Appears in *Proceedings of the 3rd Workshop on Dependable Architectures*, Lake Como, Italy. Nov. 2008.

Exploiting Value Prediction for Fault Tolerance

Xuanhua Li and Donald Yeung
Department of Electrical and Computer Engineering
Institute for Advanced Computer Studies
University of Maryland at College Park

Abstract

Technology scaling has led to growing concerns about reliability in microprocessors. Currently, fault tolerance techniques rely on explicit redundant execution for fault detection or recovery which incurs significant performance, power, or hardware overhead. This paper makes the observation that value predictability is a low-cost (albeit imperfect) form of program redundancy that can be exploited for fault tolerance. We propose to use the output of a value predictor to check the correctness of predicted instructions, and to treat any mismatch as an indicator that a fault has potentially occurred. On a mismatch, we trigger recovery using the same hardware mechanisms provided for mispeculation recovery. To reduce false positives that occur due to value mispredictions, we limit the number of instructions that are checked in two ways. First, we characterize fault vulnerability at the instruction level, and only apply value prediction to instructions that are highly susceptible to faults. Second, we use confidence estimation to quantify the predictability of instruction results, and apply value prediction accordingly. In particular, results from instructions with higher fault vulnerability are predicted even if they exhibit lower confidence, while results from instructions with lower fault vulnerability are predicted only if they exhibit higher confidence. Our experimental results show such selective prediction significantly improves reliability without incurring large performance degradation.

1 Introduction

Soft errors are intermittent faults caused by cosmic particle strikes and radiation from packaging materials. They do not cause permanent damage, but still corrupt normal program execution. Technology scaling combined with lower supply voltages make systems more vulnerable to soft errors. Hence, soft errors have become an increasingly important design consideration with each successive generation of CPUs.

To enhance system reliability, existing techniques typically introduce redundant execution–by taking either a hardware or software approach–to detect or recover from faults. On the hardware side, error-detection circuitry (ECC or parity bits) can be added to storage structures. Other hardware techniques utilize additional structures such as extra processor cores, hardware contexts, or functional units [1, 2, 3, 4, 5] to execute redundantly in order to compare results and detect faults. In contrast to hardware techniques, software-based techniques rely on the compiler to duplicate program code [6, 7, 8]. This software redundancy also permits comparison of results at runtime, but without any additional hardware cost. While differing in implementation, both hardware and software approaches create explicit redundancy to provide fault tolerance which incurs significant performance, power, or hardware overhead.

Prior studies have shown that program execution itself contains a high degree of redundancy–*i.e.*, instruction and data streams exhibit repeatability. One example of exploiting such inherent redundancy is value prediction which predicts instruction results through observation of past values. By predicting values before they are executed, data dependency chains can be broken, permitting higher performance. Unfortunately, value prediction has had limited success in commercial CPUs due to its relatively low prediction accuracy and high misprediction penalty, the latter becoming increasingly severe with deeper processor pipelines.

In this work, we employ value prediction to improve system reliability. Compared to explicit redundant execution techniques, the advantage of value prediction is it exploits programs' inherent redundancy, thus avoiding the cost of explicitly duplicating hardware or program code as well as the associated area, power, and performance overheads. Although a value predictor itself incurs some additional hardware, we find a relatively small predictor can effectively detect faults; hence, our approach incurs less hardware than traditional explicit duplication techniques.

In addition to exploiting inherent program redundancy, another advantage of our approach is it is less sensitive to the negative effects of misprediction. Mispredictions are always undesirable from the standpoint of performance since they require flushing the pipeline. Because traditional uses of value prediction are focused on improving performance, such flushes undermine their bottom line. However, in the context of fault detection/recovery, flushes can be desirable because they reduce the time that program instructions (particularly those that are stalled) spend in the pipeline, thus improving architectural vulnerability. Rather than always being undesirable, for our technique, mispredictions represent a tradeoff between performance and reliability. Lastly, compared to traditional uses of value prediction, our technique does not require as fast value predictors. For performance-driven techniques, value predictions are needed early in the pipeline. In contrast, for fault detection/recovery, value predictions can be delayed until the writeback stage, where value checking occurs.

To maximize the efficacy of our technique, we focus value prediction only on those instructions that receive the greatest benefit. In particular, we characterize fault vulnerability at the instruction level, and apply value prediction only to those instructions that are most susceptible to faults.[1] An instruction's fault vulnerability in a specific hardware structure is quantified by measuring the fraction of the structure's total AVF (Architectural Vulnerability Factor) that the instruction accounts for. Our results show a small portion of instructions account for a large fraction of system vulnerability. For example, for the fetch buffer in our processor model, about 3.5% of all instructions are responsible for 53.9% of the fetch buffer's total AVF in the TWOLF benchmark. This suggests that selectively protecting a small number of instructions can greatly enhance the overall reliability. Because we apply value prediction only on a small number of instructions, the potential performance loss due to mispredictions is also quite small.

[1]Identifying the most vulnerable instructions occurs late in the pipeline. Thus for implementation with more advanced but slower value predictor, all result-producing instructions are eligible for prediction once they enter the pipeline, but only those that are later identified as the most susceptible to faults will have their results checked and update the predictor.

To further reduce the impact of mispredictions, we use an adaptive confidence estimation technique to assess the predictability of instructions, and apply prediction accordingly. Our approach is adaptive because it applies prediction more or less aggressively depending on each instruction's fault vulnerability (which can be quantified through its latency). Instructions with high fault vulnerability are predicted even if they exhibit low confidence, while instructions with low fault vulnerability are predicted only if they exhibit high confidence. Our results show that this technique achieves significant improvements in reliability without sacrificing much on performance.

The rest of the paper is organized as follows. Section 2 introduces how we apply value prediction for fault detection. We mainly discuss our study on characterizing instructions' vulnerability to faults, as well as our methods for selecting instructions for fault protection. Then, Section 3 describes our experimental methodology, and reports on the reliability and performance results we achieve. Finally, Section 4 presents related work, and Section 5 concludes the paper.

2 Reducing Error Rate with Value Prediction

This section describes how value prediction can be used to reduce error rate. First, Section 2.1 discusses how we use value predictors to check instruction results. Then, Section 2.2 briefly describes fault recovery. Finally, Section 2.3 quantifies instructions' vulnerability to faults, and proposes selectively predicting instructions to mitigate performance loss.

2.1 Predictor-Based Fault Detection

To identify potential faults, we use a value predictor to predict instruction outputs. We employ a hybrid predictor composed of one stride predictor and one context predictor [9]. Prediction from the context predictor is attempted first. If the context predictor cannot make a prediction (see Section 3.1), then the stride predictor is used instead to produce a result. After a prediction is made, the result is compared with the actual computation result. The comparison is performed during the instruction's writeback stage, so the predictor's output is not needed until late in the pipeline. Since prediction can be initiated as soon as the instruction is fetched, there is significant time for the predictor to make its prediction, as mentioned in Section 1.

During each predictor comparison, the prediction and actual instruction result will either match or differ. If they match, two interpretations are possible. First, the predictor predicted the correct value. In this case, no fault occurred since the instruction also produced the same correct value. Second, the predictor predicted the wrong value, but a fault occurred such that the instruction produced the same wrong value. This case is highly unlikely, and for all practical purposes, will never happen. Hence, on a match, we assume no fault has occurred, and thus, no additional action is required.

Another possibility is the prediction and actual instruction result differ. Again, two interpretations are possible. First, the predictor predicted the correct value. In this case, a fault has occurred since the instruction produced a different value. Second, the predictor predicted the wrong value, and the instruction either produced a correct or wrong value (again, we assume a misprediction and incorrect result will never match). Unfortunately, there is no way to tell which of these has occurred, so at best on a mismatch, we can only assume that there is the *potential* for a fault. We always assume conservatively that a fault has occurred, and initiate recovery by squashing the pipeline and re-executing the squashed instructions in the hopes of correcting the fault. During re-execution, if the instruction produces the same result, then with high probability the original instruction did not incur a fault.[2] If no fault occurred (the most likely case), the pipeline flush was unnecessary, and performance is degraded. (However, as we will see in Section 2.2, such "unnecessary" flushes can actually improve reliability in many cases).

To mitigate the performance degradation caused by false positives, we use confidence estimation. In particular, we employ the confidence estimator described in [10]. We associate a saturating counter with each entry in the value predictor table. A prediction is made only when the corresponding saturating counter is equal to or above a certain threshold. If the prediction turns out to be correct (the match case), the saturating counter is incremented by some value. If the prediction turns out to be incorrect (the mismatch case in which the original and re-executed results are the same), the saturating counter is decremented by some value. Given confidence estimation, we can tradeoff the number of false positives with the number of predicted instructions (and hence, the fault coverage) by varying the confidence threshold. Section 3 will discuss how we select confidence thresholds.

[2]The comparison of a re-executed result with the originally executed result is not necessary on-line for our technique to work properly. In fact, our technique never knows whether a mismatch was caused by a misprediction or an actual fault. The main issue with re-execution is predictor updates, which is discussed in Section 3.1.

2.2 Fault Recovery

When an instruction's prediction differs from its computed value, it is possible a fault occurred before or during the instruction's execution. To recover from the fault, it is necessary to roll back the computation prior to the fault, and re-execute. In our work, we perform roll back simply by flushing from the pipeline the instruction with the mismatch as well as all subsequent instructions. Then, we refetch and re-execute from the flush point. (A similar mechanism for branch misprediction recovery can be used for our technique).

Notice, our simple approach can only recover faults that attack predicted instructions, or instructions that are downstream from a mispredicted instruction (which would flush not only the mispredicted instruction, but also all subsequent instructions). If a fault attacks a non-predicted instruction that is not flushed by an earlier mispredicted instruction, then even if the fault propagates to a predicted instruction later on, recovery would not roll back the computation early enough to re-execute the faulty instruction. However, even with this limitation, we find our technique is still quite effective.

Because soft errors are rare, most recoveries are triggered by the mispredictions of the value predictor. As mentioned in Section 2.1, such false positives can degrade performance. However, they can also improve reliability. Often times, re-executed instructions run faster than the original instructions that were flushed (the flushed instructions can prefetch data from memory or train the branch predictor on behalf of the re-executed instructions). As a result, the re-executed instructions occupy the instruction queues for a shorter amount of time, reducing their vulnerability to soft errors compared to the original instructions. This effect is particularly pronounced for instructions that stall for long periods of time due to cache misses. Hence, while false positives due to mispredictions can degrade performance, this degradation often provides a reliability benefit in return. The next section describes how we can best exploit this tradeoff.

2.3 Instruction Vulnerability

In order to reduce the chance of mispredictions and unnecessary squashes, we not only apply confidence estimation (as described in Section 2.1), but we also limit value prediction to those instructions that contribute the *most* to overall program reliability. This section describes how we assess the reliability impact of different instructions.

Recently, many computer architects have used Architectural Vulnerability Factor (AVF) to reason about hardware reliability [11]. AVF captures the probability that a transient fault in a processor structure will result in a visible error at a program's final outputs. It provides a quantitative way to estimate the architectural effect of fault derating. To compute AVF, bits in a hardware structure are classified as critical for architecturally correct execution (ACE bits), or not critical for architecturally correct execution (un-ACE bits). Only errors in ACE bits can result in erroneous outputs. A hardware structure's AVF is the percentage of ACE bits that occupy the hardware structure on average.

To identify ACE bits, instructions themselves must first be distinguished as ACE or un-ACE. We make the key observation that not all ACE instructions contribute equally to system reliability. Instead, each ACE instruction's occupancy in hardware structures determines its reliability contribution. As observed by Weaver *et al.* [12], the longer instructions spend in the pipeline, the more they are exposed to particle strikes, and hence, the more susceptible they become to soft errors. Weaver *et al.* proposed squashing instructions that incur long delays (*e.g.*, L2 cache misses) to minimize the occupancy of ACE instructions. We extend this idea by quantifying fault vulnerability at the instruction level, and selectively protecting the instructions that are most susceptible to faults.

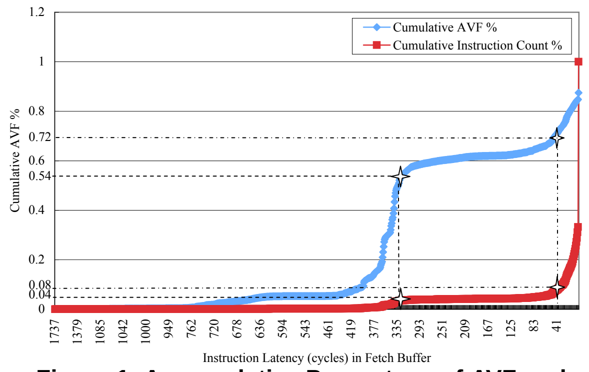

Figure 1. Accumulative Percentage of AVF and Instruction Count in Fetch Buffer on TWOLF.

Our approach is particularly effective because we find a very small number of instructions account for a majority of the AVF in hardware structures. Figure 1 illustrates this for the processor's fetch buffer when executing TWOLF, a SPEC2000 benchmark. In Figure 1, the top curve plots the cumulative fraction of overall AVF (y-axis) incurred by instructions that occupy the fetch buffer for different latencies (x-axis) sorted from highest latency to lowest latency. The bottom curve plots the cumulative fraction of dynamic instructions (y-axis) that experience the given latencies. In total, there are 1,944 static instructions that have been simulated. As Figure 1 shows (the two datapoints marked on the left of the graph), 53.9% of the fetch buffer's AVF is incurred in 3.5% of all dynamic instructions. These instructions have large latencies–300 cycles or more. As indicated by the other two datapoints marked on the right side of the graph, the majority of instructions (about 91.8%) exhibit a latency smaller than 40 cycles, and account for a relatively small portion of the overall AVF (about 28.4%). We find similar behavior occurs for the other benchmarks as well as for the other hardware structures. Such results show that using our value predictor to target a small number of instructions–those with very large latencies–is sufficient to provide the majority of fault protection. This is good news since it will minimize the performance impact of mispredictions.

In our study, we find that even though an instruction may stall for a long time in one hardware structure, it may not stall for very long in other structures. In other words, a single instruction can contribute differently to different structures' vulnerability. Thus, an important question is how can we select the smallest group of instructions that will provide the largest benefit to reliability? In our work, we measure the latency an instruction incurs from the fetch stage to the issue stage, and use this to determine each instructions' contribution to reliability, applying value prediction only to those instructions that meet some minimum latency threshold. Because our approach accounts for "front-end" pipeline latency, we directly quantify the occupancy of instructions in the fetch and issue queues, and hence, are able to identify the instructions that contribute the most to reliability in these 2 hardware structures. This is appropriate for our work since later on (in Section 3) we study our technique's impact on both fetch and issue queue reliability (we also study the impact on the physical register file's reliability, though our latency metric does not directly quantify result occupancy in this structure). If improving reliability in other hardware structures

Processor Parameters	
Bandwidth	8-Fetch, 8-Issue, 8-Commit
Queue size	64-IFQ, 40-Int IQ, 30-FP IQ, 128-LSQ
Rename reg/ROB	128-Int, 128-FP / 256 entry
Functional unit	8-Int Add, 4-Int Mul/Div, 4-Mem Port 4-FP Add, 2-FP Mul/Div
Branch Predictor Parameters	
Branch predictor	Hybrid 8192-entry gshare/2048-entry Bimod
Meta table	8192 entries
BTB/RAS	2048 4-way / 64
Memory Parameters	
IL1 config	64kbyte, 64byte block, 2 way, 1 cycle lat
DL1 config	64kbyte, 64byte block, 2 way, 1 cycle lat
UL2 config	1Mbyte, 64byte block, 4 way, 20 cycle lat
Mem config	300 cycle first chunk, 6 cycle inter chunk
Hybrid Value Predictor Parameters	
VHT size	1024
value history depth	4
PHT size	1024
PHT counter thresh	3

Table 1. Parameter settings for the detailed architectural model used in our experiments.

is desired, it may be necessary to use a different latency metric. This is an important direction for future work.

3 Experimental Evaluation

In Section 2, we showed a small number of instructions account for a large portion of hardware vulnerability. We also qualitatively analyzed the impact of pipeline flushes: flushing degrades performance, but in some cases may improve program reliability. We consider both findings in our design, and use insights from both to drive confidence estimation (which ultimately determines which instructions will be predicted).

This section studies these issues in detail. First, we present the simulator and benchmarks used throughout our experiments (Section 3.1). Then, we present our experiments on applying value prediction without confidence estimation. (Section 3.2). The goal of these experiments is to show that we can limit performance degradation by focusing the value predictor on the portion of instructions that impact system reliability the most. Finally, we add confidence estimation, and show the improvements this can provide (Section 3.3).

3.1 Simulator and Benchmarks

Throughout our experiments, we use a modified version of the out-of-order processor model from Simplescalar 3.0 for the PISA instruction set [13], configured with the simulator settings listed in Table 1. Our simulator models an out-of-order pipeline consisting of fetch, dispatch, issue, execute, writeback, and commit pipeline stages. Compared to the original, our modified simulator models rename registers and issue queues separately from the Register Update Unit (RUU). We also model a hybrid value predictor that includes a single stride predictor and a single context predictor, as described in [9]. The value predictor configuration is shown in Table 1.

Our stride predictor contains a Value History Table (VHT). For each executed instruction, the VHT maintains a last-value field (which stores the instruction's last produced value) and a stride field. When a new instance of the instruction is executed, the difference between the new value and the last-value field is written into the stride field, and the new value itself is written into the last-value field. If the same stride value is computed twice in a row, the predictor predicts the instruction's next value as the sum of the last-value and stride fields. When a computed stride differs from the previously computed stride, the predictor stops making predictions until the stride repeats again.

Our context predictor is a 2-level value predictor consisting of a VHT and a Pattern History Table (PHT). For each executed instruction, the VHT maintains the last history-depth number of unique outcomes produced by the instruction (we employ a history depth = 4). In addition, the VHT also maintains a bit field that encodes the pattern in which these outcomes occurred during the last pattern-length dynamic instances of the instruction (we employ a pattern length = 4). During prediction, the instruction's bit field is used to index the PHT. Each PHT entry contains several frequency counters, one for each instruction outcome in the VHT. The counter with the highest count indicates the most frequent successor value given the instruction's current value pattern. If this maximum count is above some threshold (we employ a threshold = 3), then the corresponding outcome is predicted for the instruction; otherwise, no prediction is made. After an instruction executes and its actual outcome is known, the corresponding PHT entry counter is incremented by 3 while the other counters from the same PHT entry are decremented by 1. Lastly, the corresponding bit field in the VHT is updated to reflect the instruction's new outcome pattern.

For some of our experiments (*e.g.*, Section 3.3), we employ confidence estimation along with value prediction. As discussed in Section 2.1, we use the confidence estimator described in [10] which associates a 4-bit saturating counter with each PHT entry. Update to all predictor structures (stride, context, and confidence estimator) only occurs on predicted instructions. In our technique, many instructions are not predicted because their latencies are short, making them less important to overall reliability. These non-predicted instructions do not update the predictor structures. During re-execution after a misprediction, the CPU will likely re-execute the mispredicted instruction, and the predictor may predict again.[3] In this case, the predictor is very likely to generate a correct prediction due to training from the misprediction. In any case, we still update the predictor after the prediction (*i.e.*, predictor updates do not distinguish between the first execution of some instruction and its re-execution after a misprediction).

In terms of timing, our simulator assumes the stride and context predictors can always produce a prediction by each instruction's writeback stage. We believe this is reasonable given the small size of our predictor structures in Table 1. In particular, our predictors are either smaller than or equal to the value predictors found in the existing literature for performance enhancement [10, 14, 9]. Since our technique is not as timing critical (conventional value predictors must make predictions by the issue stage), we believe there will not be any timing-related problems–both in terms of latency and bandwidth–when integrating our predictors into existing CPU pipelines. On a misprediction, our simulator faithfully models the timing of the subsequent pipeline flush as well as the cycles needed to re-fetch and re-execute the flushed instructions. Our simulator also assumes a 3-cycle penalty from when a misprediction is detected until the first re-fetched instruction can enter the pipeline.

Table 2 lists all the benchmarks used in our experiments. In total, we employ 9 programs from the SPEC2000 benchmark suite. All of our benchmarks are from the integer portion of the suite; we did not study floating-point benchmarks since our value predictors only predict integer outcomes. In Table 2, the column labeled "Input" specifies the input dataset used for each benchmark, and the column labeled "Instr Count" reports the number of instructions executed by each benchmark. The last column, labeled "IPC," reports each benchmark's average IPC without value prediction. The latter represents baseline performance from which the IPC impact of our technique is computed.

Finally, throughout our experiments, we report

[3]With confidence estimation, this will not happen because the original misprediction would lower the confidence value for the re-executed instruction enough to suppress prediction the second time around. But without confidence estimation, prediction during re-execution can happen.

Benchmark	Input	Instr Count	IPC
300.twolf	ref	109546670	0.79
176.gcc	166.i	240000000	1.42
254.gap	train.in	411061781	1.65
164.gzip	input.compressed	192015257	2.06
256.bzip2	input.compressed	2346534735	3.20
253.perlbmk	diffmail.pl	1000000000	1.57
197.parser	ref.in	1404572471	1.32
181.mcf	inp.in	500000000	0.13
175.vpr	test	1512992144	1.87

Table 2. Benchmarks and input datasets used in our experiments. The last two columns report instructions executed and baseline IPC for each benchmark.

both performance and reliability to investigate their tradeoff. In particular, we measure IPC for performance and AVF for reliability. We analyze reliability for three hardware structures only–the fetch queue, issue queue, and physical register file. Since we use value prediction to perform fault checking on architectural state at writeback, we can detect faults that attack most hardware structures in the CPU, including functional units, the reorder buffer, etc. But our results do not quantify the added protection afforded to structures outside of the three we analyze. Furthermore, we do not analyze reliability for the value predictors themselves. Predictors do not contain ACE bits; however, soft errors that attack the value predictors could cause additional mispredictions and flushes that can impact both performance and reliability. Again, our results do not quantify these effects. Lastly, our technique incurs additional power consumption in the value predictor tables. Since we do not model power, our results do not quantify these effects. However, we believe the power impact will be small given the small size of our predictors. Furthermore, given their relaxed timing requirements, there is room for voltage scaling optimizations to minimize the power impact.

3.2 Value Prediction Experiments

We first present our experiments on applying value prediction without confidence estimation. We evaluate the impact on both reliability and performance when predicting all or a portion of the result-producing instructions. We call these full and selective prediction, respectively. For selective prediction, we predict instructions based on their latency measured from the fetch stage to the issue stage. Since we do not know if an instruction should be predicted when we fetch it, we initiate prediction for all result-producing instructions upon fetch, but only perform fault checking and predictor updates for those instructions that meet the latency thresholds when they arrive at the writeback stage.

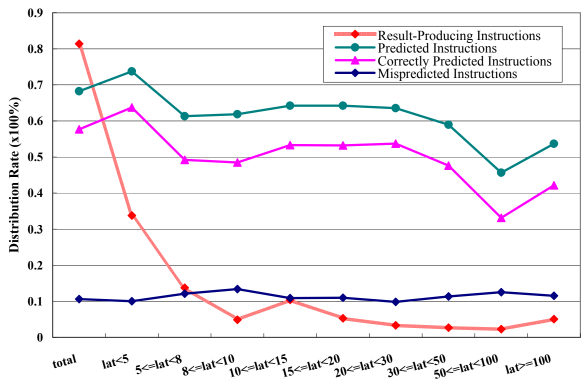

Figure 2. Fraction of instructions that are result-producing as well as the fraction of predicted, correctly predicted, and mispredicted instructions across different latency ranges over all 9 spec2000 integer benchmarks.

Figure 2 reports the number of result-producing instructions, first across all instructions (labeled "total") and then for different latency ranges, as a fraction of all executed instructions. The figure also reports the fraction of instructions from each category ("total" and the different latency ranges) that are predicted, predicted correctly, and mispredicted. (Note, the fraction of predicted instructions–and hence, the sum of the fraction of correctly predicted and mispredicted instructions–is not 1.0 because the predictor is unable to make predictions for some instructions, as described in Section 3.1). Every datapoint in Figure 2 represents an average across all the benchmarks listed in Table 2. Figure 2 shows result-producing instructions account for 81.4% of all instructions. In particular, instructions with a latency less than 5 cycles (from fetch to issue) account for 32.1% of all instructions, or 41.4% of result-producing instructions. Moreover, these short-latency instructions exhibit relatively good prediction rates–63.7% on average. In contrast, instructions with greater than 5-cycle latency have slightly lower prediction rates–around 40% to 50%. However, given that long-latency instructions contribute the most to fault vulnerability, it is still worthwhile to check their values via prediction.

As our results will show, the mispredictions in Figure 2 (which represent false positives in our technique) lead to performance degradation because they initiate pipeline squashes. Table 3 reports the number of such performance-degrading mispredictions for all our benchmarks. In particular, the rows in Table 3 numbered 1 through 4 report mispredictions for the "Total," "lat<5," "15<=lat<20," and "lat>=100" datapoints, respectively, from Figure 2. As Table 3 shows, the number of mispredictions, and hence the amount of performance degradation, generally reduces for selective prediction of longer latency instructions.

Next, we study the impact of value prediction on actual program reliability and performance. Figure 3 reports the percent AVF reduction (*i.e.*, reliability improvement) with value prediction in three hardware structures compared to no value prediction averaged across our 9 SPEC2000 integer benchmarks. In particular, the curves labeled "issue queue," "fetch buffer," and "physical register file" report the AVF reductions for the issue queue, fetch buffer, and physical register file, respectively. Also, the datapoints labeled "pred all" report the AVF reduction assuming full prediction, while the remaining datapoints report the AVF reduction with selective prediction based on instruction latency (*e.g.*, "pred lat $\geq$ 15" performs prediction only for instructions with at least 15-cycle latency between the fetch and issue stages).

Figure 3 shows prediction-based fault protection can be very effective at improving reliability (*i.e.*, reducing AVF). The AVF for the fetch queue, issue queue, and register file is reduced by as much as 96.0%, 89.8%, and 59.0%, respectively (under full prediction) compared to no prediction. This is due to both correct and incorrect predictions. On a correct prediction, the value of the predicted instruction is checked, so the instruction is no longer vulnerable, and hence, does not contribute to the AVF of the structures it occupies. On a misprediction, the pipeline is flushed. As discussed in Section 2.2, re-execution after flushing is typically faster than the original execution, thus reducing the occupancy of ACE instructions in the hardware structures. Both combine to provide the AVF improvements shown in Figure 3.

Unfortunately, these reliability improvements come at the expense of performance. In Figure 3, the curve labeled "IPC" reports the percent IPC reduction (*i.e.*, performance degradation) for the same experiments. This curve shows IPC can degrade significantly due to the penalty incurred by mispredictions, particularly when a large number of instructions are predicted. Under full prediction, IPC reduces by 55.1% compared to no prediction. But the performance impact lessens as fewer instructions are predicted (moving towards the right

	twolf	gcc	gap	gzip	bzip2	perl	parser	mcf	vpr
1.	12283990	32105002	43971177	6637540	70107090	69058496	130089349	21126931	240467394
2.	4323690	18770890	20481573	1973416	12948479	24132481	53091620	8499490	98006932
3.	717996	430799	4417390	100555	21505023	9639124	1611219	353809	12920875
4.	986816	96830	996890	382	3960326	2124938	2446684	7411123	74132

Table 3. Number of mispredictions for the 1. "Total," 2. "lat<5," 3. "15<=lat<20," and 4. "lat>=100" datapoints from Figure 2 for all our benchmarks.

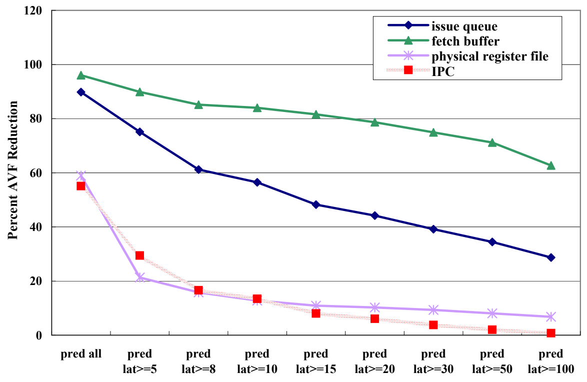

Figure 3. Percent AVF reduction in 3 hardware structures averaged across 9 SPEC2000 integer benchmarks by applying value prediction to instructions with varying latencies. The curve labeled "IPC" reports the percent IPC reduction for the same. All reductions are computed relative to no value prediction.

side of Figure 3). For example, when only predicting instructions with latency greater than or equal to 30 cycles, the performance impact is less than 3.8%. Of course, reliability improvement is not as great when predicting fewer instructions. But it can still be significant–we achieve a 74.9%, 39.2%, and 9.3% reduction in AVF for the fetch queue, issue queue, and register file, respectively at $\geq$ 30-cycle latency.

In general, Figure 3 shows there exists a tradeoff between reliability and performance. The more instructions we predict, the larger the improvement in reliability, but also the larger the degradation in performance. We find a good strategy is to focus the value predictor on long-latency instructions (*e.g.*, instructions with $\geq$ 30-cycle latency). This is because the longer the instruction latency, the smaller the impact mispredictions will have on performance. Furthermore, the longer the instruction latency, the more critical the instructions are from a reliability standpoint.

3.3 Confidence Estimation

Confidence estimation can be used to reduce the number of performance-degrading mispredictions. To investigate the potential benefits of this approach, we added a confidence estimator to our value predictor. Figure 4 reports the fraction of predicted, correctly predicted, and mispredicted instructions for all instructions, labeled "total," and for instructions with different latency ranges. (The format for Figure 4 is almost identical to Figure 2, except there is no "Result-Producing Instructions" curve since it would be the same). Compared to no confidence estimation, our value predictor achieves fewer correct predictions with confidence estimation. The reduction ranges between 10% and 15%. This is because the confidence estimator prevents predicting the less predictable instructions. As a result, the fraction of mispredicted instructions goes down to almost 0 across all latency ranges. As Figure 4 shows, our confidence estimator is quite effective at reducing mispredictions with only a modest dip in the number of correct predictions.

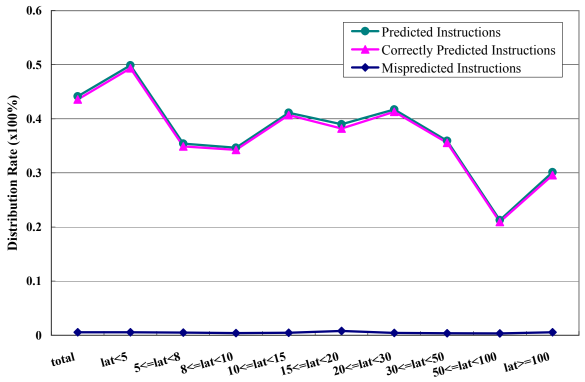

Figure 4. Fraction of predicted, correctly predicted, and mispredicted instructions–with confidence estimation–across different latency ranges over all 9 spec2000 integer benchmarks.

Table 4 reports the actual number of mispredictions with confidence estimation for all our benchmarks. (The format for Table 4 is identical to the format used in Table 3). Compared to Table 3, Table 4 shows the number of mispredictions with confidence estimation is indeed dramatically reduced relative to no confidence estimation.

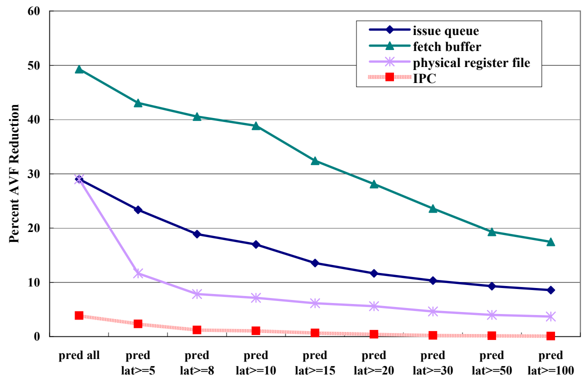

Figure 5. Percent AVF reduction in 3 hardware structures averaged across 9 SPEC2000 integer benchmarks by applying value prediction and confidence estimation to instructions with varying latencies. The curve labeled "IPC" reports the percent IPC reduction for the same. All reductions are computed relative to no value prediction.

Figure 5 shows the impact of confidence estimation on the AVF of our three hardware structures, as well as on IPC. (This figure uses the exact same format as Figure 3). In Figure 5, we see IPC never degrades more than 4% compared to no prediction, even when performing full prediction. These results show confidence estimation is indeed effective at mitigating performance degradation. Unfortunately, with confidence estimation, the reliability improvement is not as significant as before. In particular, under full prediction, the AVF for the fetch queue, issue queue, and register file is reduced by at most 49.3%, 29.0%, and 29.0%, respectively; under selective prediction with baseline latency of 30 cycles, the AVF for the fetch queue, issue queue, and register file is reduced by about 23.6%, 10.3%, and 4.6%, respectively. The lower reliability improvements compared to Figure 3 are due to the fact that confidence estimation suppresses prediction of many instructions, reducing the coverage achieved by the value predictor.

Thus far, we have applied confidence estimation uniformly across all instructions–*i.e.*, we use a single confidence threshold to determine whether any particular instruction should be predicted or not. However, predicting all instructions using a uniform confidence level may not be the best policy since instructions do not contribute equally to reliability nor to performance impact. In particular, for longer latency instructions which contribute more to overall reliability and incur less performance degradation during mispredictions, it may be better to perform value prediction more aggressively. Conversely, for shorter latency instructions which contribute less to overall reliability and incur more performance degradation during mispredictions, it may be better to perform value prediction less aggressively. This suggests an adaptive confidence estimation technique has the potential to more effectively tradeoff reliability and performance.

We modify our confidence estimation scheme to adapt the confidence threshold based on each instruction's latency. In particular, we employ three different threshold levels, similar to what is proposed in [10]. (The thresholds for low, medium, and high confidence are 3, 7, and 15, respectively for a saturating value of 15). We use the lowest confidence threshold for instructions that incur a latency equal to or larger than 4 times the baseline latency; we use the medium confidence threshold for instructions that incur a latency equal to or larger than 2 times the baseline latency but smaller than 4 times the baseline latency; and we use the highest confidence threshold for instructions that incur a latency equal to or larger than the baseline latency but smaller than 2 times the baseline latency. Here, the baseline latency is the minimum instruction latency that is considered for prediction as given by latency-based selective prediction. (For example, if we only predict instructions with latency 5 cycles or larger, then the low, medium, and high thresholds are applied to instructions with latency in the ranges $\geq$ 20 cycles, 10-19 cycles, and 5-9 cycles, respectively).

Figure 6 shows the impact of adaptive confidence estimation on the AVF of our three hardware structures, as well as on IPC. (This figure uses the exact same format as Figures 3 and 5). As suggested by the above discussion, in these experiments we combine latency-based selective prediction with adaptive confidence estimation. In other words, we only consider for prediction those instructions that meet the latency threshold given along the X-axis of Figure 6, and for a given candidate instruction, we only predict it if its saturating counter meets the corresponding confidence threshold for its latency. As Figure 6 shows, adaptive confidence estimation incurs a relatively small performance degradation similar to the baseline confidence estimation technique shown in Figure 5. A particularly small performance degradation, about 5.4%, is achieved when

	twolf	gcc	gap	gzip	bzip2	perl	parser	mcf	vpr
1.	861000	840153	2206916	328586	12479900	4512412	7806876	2712208	800670
2.	219590	545557	1036604	176886	2786259	1498557	4007555	691834	637547
3.	145622	10486	78163	4943	6583023	720734	172598	18595	14652
4.	82034	2158	10958	5	46766	388546	129277	1620071	263

Table 4. Number of mispredictions for the 1. "Total," 2. "lat<5," 3. "15<=lat<20," and 4. "lat>=100" datapoints from Figure 4 for all our benchmarks.

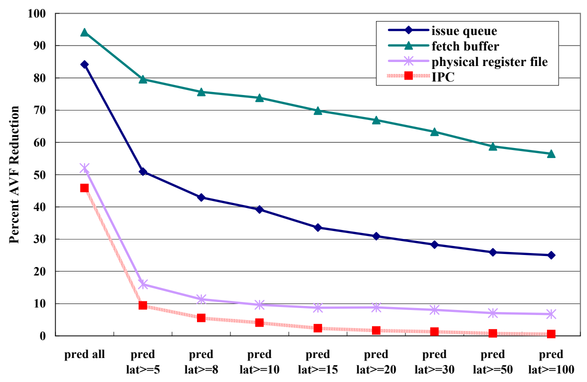

Figure 6. Percent AVF reduction in 3 hardware structures averaged across 9 SPEC2000 integer benchmarks by applying value prediction and adaptive confidence estimation to instructions with varying latencies. Confidence threshold used for each prediction (high, medium or low) varies according to the instruction's latency. The curve labeled "IPC" reports the percent IPC reduction for the same. All reductions are computed relative to no value prediction.

limiting prediction to instructions with a latency of at least 8 cycles or larger. However, adaptive confidence estimation achieves a much better reliability improvement (AVF reduction) than the baseline confidence estimation, and approaches the reliability improvement achieved by value prediction without confidence estimation shown in Figure 3. For example, under selective prediction with baseline latency of 30 cycles, the AVF for the fetch queue, issue queue, and register file is reduced by about 63.3%, 28.3%, and 8.1%, respectively, while the performance is only degraded about 1.3%. Thus, by more aggressively predicting only the longer latency instructions, adaptive confidence estimation can cover the most critical instructions for reliability without sacrificing too much on performance.

4 Related Work

This work is related to several areas of research in fault tolerance. The first area includes studies which exploit explicit redundancy–by duplicating program execution either in hardware [1, 2, 3, 4, 5] or software [6, 7, 8]–to detect or recover from faults. In contrast, we study value prediction to explore the redundancy inherent in programs. Our technique avoids the overhead from explicitly duplicating computation for fault detection. However, value prediction cannot achieve 100% correctness, thus it cannot ensure failure-free execution while explicit duplication can. Our goal is to reduce the fault rate in a more cost-effective way, which is still meaningful for most systems that do not require failure-free execution. In addition, our technique considers fault vulnerability at the instruction level which is ignored by most existing techniques. By quantifying instruction's vulnerability, we selectively protect instructions that are most susceptible to faults, thus reducing the impact of mispredictions while still maintaining acceptable reliability.

In the area of exploiting inherent program redundancy, the work most related to ours is [15]. Racunas *et al* make use of value perturbation to prevent possible faults. Their technique tries to identify the valid value space of an instruction, which is done by tracking the instruction's past results. Future outputs that are not within the recorded valid value space are considered as potentially corrupted. Compared to value perturbation, value prediction tries to predict an instruction's result exactly. Outputs that are not equal to predicted values are considered as potentially corrupted. Compared to detecting value perturbations, value prediction can be more precise in finding discrepancies. For example, an instruction's past value space may be so big that corrupted values may still fall in the valid value space, and hence, cannot be detected.

Our technique is also related to the area of partial fault protection. Recently, some studies [12, 16] propose that traditional full-coverage fault-tolerant techniques are only necessary for highly-reliable and specialized systems, while for most other systems, techniques which tradeoff performance and reliability are more desirable. For example, Weaver *et al* [12] try to reduce error rate by flushing the pipeline on L2 misses. Gomaa *et al* [16] propose a partial-redundancy technique which selectively employs redundant thread or instruction-reuse buffer for fault detection. The triggering of their redundancy technique is determined by program performance. Compared to their work, we exploit program's inherent redundancy for detecting possible faults. In addition, by characterizing instruction vulnerability, we selectively protect the most fault-susceptible instructions to achieve better coverage.

5 Conclusion

This paper investigates applying value prediction for improving fault tolerance. We make the observation that value predictability is a low-cost (albeit imperfect) form of program redundancy. To exploit this observation, we propose to use the output of a value predictor to check the correctness of predicted instructions, and to treat any mismatch as an indicator that a fault has potentially occurred. On a mismatch, we trigger recovery using the same hardware mechanisms provided for mispeculation recovery. To reduce the misprediction rate, we characterize fault vulnerability at the instruction level and only apply value prediction to instructions that are highly susceptible to faults (*i.e.*, those with long latency). We also employ confidence estimation, and adapt the confidence estimator's threshold on a per-instruction basis tuned to the instruction's latency. Instructions with higher latency are predicted more aggressively, while instructions with lower latency are predicted less aggressively. Our results show significant gains in reliability with very small performance degradation are possible using our technique.

6 Acknowledgements

This research was supported in part by NSF CAREER Award #CCR-0093110, and in part by the Defense Advanced Research Projects Agency (DARPA) through the Department of the Interior National Business Center under grant #NBCH104009. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsement, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

References

[1] R. W. Horst, R. L. Harris, and R. L. Jardine, "Multiple instruction issue in the NonStop Cyclone processor," in *Proc. of the 17th Int'l Symp. on Computer Architecture*, May 1990.

[2] Y. Yeh, "Triple-triple redundant 777 primary flight computer," in *Proc. of the 1996 IEEE Aerospace Applications Conference*, Feb. 1996.

[3] S. K. Reinhardt and S. S. Mukherjee, "Transient Fault Detection via Simultaneous Multithreading," in *Proc. of the 27th Annual Int'l Symp. on Computer Architecture*, June 2000.

[4] J. Ray, J. C. Hoe, and B. Falsafi, "Dual use of superscalar datapath for transient-fault detection and recovery," in *Proc. of the 34th annual IEEE/ACM Int'l Symp. on Microarchitecture*, Dec. 2001.

[5] S. S. Mukherjee, M. Kontz, and S. K. Reinhardt, "Detailed design and evaluation of redundant multithreading alternatives," in *Proc. of the 29th annual Int'l Symp. on Computer Architecture*, May 2002.

[6] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Control-flow checking by software signatures," in *IEEE Transactions on Reliability*, March 2002.

[7] N. Oh, P. P. Shirvani, and E. J. McCluskey, "Error detection by duplicated instructions in super-scalar processors," in *IEEE Transactions on Reliability*, March 2002.

[8] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. Aug., "SWIFT: Software implemented fault tolerance," in *Proc. of the 3rd Int'l Symp. on Code Generation and Optimization*, March 2005.

[9] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors," in *Proc. of the 13th annual IEEE/ACM Int'l Symp. on Microarchitecture*, Dec 1997.

[10] B. Calder, G. Reinman, and D. Tullsen, "Selective Value Prediction," in *Proc. of the 26th Annual Int'l Symp. on Computer Architecture*, May 1999.

[11] S. S. Mukherjee, C. T. Weaver, J. Emer, S. K. Reinhardt, and T. Austin, "A Systematic Methodology to Compute the Architectural Vulnerability Factor for a High-Performance Microprocessor," in *Proc. of the 36th annual IEEE/ACM Int'l Symp. on Microarchitecture*, Dec. 2003.

[12] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt, "Techniques to reduce the soft error rate of a high-performance microprocessor," in *Proc. of the 31st Annual Int'l Symp. on Computer Architecture*, June 2004.

[13] D. Burger, T. Austin, and S. Bennett, "Evaluating future microprocessors: the simplescalar tool set," Tech. Rep. CS-TR-1996-1308, Univ. of Wisconsin - Madison, July 1996.

[14] B. Goeman, H. Vandierendonck, and K. de Bosschere, "Differential FCM: Increasing Value Prediction Accuracy by Improving Table Usage Efficiency," in *Proc. of the 7th Annual International Symp. on High-Performance Computer Architecture*, 2001.

[15] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee, "Perturbation-Based Fault Screening," in *Proc. of the 2007 IEEE 13th Int'l Symp. on High Performance Computer Architecture*, Feb 2007.

[16] M. Gomaa and T. N. Vijaykumar, "Opportunistic Transient-Fault Detection," in *Proc. of the 32nd Annual Int'l Symp. on Computer Architecture*, June 2005.