction. This is required to implement our adaptive approach that we describe in Section 4. It is also required to make sure that the re-execution is not affected by a transient fault. We determine a false trigger by maintaining a running signature that is a hash of the output of store instructions executed since the beginning of the last checkpoint [18]. When the execution is rolled back and re-executed, we compare the signature generated during replay with the signature generated from the previous execution. If they match, then it is a false trigger. Also, the replay is guaranteed to be free of transient faults. However, if they don't match, then we cannot be sure which of the two executions is correct. Hence, we re-execute once more for a third time. If the third execution's signature match that of the second execution, then the original prediction is determined to be correct. Note that the third execution will be initiated only rarely when a transient fault has occurred.

We guarantee forward progress by suspending the fault detection during roll-back and replay. Before a rollback, we record the roll-back point in the program's execution by recording the number of instructions that are rolled back. Then during replay, no further recovery would be initiated until the execution gets past the rollback point, and the only time we will roll-back twice is when there is a difference in signatures as described above.

3. Predicting Anomalies

We next describe how we predict transient faults based by detecting anomalies in the outcomes of different structures in the processor.

3.1 Transient Fault Predictor

We use the standard architectural structures and identify any anomalous behavior in their outcomes to predict a transient fault. We consider that a structure is exhibiting anomalous behavior when it produces an abnormal number of *undesirable outcomes* over a period of time. For a branch predictor an undesirable outcome is a misprediction. For an L2 cache it is an L2 cache miss.

Instead of looking at each undesirable outcome in isolation, we predict anomalous behavior by monitoring all the outcomes over an interval of execution (called *monitoring interval*). The insight here is that the number of *undesirable outcomes* for a structure within the monitoring interval will differ significantly between the faulty and the correct execution.

The *monitoring interval* for a structure consists of N *uses* of that structure – that is, the number of times the structure is accessed. We do not use the number of committed instructions to determine the monitoring interval. Because in an interval determined using committed instructions, different structures have different number of uses. For example, the number of branches executed in an interval of 1000 instructions can vary considerably depending on program characteristics. Thus, we define the monitoring interval for branch predictor as the number of branches executed. For the store set predictor, the interval length is defined in terms of the number of loads executed. For L2, each L2 access (L1-miss) is considered a use.

To predict a transient fault, we first calculate an *expected* number of undesirable outcomes for a monitoring interval (which is what we consider as normal behavior). This is calculated simply by averaging the number of undesirable outcomes observed in each of the past *M* monitoring intervals. We found that setting M as 10 provides an accurate estimate of the number of events for the current monitoring interval. At the end of a monitoring interval, we compare the number of undesirable outcomes seen in that interval with the *expected* number of events. If the absolute difference is greater than a threshold T, then we predict that a transient fault has occurred. The following equation summarizes the predictor logic:

```
| (Avg # undesirable outcomes in past 10 intervals)
  - (# of undesirable outcomes in current interval) | >= T
```

Note that we check to see if the absolute difference is greater than the threshold value. This means that we will capture a transient fault that results in fewer outcomes than the expected value. For example, consider a fault that changes the control in such a way that a `for` loop ends up looping indefinitely performing the same computation over and over again. This will cause the number of branch mispredictions (and perhaps the cache misses) in the subsequent intervals of execution to be fewer than normal. Our predictor can capture such faults as well.

We experimented with various lengths for the monitoring interval and found that counting 100 uses works effectively for predicting transient faults for our benchmarks. If the monitoring interval is large, the processor would have to roll back a large number of instructions when a fault is predicted. If we choose the number of uses to be too small (e.g., 10), then it would be difficult to determine the average (normal) behavior.

3.2 Speculative Structures Used for Identifying Faults

We predict transient faults based on the outcomes of branch predictor, store set predictor and L2 cache accesses. These structures are able to catch faults that manifest in control and address streams. We also analyzed TLB outcomes, but found it to add little value to these structures.

3.2.1 Branch Predictors to Detect Faults that Change Control Flow

Branch predictors are capable of predicting the target addresses and directions for branches with high accuracy. The Branch Target Buffer (BTB) provides the target addresses for branches. They are fully tagged with the branch addresses. Thus, if a transient fault ends up changing the control flow, then this can increase the number of BTB misses allowing us to detect the fault.

We implement McFarling's [11] combined predictor to predict the branch directions. It consists of a `gshare` and a `bimodal` predictor, and has a meta predictor to choose between the two. The configuration for the predictor is listed in Table 1. We call this configuration as "BNC" (Branch-No-Confidence), which counts the branch mispredictions for the monitoring interval without using confidence. For BNC, any misprediction is considered an undesirable outcome.

We also consider branch predictors with different levels of confidence. Similar to the JRS predictor [10] we associate a 4-bit saturating counter with each entry in the bimodal predictor and the gshare predictor. The misprediction penalty for the confidence counters is 7, while the bonus for correct prediction is set to 1. We examine two configurations suggested by Grunwald et al. [9]. The first is "Branch Both Strong" (BBS), where both the bimodal and gshare predictor's confidence counters have to be 15 to make a high confidence prediction. The second is "Branch Either Strong" (BES), where only one of the two predictor's confidence counters needs to be 15 to make a high confidence prediction. For BBS and BES we define a *high confidence misprediction* as the undesirable outcome.

For all the branch prediction structures we define a monitoring interval based on the number of branches executed. We then count the number of undesirable outcomes during that interval. We found that BNC, without any confidence counters, performs comparable to BBS and BES. Because, for our fault predictor, accuracy of the predictor is not as important as being able to accurately estimate the number of mispredictions for a monitoring interval.

3.2.2 Detecting Faults Causing Memory Address Errors

We employ two techniques to capture faults that affect the memory addresses. The first technique is based on the principle that if the fault propagates to the address stream, then it is highly probable that the requested memory address does not exist in the cache. Therefore, we can potentially treat a cache miss as a predictor of faulty behavior. This method is attractive especially for the reason that on a cache miss, the processor will have more idle cycles at its disposal [21, 8], which can be utilized for rolling back and replaying execution without severely degrading performance. We monitor L2 cache accesses over a monitoring interval and consider an L2 cache miss as an undesirable outcome. We could have used L1 cache's outcomes, but our predictor's estimate for the L2 cache misses for an interval is more accurate than it is for L1 misses.

To capture faults in the address stream, we also use store sets [5]. Store sets are used for memory disambiguation, and they have been implemented in the Alpha 21264 processor. The purpose of store sets is to predict if a load will be dependent on the stores that were dispatched before the load, even if we do not know the addresses that the prior stores will access. If one can predict that the load is independent of all the stores dispatched before, then it can be executed aggressively out-of-order.

We count the number of loads with incorrect store set predictions within the monitoring interval. An incorrect prediction could mean either that the load was incorrectly predicted to have no dependencies, or that the load was incorrectly predicted to be dependent on a specific store set and it received a forwarded store value. If the number of incorrect store set predictions for loads varies significantly in the current monitoring interval in comparison to the expected value, we predict that there was a transient fault. We define the monitoring interval for the store set as the number of load instructions. The store set predictor that we used (see its configuration in Table 1) can predict the load-store dependency with 99.3% accuracy. We call this transient fault predictor based on the store set as "SSP".

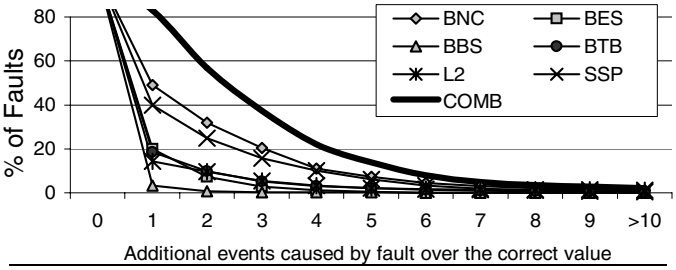

Figure 1. `Number of additional undesirable outcomes observed in the faulty executions.`

3.2.3 Combined Predictor

We examine predicting a transient fault by combining the BNC, BTB, SSP and L2 together. That is, if any of those four predictors predicts a fault, then a fault will be predicted. We call this configuration as "COMB". For a given threshold value, a "COMB" predictor can give rise to more false triggers than any of the other single predictor, but we solve the issue by adaptively increasing the threshold for this predictor as necessary. The technique we use to choose the threshold will be described in Section 4.

3.3 Results

3.3.1 Magnitude of Anomalies

In this section, we first quantify how much the number of undesirable outcomes for a faulty execution *deviates* from the correct execution. For an architectural structure, the deviation for a monitoring interval is the *absolute* difference between the number of undesirable outcomes observed in the faulty execution and the number of undesirable outcomes observed in the corresponding monitoring interval in the correct (golden) execution. For each injected fault, we calculate the maximum deviation for each architectural structure. The maximum deviation is calculated as the maximum of the deviations for all the monitoring intervals that was executed after the fault injection point (we perform our analysis for up-to 10,000 instructions after the injection of the fault). This analysis is used for guiding the selection of prediction thresholds. We want to choose a threshold that it is close to the maximum deviation. To examine the maximum deviation, we use a monitoring interval of length 100. Out of the 1000 faults that we experimented with, masked faults (not important) and faults that cause exceptions (trivially detected) are excluded from our analysis.

Figure 1 shows the maximum deviations for different structures that we considered. The x-axis in Figure 1 shows the magnitude of deviation. The y-axis shows the cumulative percentage of faults. For example, the point 0 along x-axis refers to a deviation of *at-least* 0, and as a result the y-axis value for this point is 100% for all the structures.

Note that higher the deviation, the better we can predict the transient faults, as we can use a higher threshold and still detect transient faults. For the "COMB" configuration, for about 60% of faults we see a difference of *at least* two mispredictions in one of the monitoring intervals executed after the fault injection. The results show that the "BNC" (branch predictor without confidence) can achieve higher coverage as expected, and that the "BTB" result achieve similar coverage to the "BES" technique. The results also show that in using the store set predictor, we have the potential to capture a large number of faults. For example, about 35% of faults encounter deviation of 2 or greater for store set predictions.

"COMB" configuration works well because the predictors and the caches capture faults that are complementary to one another. Branch predictors reflect faults in the control stream. Store set mispredictions and cache misses indicate faults in the address stream. Combining these orthogonal predictors' behavior yields good results.

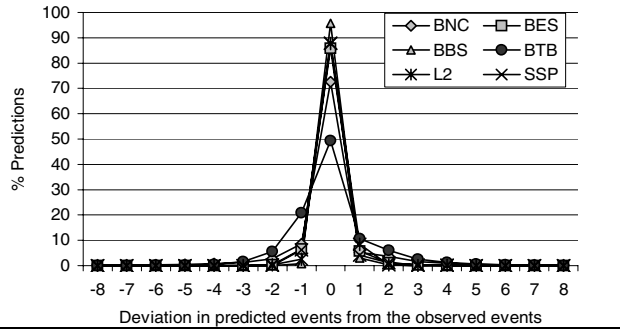

Figure 2. Accuracy of determining the number of undesirable outcomes for a monitoring interval of length 100.

3.3.2 Estimating Undesirable Outcomes

We described a simple predictor to estimate the number of undesirable outcomes for a given monitoring interval in Section 3.1. The accuracy of this predictor is shown in Figure 2.

We determine the accuracy by calculating the difference between the predicted (estimated number of undesirable outcomes) and the actual observed number for each monitoring interval executed in the program. If the difference for an interval is zero, then we have accurately predicted the number of undesirable outcomes for that interval. The differences between the prediction and the actual number are shown along the x-axis in the Figure 2. For each difference, the y-axis shows the percentage of predictions. (There is one prediction for each monitoring interval). Again, for these results we used a monitoring interval of length 100.

We can see that for about 50% of the monitoring intervals, we can accurately predict the number of BTB misses (that is, for 50% of intervals, the difference between the predicted and the actual number of undesirable outcomes is zero). For about 20% of monitoring intervals, we predict one less than the actual number of BTB misses and for about 10% of monitoring intervals, we predict one more than the actual number of BTB misses. Predicting the BTB misses is the toughest as we are able to predict the number of undesirable outcomes for other structures more accurately.

To improve the confidence of our prediction, we can use an appropriate threshold value. It can be observed in Figure 2 that if we use a threshold of 4, then the fault misprediction rate is less than 0.5% for all the structures. If we use a threshold of 2, then the misprediction rate is less than 5% for all the structures except BTB, for which it is about 6%. Earlier, in the Figure 1 we showed that it is possible to capture about 60% of faults with a threshold of 2 and about 20% of faults with a threshold of 4.

An important precaution we take to maintain accuracy is to avoid counting the undesirable outcomes and uses for speculative instructions – that is our transient fault predictor is updated only for committed instructions.

3.3.3 False Triggers and Performance Overhead

Even when the program execution is fault free, it is possible for our predictor to incorrectly predict a transient fault if the estimated number is different (by a threshold) from the actual observed number of undesirable outcomes. This can occur when there are phase or working set changes. Such false positive triggers results in unnecessary rollbacks and re-executions and thereby degrades performance. Next we discuss the performance cost due to these false triggers.

A key variable that controls the false positives is the threshold that we use to predict a transient fault. A higher threshold will result in less number of false triggers but at the same time might fail to detect some transient faults. (We discuss the fault coverage in more detail in the next section). Figure 3 and Figure 4 show the performance overhead of using each of our fault predictors for fixed thresholds of 2 and 4 respectively. In these results, when a transient fault is predicted

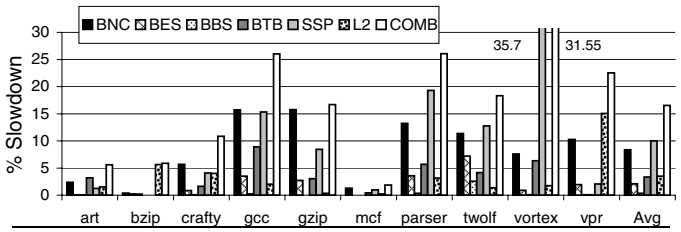

Figure 3. Performance overhead while using predictors with a static threshold of 2 to predict the number of mispredictions.

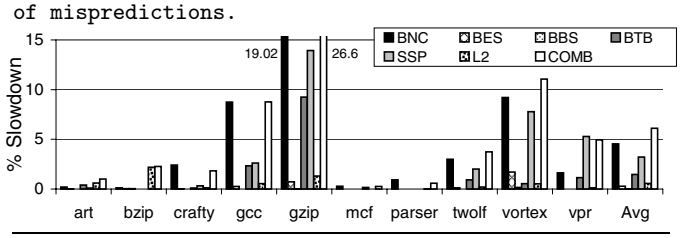

Figure 4. Performance overhead while using predictors with a static threshold of 4 to predict the number of mispredictions.

we rollback and re-execute 1000 instructions using the fault recovery model we described in Section 2.3.

Clearly, using threshold 4 results in less performance degradation than using threshold 2. For example, let us consider the "COMB" configuration. For threshold 4, we experience only 6% performance overhead whereas the overhead is 16% if we use a threshold of 2.

It should be noted that the performance degradation for a given threshold is not uniform across different programs or across different structures. For applications such as `art` and `mcf`, it is desirable to use a lower threshold as the performance impact is not severe even for a lower threshold of 2. However, for programs such as `gzip` and `vortex`, the slowdowns seen are 26% and 12% respectively, even for a threshold of 4. The reason for this is that, for programs with low IPC such as *mcf*, we incur less penalty from rollback and re-execution. In addition, the cycles spent in re-execution overlaps with the idle cycles due to mispredictions or cache misses. Therefore, when a program behaves poorly, there is more opportunity for us to do "cheap" rollback and re-execution.

Since the performance overhead for a given threshold significantly differs across different programs, it motivates the need for using an adaptive threshold based on the number of false triggers observed. In the next section, we discuss a scheme to adaptively modify the threshold value so that the number of false triggers is maintained within a target value irrespective of the variability in the program and the predictor characteristics. This allows us to limit the performance loss.

Finally, the number of false triggers caused by the outcomes of a structure is not directly correlated with the accuracy of that structure. For example, `crafty` is a control intensive application, and it experiences a high number of branch mispredictions. However, in our scheme this does not translate to a high number of false triggers for `crafty` (which can be seen from the low performance overhead that we see for `crafty` in comparison to other programs like `gzip` for the "BNC" configuration). The reason for this is that, even though the number of mispredictions for the structure is high, as long as we can correctly predict the number of mispredictions, we will experience less number of false triggers. The same argument holds for `mcf`, which experiences a high number of cache misses.

4. Adaptive Threshold

In this section, we describe a mechanism to adaptively change the threshold value to meet a target false positive rate. Then we analyze the performance overhead and the percentage of faults detected (that is, fault coverage) using our techniques and compare it with previous low cost solutions [19, 21].

The goal of adaptively changing the threshold is to limit the performance overhead by using a higher threshold when necessary, but still achieve as much coverage as possible by using a lower threshold when the performance is within the tolerable limits. The threshold value of each structure for achieving a low fault misprediction rate varies across the execution of program phases, as well as for different applications. We therefore adapt the threshold value for each predictor during program execution.

The threshold is initialized to 2 for each predictor. Thus, if the number of undesirable outcomes observed in the current interval differs from estimated value by 2 or more, then a fault is predicted. When a fault is predicted, we remember which predictor triggered the fault prediction. We then rollback and re-execute. In Section 2.3, we described how we can determine if a fault prediction is a false trigger or not. For each predictor (BNC, BES, BBS, BTB, L2, SSP), we keep track of the number of fault triggers for the last N thousand instructions executed. If the percentage of false trigger (hereafter referred to as false trigger rate) goes above X% for a given predictor, then we increase the predictor's threshold by 1. We examined a number of values for this target false trigger rate, but here we report results only for 0.01% and 0.001% due to space constraints. Using a target rate of 0.01% means that the threshold would be increased by 1 if a structure has more than 1 false trigger per 10,000 instructions executed. Using target rate of 0.001% means that the threshold is increased until there is no more than 1 false trigger every 100,000 instructions executed.

For a particular transient fault predictor, we adapt the threshold by increasing it by 1, whenever the target false trigger rate is not met. We check the false trigger rate and update the threshold at the end of each monitoring interval. The execution may become more predictable after we have increased the threshold. Hence, we periodically (after every 10 million instructions) reset the threshold to a base value of 2.

For the combined predictor (COMB), we do not want to equally penalize every predictor that it uses, because some of them might be predicting accurately for the current threshold value. Instead, we independently keep track of the false trigger rate for each predictor used by the combined predictor. We penalize only the predictor with the highest false trigger rate by increasing its threshold by 1 until all of the structures' false trigger rates are below the target rate.

4.1 Coverage and Performance Analysis

Figure 5 shows the performance overhead and Figure 6 shows the coverage for the various transient fault predictor configurations we examined. All results for our technique presented in this section use the "COMB" configuration, which is our best performing predictor.

We show results for two prior schemes. Weaver et al [21], proposed to flush the pipeline on an L2 miss to avoid entries in the queues getting affected by a transient fault. We present results, labeled as $Prior - L2$, which correspond to re-executing a certain number of instructions (we analyze various rollback window lengths) on an L2 miss.

ReStore [19] proposed using branch mispredictions as symptoms of transient faults. In their scheme, they initiate rollback and re-execution, whenever there is a high confidence misprediction. The confidence scheme they use is similar to our "BES" configuration. We call this result as $Prior - JRS$ and this constitutes the baseline comparison to prior work. Note that, unlike Restore, in our work, we study only the silent data corruption faults. We exclude the faults that are masked (harmless) and the ones that cause exceptions (trivial to detect and recover) from our analysis. Also, we do not consider faults that affect the cache subsystem, which can be protected using ECC. Hence the absolute percentages reported here are lower than in Restore.

In addition to the above baseline results, we show results for our proposed techniques. We provide results for using a static threshold of 2 and 4. We also show results for adaptively changing the threshold with a false trigger rate target of 0.01% and 0.001%.

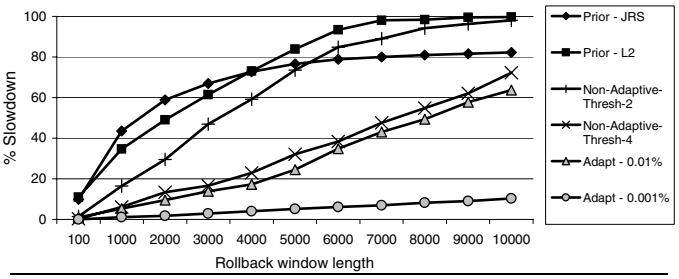

Figure 5. Performance overhead while using prior techniques in comparison to interval-based anomaly prediction.

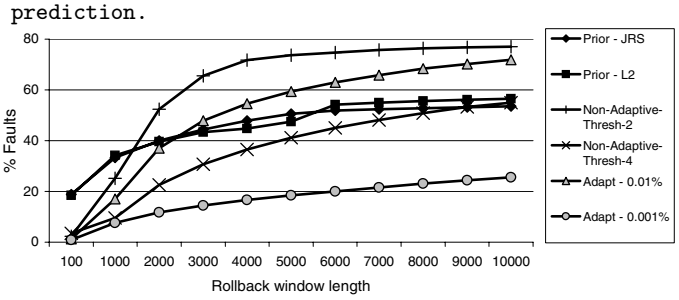

Figure 6. Fault coverage obtained using prior techniques in comparison to interval-based anomaly prediction. Normalized to all the faults (including masked and faults causing exception.

When a fault is predicted we rollback to the last checkpoint and re-execute the program execution from thereon. Fault coverage also depends on how far back we rollback, which we refer to as the rollback window length. In Figures 5 and 6 we vary the rollback window length on the x-axis. Figure 5 shows the performance overhead. The performance overhead is caused by false triggers. Figure 6 shows the percentage of *silent data corruption* faults captured by various detection techniques for different rollback window lengths.

The results show that while $Prior - L2$ and $Prior - JRS$ achieve fault coverage comparable to our techniques, the performance overhead is severe. For example, $Prior - L2$ with a rollback window length of 1000 achieves 34% coverage but incurs 35% performance overhead. However, in order to achieve the equivalent coverage (37%), we need to incur only about 9% overhead using our adaptive technique with a target rate of 0.01% (using a rollback window of length 2000).

Changing the threshold dynamically reduces the performance overhead significantly. For example, for a rollback window length 2000, our non-adaptive scheme using threshold 4, achieves 22% fault coverage at a cost of 13% performance overhead. However, for a smaller overhead cost of 9.5%, our adaptive strategy with 0.01% target achieves 37% fault coverage (for the same rollback window length 2000).

5. Related Work

A low cost partial fault tolerant mechanism is desirable over techniques that advocate full scale temporal [7, 18, 3, 17] or structural redundancy [15, 13, 14] for the reasons that we described in Section 1. In this section, we place our work in context with previous works that proposed low cost solutions.

5.1 Fault Tolerance with Low Overhead

Gomaa et al. [8] proposed opportunistic fault tolerance where a program is redundantly executed in one of the contexts in an SMT processor only when the main thread is in a low IPC phase, such as when the main thread is stalled due to a L2 cache miss. This approach incurs low performance overhead. However, it assumes that a context in a SMT is available for redundant execution. To improve the fault coverage, they also proposed using an additional hardware structure

called the Instruction Reuse Buffer (IRB), which caches the input and output values of previously executed instructions. Using IRB, one can determine an output value for an instruction if its input values match with any of the entries in the IRB. A similar technique was also proposed by Parashar et al. [12]. However, IRB adds additional complexity to the processor. Unlike these schemes, our proposed technique neither assumes multi-threading support nor any additional structures like IRB, but leverages the existing hardware structures to predict and recover from transient faults.

Weaver et al. [21] observed that the instructions stalling in the issue queue during an L2 cache miss are more vulnerable to single event upsets. Therefore, they proposed flushing the instructions in the pipeline upon encountering a L2 cache miss. This technique is simple and has very low performance overhead. However, it protects only the issue queue of a processor, and thus offers low fault coverage.

To our knowledge ReStore [19] is the first work that proposed predicting transient faults based on mispredictions in speculative structures. In addition to detecting faults that cause exceptions and protecting caches and registers using ECC, ReStore predicts a transient fault when there is a high confidence branch misprediction. This is similar to our approach in that ReStore utilizes an existing structure in the processor, the branch predictor, to predict faults. We improve upon this work in a number of ways. We show that predicting a transient fault for *every* high confidence misprediction leads to a high number of false triggers. This results in performance overhead due to unnecessary rollbacks and re-executions. To overcome the problem, we propose looking at the misprediction rate or cache miss rate over an interval of execution to predict a fault, and also adapt the predictor's parameters to the behavior of the program. In addition, unlike Restore, we use store set [5] mispredictions to aid transient fault detection. We also demonstrate that combining the outcomes of various complimentary structures in the processor is useful in getting higher coverage for a given performance target.

6. Conclusion

In this paper, we introduced an architectural technique to achieve partial fault tolerance with low performance overhead and minimal modifications to hardware. Following the hypothesis that during faulty execution, the control flow and address computations diverge from the correct execution, our technique monitors the mispredictions in branch and store set predictors, and the cache misses. We then use the execution history to predict the number of undesirable outcomes for the current monitoring interval. Whenever the mispredictions significantly deviate from our estimate by more than a threshold value, we predict a fault and trigger recovery. We recover from faults through rollback and replay.

We proposed an effective way to combine the fault symptoms we observe in different architectural units. We presented an adaptive approach for determining the best threshold value for predicting a fault for each structure to reduce the false trigger rate and at the same time achieve as much coverage as possible. We showed that for a performance cost of less than 9.5% we can detect and recover from 37% of silent data corruption faults, and for less than 24% slowdown we can achieve about 60% fault coverage rate.

Acknowledgments

We would like to thank the anonymous reviewers for providing valuable feedback on this paper. This work was funded in part by NSF grant CNS-0509546, and grants from Intel and Microsoft.

References

[1] H. Akkary, R. Rajwar, and S. T. Srinivasan. An analysis of a resource efficient checkpoint architecture. *TACO 04: ACM Transactions on Architecture and Code Optimization*, 1(4):418–444, December 2004.

[2] Anonymous. HP integrity nonstop computing. http://h20223.www2.hp.com/nonstopcomputing/cache/76385-0-0-0-121.html.

[3] T. M. Austin. DIVA: a reliable substrate for deep submicron microarchitecture design. In *MICRO 32: Proceedings of the 32nd annual ACM/IEEE international symposium on Microarchitecture*, pages 196–207, 1999.

[4] D. C. Burger and T. M. Austin. The simplescalar tool set, version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, Madison, June 1997.

[5] G. Chrysos and J. Emer. Memory dependence prediction using store sets. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[6] M. J. Flynn, P. Hung, and K. W. Rudd. Deep-submicron microprocessor design issues. *IEEE Micro*, 19(4):11–22, 1999.

[7] M. Gomaa, C. Scarbrough, T. N. Vijaykumar, and I. Pomeranz. Transient-fault recovery for chip multiprocessors. In *ISCA'03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 98–109, 2003.

[8] M. A. Gomaa and T. N. Vijaykumar. Opportunistic transient-fault detection. In *ISCA'05: Proceedings of the 32nd annual international symposium on Computer architecture*, pages 172–183, 2005.

[9] D. Grunwald, A. Klauser, S. Manne, and A. Pleskun. Confidence estimation for speculation control. In *25th Annual International Symposium on Computer Architecture*, June 1998.

[10] E. Jacobsen, E. Rotenberg, and J. Smith. Assigning confidence to conditional branch predictions. In *29th International Symposium on Microarchitecture*, Dec. 1996.

[11] S. McFarling. Combining branch predictors. Technical Report TN-36, Digital Equipment Corporation, Western Research Lab, June 1993.

[12] A. Parashar, S. Gurumurthi, and A. Sivasubramaniam. A complexity-effective approach to alu bandwidth enhancement for instruction-level temporal redundancy. In *ISCA'04: Proceedings of the 31st annual international symposium on Computer architecture*, page 376, 2004.

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

[14] S. K. Reinhardt and S. S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA'00: Proceedings of the 27th annual international symposium on Computer architecture*, pages 25–36, 2000.

[15] G. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the Third International Symposium on Code Generation and Optimization*, March 2005.

[16] T. Sherwood, E. Perelman, G. Hammerley, and B. Calder. Automatically characterizing large-scale program behavior. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[17] T. J. Slegel, R. M. A. III, M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, 1999.

[18] J. Smolens, B. Gold, J. Kim, B. Falsafi, J. Hoe, and A. Nowatzyk. Fingerprinting: bounding soft-error detection latency and bandwidth. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 224–234, 2004.

[19] N. J. Wang and S. J. Patel. ReStore: Symptom based soft error detection in microprocessors. In *DSN*, pages 30–39, 2005.

[20] 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 04: Proceedings of the 2004 International Conference on Dependable Systems and Networks*, page 61, 2004.

[21] 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'04: Proceedings of the 31st annual international symposium on Computer architecture*, page 264, 2004.