Cost-Effective Graceful Degradation in Speculative Processor Subsystems: The Branch Prediction Case*

Sobeeh Almukhaizim, Thomas Verdel and Yiorgos Makris
Electrical Engineering Department
Yale University
{sobeeh.almukhaizim, thomas.verdel, yiorgos.makris@yale.edu}@yale.edu

Abstract

We analyze the effect of errors in branch predictors, a representative example of speculative processor subsystems, to motivate the necessity for fault tolerance in such subsystems. We also describe the design of fault tolerant branch predictors using general fault tolerance techniques. We then propose a fault-tolerant implementation that utilizes the Finite State Machine (FSM) structure of the Pattern History Table (PHT) and the set of potential faulty states to predict the branch direction, yet without strictly identifying the correct state. The proposed solution provides virtually the same prediction accuracy as general fault tolerant techniques, while significantly reducing the incurred hardware overhead.

1 Introduction

VLSI advances have led to great increases in processor speeds. However, the overall computer system performance has not improved proportionately to processor speeds. System performance is dominated by the performance of its subsystems [1, 2]. Therefore, in an act to boost system performance, various speculative enhancements have been proposed. Data prefetching [3], cache management techniques [4], value prediction techniques [5] and branch prediction [6, 7, 8] are all examples of such speculative subsystems.

Speculative execution introduces a whole new set of challenges and opportunities in designing reliable and fault-tolerant systems. The underlying principle is that speculative subsystems do not always perform useful computation. Often, speculation leads to invalid operation execution and the corresponding results are discarded, thus not providing any performance benefit. Nevertheless, functional correctness is maintained through an inherent recovery mechanism. Interestingly, such speculative execution complicates the behavior of a system in the presence of a fault [9, 10]. While functional correctness is not be violated, the performance benefit of speculative execution is lost. However, a reliable system should ensure not only the correct functional operation of the design, but also its performance specifications. Consequently, the general framework for providing fault-tolerance needs to be reexamined when designing speculative subsystems.

*This work is partially supported through a fellowship from Kuwait University.

In general, fault-tolerance methods explore the trade-off between attained fault coverage and incurred hardware overhead. Speculative subsystems add a third dimension to this decision space, namely the level of performance that can be maintained in the presence of a fault. Performance reduction in a faulty speculative subsystem depends on the speculative subsystem under consideration. In this work, we investigate the effect of performance faults in the Branch Predictor (BP) on the prediction accuracy. We propose a fault tolerant implementation that utilizes the FSM structure of the branch predictor and a set of potentially faulty states in order to correctly predict the direction of the branch. As compared to two general fault tolerant implementations, the proposed solution identifies a more cost-effective point in the trade-off space between hardware overhead and prediction accuracy.

The remainder of this paper is organized as follows: Traditional fault tolerant implementations of the BP are described in section 2. The proposed method is outlined in section 3. Experimental results are provided in section 4 and conclusions are drawn in section 5.

2 General Fault Tolerant BP Design

The general structure of a BP is shown in figure 1.a. The operation of a BP can be divided into three stages: 1) obtain branch history, 2) predict branch direction, and 3) update branch history [11]. When a branch is encountered, the BP indexes into a Pattern History Table (PHT) and obtains the branch history stored in one of the PHT 2-bit counters. The history is used to predict the direction of the branch. The PHT counter used for prediction is updated depending on the branch direction. We limit the added fault tolerance power to the branch prediction stage, as it is the only exact common stage in all BPs [6, 7, 8] and the dominant component in terms of area. A straightforward triplication of the indexing logic and the update logic, along with the addition of a majority voting function, would suffice to tolerate a fault in these stages.

2.1 Triple Modular Redundancy

The idea of Triple Modular Redundancy (TMR) [12] is to tolerate faults in a circuit by comparing the output behavior of three identical circuit replicas. Since only one circuit is assumed faulty, we can utilize a majority function to tolerate the effect and, therefore, continue to produce the correct output behavior. Hence, the counter of every PHT entry is

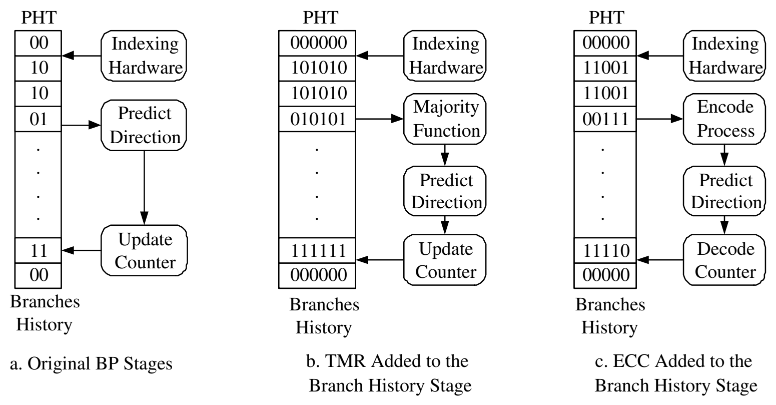


Figure 1. Original BP and General Fault Tolerant Variations.

triplicated, as shown in figure 1.b. The prediction is made by taking a majority vote among the three counters. The update process of the 2-bit counter is slightly modified in order to reflect the update on all three counters.

The area cost of the fault tolerant BP using TMR is almost three times the cost of the original design, as the size of the PHT, the dominant component of the BP in terms of area, is triplicated. The BP is a component on the critical path of application execution, thus careful consideration of the added delay should be taken. Nonetheless, the time taken by the majority function can be typically hidden in the same cycle required to obtain the prediction.

2.2 Error Correcting Codes

Hamming code bits [13], are added to the information bits of the PHT counters to encode the states such that a distance of 3 is enforced between the code words. Decoding hardware is utilized to determine the error location, and consequently, flip the erroneous bit to obtain the correct result. In this case, each entry in the PHT now contains 5 bits, i.e. 2 information bits and 3 code bits, to represent the encoded counter state. Once a branch is encountered, the encoded history is fed into a decoder to check whether there exists a fault. If a fault is present, the history is corrected and the direction of the branch is predicted. When the branch direction is resolved, the PHT counter used for the prediction is updated and the new state is encoded and stored back into the PHT entry. A fault tolerant BP using Hamming code is illustrated in figure 1.c.

Compared to the 4 additional bits of TMR, ECC require only 3 additional bits. The encoding latency does not affect the critical path operation as it takes place after predicting the branch. The decoding process, however, affects the critical path as the state must be decoded to predict the branch direction; nonetheless, it can also be typically hidden in the same cycle required to obtain the prediction.

3 Proposed Fault Tolerant Design

A BP does not require a strict identification of the correct state; rather, only the correct *prediction* that should be made. Furthermore, the set of possible next states depends on the current state and branch direction; therefore, the counter can only be set to a value in the *set of reachable next states*. We will demonstrate how to utilize the FSM structure to reduce the area overhead at a minor performance loss.

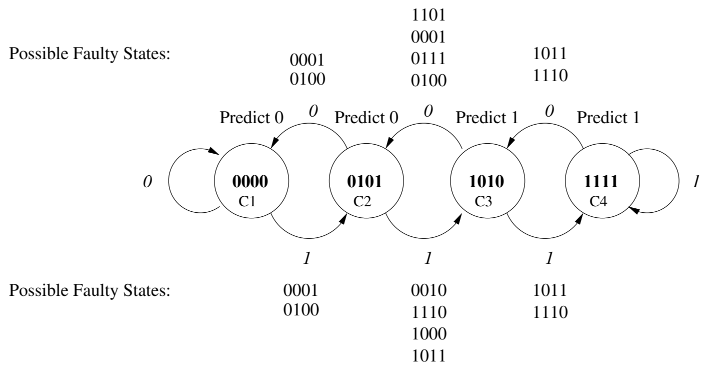
3.1 Overview

We replicate the counter once, thus storing two 2-bit counters in each entry. The new FSM of the counter is described in figure 2.a; states C1 and C4 represent the strongly not taken and taken predictions, while states C2 and C3 represent the weakly not taken and taken predictions, respectively. Following the transitions in the FSM, we obtain the set of faulty states at a *distance of one* from the final state, assuming a single-bit error. The sets of faulty states between the correct states of the FSM counter are shown in figure 2.a. We notice that, based on the *prediction* to be made, the sets are disjoint. This is further illustrated in the Karnaugh map of figure 2.b. F1 and F2 states are the faulty states that can be reached when the correct state should be C1 or C2; F3 and F4 are similarly defined. The prediction, when the current state is C1, C2, F1 or F2 (C3, C4, F3 or F4), should be 0 (1), regardless of whether the state is correct or faulty.

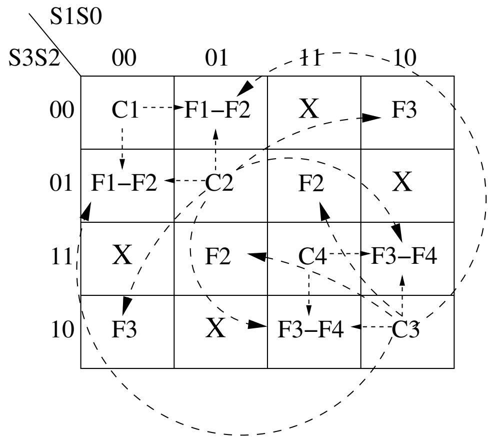
The operation of the proposed method is identical to the standard operation if a fault is not present. When a fault is detected, the correct prediction is made and the history must be updated with the branch outcome. However, setting the next state to any state might result in an overlap between the sets of faulty states. The only states where the associated sets of faulty states are disjoint are states C2 and C3. Consequently, when a fault is present, the next state is set to C2 or C3 depending on the branch direction. Based on this new set of next states, the counter will behave as a *1-bit* counter with C2 and C3 corresponding to predict-0 and predict-1.

3.2 Implementation

Figure 3 illustrates the prediction identification, predict direction and update counter stages of the BP. When a state is read from the PHT entry, a prediction is made based on the function described in the Karnaugh map of figure 2.b; the corresponding hardware is illustrated in the lower right part



a. New Counter FSM with all Possible Single Faults.



b. Karnaugh Map Illustrating the Faulty States Behavior.

Figure 2. New Counter FSM and a k-map Explanation of its Behavior.

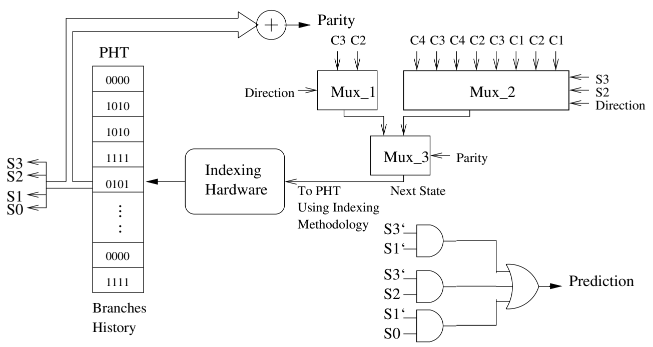


Figure 3. Proposed fault tolerant branch predictor.

of figure 3. An XOR gate identifies any change in the parity of a state, as all correct states have even parity. In fault-free operation, the update stage utilizes *Mux_2* to select the next state following the FSM description of figure 2.a. If a fault is present, *Mux_1* is used to set the next state to C2 or C3. All the BP stages can be implemented using 17 gates only. The ECC and TMR implementations can predict in the *exact* same way as a fault-free BP would. The proposed methodology, however, starts making predictions based on a 1-bit saturated counter. Nevertheless, the experimental results indicate that a 1-bit counter for the faulty entry has minimal effect on the overall branch prediction accuracy.

4 Experimental Results

We compare the fault tolerant implementations in terms of area overhead and branch prediction accuracy. We implemented all three fault tolerant BPs using SimpleScalar [14] and simulated the SPEC 95 integer benchmarks suite using the gshare [6] and global [7] BPs. In order to investigate the worst case degraded BP accuracy, a single-bit error was injected in the most heavily used PHT entry.

The area required for the proposed implementation is half the TMR area and two thirds the ECC area. The increase in misprediction rate is zero for the ECC and TMR implementations as a fault is always corrected. The gshare BP accuracy is illustrated in the first major heading of Table 1 for various PHT sizes. The increase in misprediction accuracy, when an error is injected in the most used PHT entry, is illustrated in the second major heading. As expected, increasing the PHT size has little effect on reducing the increased misprediction rate since the most used entry is still used frequently to predict biased branches. The third major heading illustrates the increase of the misprediction accuracy for the proposed scheme. The average misprediction increase is less than 1% of the misprediction increase in the original implementation. In some cases, the branch prediction accuracy of the proposed method is higher than the prediction accuracy of the original gshare predictor. The prediction accuracy for these cases increases because the prediction of a 1-bit counter outperforms the prediction of a 2-bit counter for the faulty entry. The results when a global BP is used are very similar as illustrated in table 2.

The proposed scheme requires twice the area as the original BP. A question that arises is whether the area spent to make the BP fault tolerant is better spent in doubling the PHT size. Although doubling the PHT size increases the branch prediction accuracy, it does not increase the accuracy of the BP enough to offset the diminished accuracy of a fault in the BP. As an example, quadrupling the PHT size of the gshare BP from 4K to 16K increases the prediction accuracy by 2.27% on average. A fault, however, might increase the misprediction accuracy by 57.55% on average. Clearly, the area is better utilized in order to make the BP fault tolerant.

5 Conclusions

A cost-effective fault-tolerant design method for BPs is described and compared to general fault tolerant ap-

Benchmark Name	*Branch Prediction Accuracy (gshare Predictor)*				*Increase in Misprediction Rate (gshare Predictor)*				*Increase in Misprediction Rate (Proposed Method)*			
PHT Size	1K	4K	16K	64K	1K	4K	16K	64K	1K	4K	16K	64K
compress	92.16	93.78	94.43	94.78	20.92	23.52	22.79	25.16	0.00	0.00	0.00	0.00
gcc	80.01	85.19	88.56	90.76	3.60	4.54	5.67	5.85	−0.05	0.07	0.08	0.11
go	71.54	73.26	78.16	83.67	2.64	2.31	2.31	2.74	0.04	0.04	0.04	0.12
ijpeg	90.06	91.00	92.06	92.64	101.71	109.69	121.58	128.26	1.81	2.11	2.27	2.58
li	93.81	96.75	97.36	97.62	27.67	48.22	59.41	61.56	0.00	0.00	0.00	0.00
m88ksim	94.13	96.08	96.82	97.23	115.10	160.24	184.32	198.02	1.19	1.79	2.52	2.17
perl	87.18	94.46	97.24	98.15	4.08	9.44	18.07	23.15	0.00	0.00	0.00	0.00
vortex	90.98	95.55	97.89	98.76	50.60	102.42	212.75	360.38	0.22	0.00	0.47	0.00
Average	87.48	90.76	92.82	94.20	40.79	57.55	78.33	100.77	0.40	0.50	0.66	0.62

Table 1. Performance of the Proposed Method on SPEC95 Integer Benchmarks, gshare predictor.

Benchmark Name	*Branch Prediction Accuracy (global Predictor)*				*Increase in Misprediction Rate (global Predictor)*				*Increase in Misprediction Rate (Proposed Method)*			
PHT Size	1K	4K	16K	64K	1K	4K	16K	64K	1K	4K	16K	64K
compress	84.89	85.35	85.73	85.93	14.16	14.36	15.29	12.65	1.46	1.43	1.47	0.00
gcc	75.29	77.68	79.05	80.05	17.77	17.87	17.83	17.65	0.61	0.45	0.38	0.50
go	70.60	72.13	74.96	77.33	13.04	11.70	10.78	10.26	0.99	0.79	0.76	0.66
ijpeg	78.81	79.00	79.69	79.79	62.78	56.92	54.66	52.33	2.03	1.29	1.23	0.79
li	83.63	84.45	84.98	84.97	20.70	20.82	20.36	20.34	0.06	0.00	0.00	0.00
m88ksim	85.97	86.48	86.59	86.69	66.61	66.39	64.11	61.81	1.28	1.33	1.42	1.43
perl	79.26	81.59	82.76	83.42	5.70	4.18	4.79	4.63	0.15	0.00	0.00	0.00
vortex	82.01	83.31	84.03	84.52	32.01	33.62	33.91	34.27	0.50	0.48	0.25	0.07
Average	76.31	81.25	82.22	82.84	29.10	28.23	27.72	26.74	0.88	0.72	0.69	0.43

Table 2. Performance of the Proposed Method on SPEC95 Integer Benchmarks, global predictor.

proaches. Although in the presence of a fault the proposed scheme does not necessarily identify the correct state, it still yields the correct prediction. In essence, an error in a PHT FSM reduces the 2-bit counter to a 1-bit counter, but only for the faulty PHT entry. Experimental results indicate that the proposed method gracefully degrades the effectiveness of the BP in the presence of faults: while only a minor impact on the overall prediction accuracy is observed, the incurred hardware overhead is significantly reduced as compared to general fault tolerant approaches.

Acknowledgement

We would like to acknowledge the guidelines of Daniel Friendly, Yale University, on the experiments needed to evaluate the proposed technique. The first author would like to thank Ozgur Sinanoglu and Alex Orailoglu, University of California at San Diego, for their insights on this project.

References

[1] J. Ousterhout, “Why aren’t operating systems getting faster as fast as hardware?,” in *USENIX*, 1990, pp. 247–256.

[2] T. Anderson, H. Levy, B. Bershad, and E. Lazowska, “The interaction of architecture and operating system design,” in *Proc. of 4th ASPLOS*, 1991, pp. 108–120.

[3] N. P. Jouppi, “Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers,” in *Proc. of 17th ISCA*, May 1990, pp. 364–373.

[4] D. Kroft, “Lookup-free instruction fetch/prefetch cache organization,” in *Proc. of 8th ISCA*, May 1981, pp. 81–87.

[5] K. Wang and M. Franklin, “Highly accurate data value prediction using hybrid predictors,” in *Proc. of 30th MICRO*, 1997, pp. 281–290.

[6] S. McFarling, “Combining branch predictors,” Tech. Rep. TN-36, DEC, June 1993.

[7] T. Y. Yeh and Y. N. Patt, “Alternative implementations of two-level adaptive branch prediction,” in *Proc. of 19th ISCA*, May 1992, pp. 124–134.

[8] E. Sprangle, R. Chappell, M. Alsup, and Y. Patt, “The agree predictor: A mechanism for reducing negative branch history interference,” in *Proc. of 24th ISCA*, May 1997, pp. 284–291.

[9] S. Almukhaizim, P. Petrov, and A. Orailoglu, “Low-cost, software-based self-test methodologies for performance faults in processor control subsystems,” in *Proc. of CICC*, May 2001, pp. 263–266.

[10] S. Almukhaizim, P. Petrov, and A. Orailoglu, “Faults in processor control subsystems: Testing correctness and performance faults in the data prefetching unit,” in *Proc. of ATS*, November 2001, pp. 319–324.

[11] D. A. Patterson and J. L. Hennessey, *Computer Architecture: A Quantitative Approach*, chapter 4, Morgan Kaufmann Publishers, Inc, second edition, 1996.

[12] J. Neumann, “Probabilistic logics and the synthesis of reliable organisms from unreliable components,” 1956, pp. 43–98, Princeton University Press.

[13] P. W. Hamming, “Error detecting and correcting codes,” Tech. Rep., Bell Syst. Tech. J., 1950.

[14] D. Burger and T. M. Austin, “The simplescalar tool set, version 2.0,” Tech. Rep. 1342, University of Wisconsin-Madison, Computer Sciences Department, June 1997.