Evaluation of Branch Prediction Strategies

Anvita Patel, Parneet Kaur, Saie Saraf
Department of Electrical and Computer Engineering
Rutgers University

CONTENTS

LIST OF FIGURES

LIST OF TABLES

I. Introduction

Modern computer architectures implement instruction-level, data-level and thread level parallelism, among other methods, to increase the instructions per cycle (IPC) and decrease the program execution time. In instruction level parallelism, multiple instructions can be overlapped using pipelining. The pipeline must have a continuous flow of instructions in order to perform effectively. If the instructions are fetched sequentially in a pipeline, jumps and conditional branch instructions can cause control hazards. As illustrated in Figure 1 for a five level pipeline, when a branch instruction is fetched (here, I2: *BEQ* at address 1004), its decision and the target is known only at its execution stage in the piepline. If instructions I3 and I4 are already in the pipeline and the actual branch target is determined to be instruction I5, then instructions I3 and I4 must be flushed and instruction I5 should be fetched. As a result of flushing the pipeline, an instruction may not be effectively dispatched every cycle, thus, decreasing the IPC. Consequently, branch instructions limit the effectiveness of parallelism.

A dedicated hardware is provided in modern day processors to predict the direction of the branches before the branch decision is actually known. If the branch direction is predicted correctly, the processor continues a smooth flow of instructions. However, branch misprediction can lead to the penalties such as flushing the pipeline, calculation of previous addresses and registers and increased CPI. The more the number of stages in a pipeline, more is the misprediction latency. Thus, it is very important to predict the branch correctly. Branch prediction strategies can be broadly divided into two categories: static and dynamic [1]. The static strategies make the same decision every

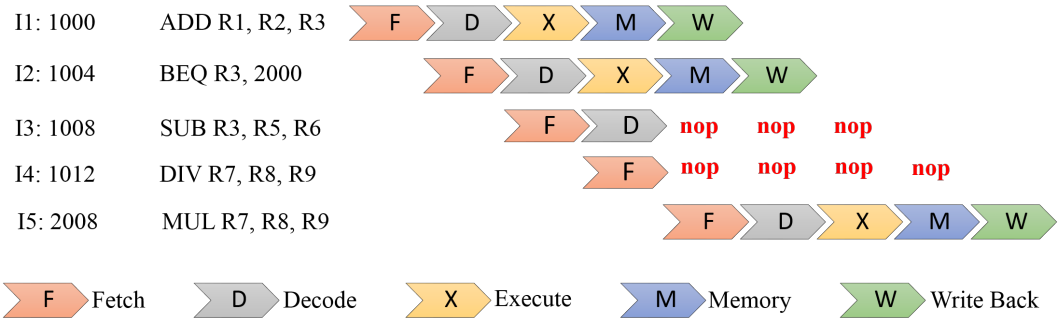

Fig. 1: Control hazard due to branch instructions. For the branch instruction I2 its branch direction decision and the target are known only at its execution stage. If instructions I3 and I4 are already in the pipeline and the actual branch target is determined to be instruction I5, then instructions I3 and I4 must be flushed and instruction I5 should be fetched. As a result of flushing the pipeline, an instruction may not be effectively dispatched every cycle, thus, decreasing the IPC.

time such as branch always taken and are program sensitive. The dynamic strategies make the decisions based on the last executions of a program and have been shown to be more accurate than the static strategies. Once it is predicted that a branch is taken or not, branch target prediction is used to guess the target address of the branch. However, branch target prediction is beyond the scope of this project.

Simplescalar [2] is an open source computer architecture simulator, which models a virtual computer system whose configurations such as cache memory and branch predictor can be specified by the user. The branch prediction simulator in Simplescalar implements the state of the art algorithms: branch always not taken, branch always taken, bimodal, two-level, combinational and meta predictors. It also provides software infrastructure for the development of new branch predictors and can be used for evaluating prediction rate and miss rate of programs.

In this project, we implement two branch prediction strategies proposed by James Smith [1] using Simplescalar simulator and compare them to its default strategy, *Bimodal*. We refer to these strategies as *Smith-strategy6* and *Smith-strategy7* in the rest of this report. Five benchmarks are used to evaluate and compare these branch prediction strategies: *anagram, cc1, compress95, go* and *perl*. Figure 2 shows the percentage of conditional branches in each benchmark. Conditional branches are a significant part of the total number of instruction, emphasizing the necessity of having an accurate branch predictor. Further, the performance of branch predictors is analyzed on the given benchmarks to identify the behavior of the benchmarks which can be improved. The conflict due to same address hasing to the same index location in a history table causes aliasing and increases the misprediction rate. Based on our analysis, we propose three ideas 1) Coincide Predictor, 2) Combinational Predictor, and 3) New hash function for Smith-strategy7. Finally, all the strategies are compared

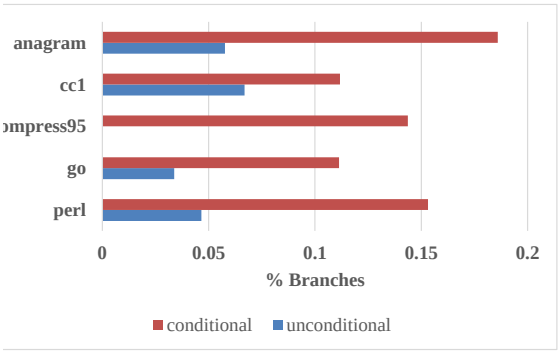

Fig. 2: Percentage of branch instructions in given benchmarks.Conditional branches are a significant part of the total number of instruction, emphasizing the necessity of having an accurate branch predictor.

using misprediction rate ans instructions per cycle (IPC).

Rest of the report is organized as follows: section III provides an overview of the existing branch predictors: bimodal, Smith-strategy6 and Smith-strategy7. In section IV, new branch predictors are proposed based on the analysis of prior predictors on the given benchmarks. Following this, section V presents the results and discussion of our experiments. Finally, section VI concludes this project.

II. Related Work

Many approaches have been implemented in literature to increase the performance by branch prediction using static as well as dynamic stratigies. In [1], seven strategies are discussed to make the branch prediction accurate including static, dynamic and improved dynamic strategies. Static branch prediction strategies predict all branches to be taken or not taken, but this prediction gives less accuracy. Other static approaches are prediction based on the operation code, or by analyzing the branch behavior and predicting taken or not taken separately for each branch instruction [3]. In case of compiler synthesized dynamic branch prediction, compiler synthesizes the branch information and statistics into one predicate bit representing most recent history of execution of the branch and then this information could be used by the hardware to make predictions for current branch instruction. Most branch predictors use a branch target buffer which is a small associative memory that will contain the addresses of the most recent branches and their target locations [4].

In [5], an agree predictor is proposed, which uses an extra one bit table for biased bit to reduce the interference problem to decrease the conflict of two branches indexing the same table entry in the dynamic branch prediction. Each biased bit is dedicated to each branch instruction of the program that uniquely identifies that branchs behavior. In [6] combining branch predictors are discussed for taking advantage of the most effective branch predictor for each branch. In [7] variable length path branch prediction is proposed in which hash function varies dynamically according to profiling information of the branch. The two-level selector mechanism can also be used as an improvement in the performance of the hybrid branch predictor [8].

III. Existing Branch Predictors

A. Bimodal

The bimodal predictor is the default branch predictor in Simplescalar simulator. The bimodal predictor takes advantage of history of the recent branch executions. It uses 2-bits for each branch to store the history and based on the bit values it predicts the current branch instructions behavior. The bits are updated after every execution based on correct or incorrect prediction. The 2-bits table is generated as a part of a random access memory. The m bits of the current branch address are hashed to index the 2-bit table which contains the history of respective branch execution. The table size 2^m depends on the selection of m bits. If the counter bit value is 2 or 3,

the branch is predicted to be taken. On the other hand, if the counter bit value is 0 or 1, the branch is predicted to be not taken. The prediction, if wrong, updates the counter in negative direction with the minimum limit to 0. Similarly, the prediction, if correct, updates the counter in positive direction with maximum limit to 3. Hence, 0 represents strongly not taken, 1 partially not taken, 2 partially taken and 3 strongly taken. The branches can be initialized as one of these four states.

B. Smith-strategy6

Smith-strategy6 is illustrated in Figure 3, which uses a 1-bit counter to predict the branch direction. The lower *m*-address bits of the branch instruction are hashed to index a history table in random access memory. The indexed memory location contains one-bit history counter, which is the outcome of the most recent execution of that branch. This data bit decides the prediction direction for current execution. If the counter value is 1, the branch direction is predicted as *taken*, else if it is 0, the branch direction is predicted as *not taken*. If the prediction is determined to be wrong, the history bit is updated accordingly. The default prediction or precisely first time location access prediction can be initialized as taken or not taken for all the branches. We initialize the branches as taken for our implementation. The history table size depends on the m-bits used for hashing of the address (2^m entries for *m*-bits).

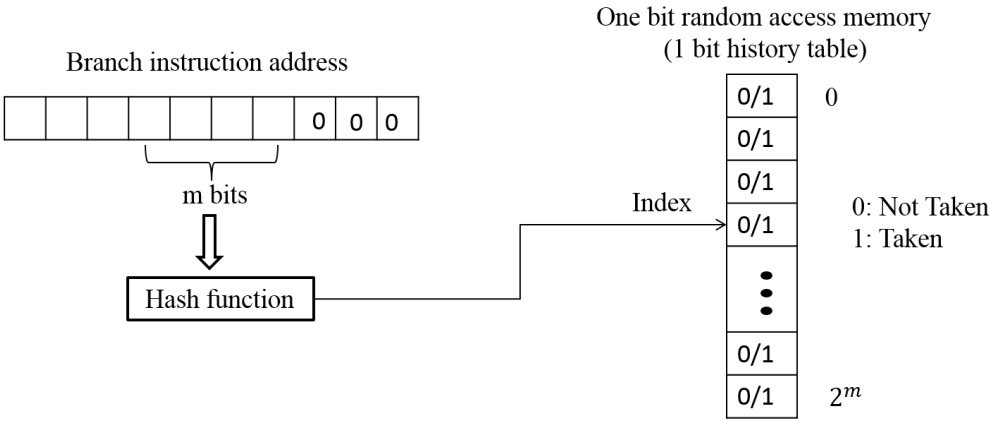

Fig. 3: Decision making for Smith-strategy6.

C. Smith-strategy7

Smith-strategy7 is illustrated in Figure 4, which is designed to deal effectively with the anomalous decisions. It uses an N-bit two‘s complement counters to predict the branch direction. Similar to Smith-strategy6, the lower *m*-address bits of the branch instruction are hashed to index a history table in random access memory. The indexed memory location contains an N-bit two‘s complement history counter,

which represents the previous executions of that branch. If the sign-bit is 0, the branch direction is predicted as *taken*. If the sign-bit is 1, the branch direction is predicted as *not taken*. The default prediction or precisely first time location access prediction can be initialized as taken or not taken for all the branches. We initialize the branches as taken (set to 0) for our implementation. The counter is incremented if the branch is *taken* and decremented if it is *not taken*. The history table size depends on the m-bits used for hashing of the address (2^m entries for *m*-bits).

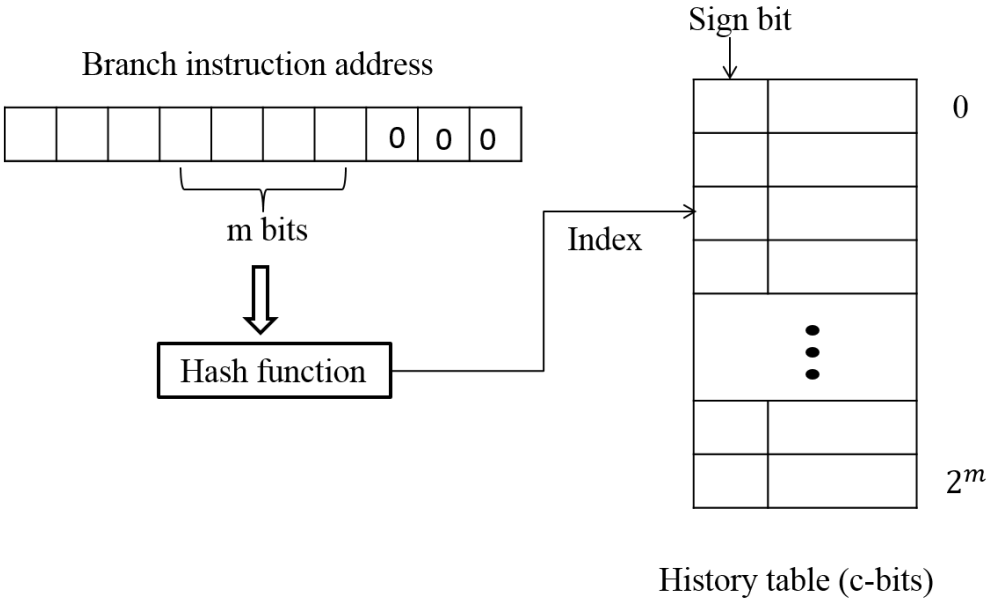

Fig. 4: Decision making for Smith-strategy7.

IV. Proposed branch predictors

We implemented three ideas to improve the performance of the existing branch predictors: 1) Coincide Predictor, 2) Combinational Predictor, 3) New hash function for Smith-strategy7. These methods are designed to decrease the conflicts arising due to aliasing, where multiple address are indexed to same table entry in the counter history tables.

A. *Coincide Predictor*

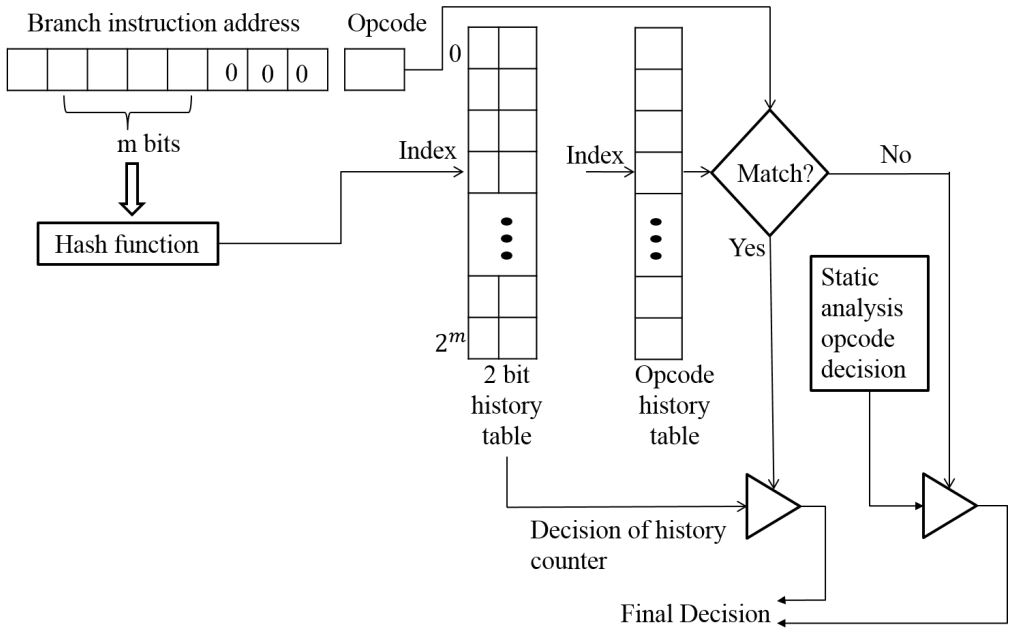

Fig. 5: Decision making for Coincide Predictor.

If two branch instructions have same lower m-bit address, they are indexed to same location in counter history table. The conflicts of such different branch instructions can be avoided by incorporating conditional branch behavior based on the given benchmarks. To take into account the behavior of the conditional branches, the profiling information of benchmarks can be incorporated within a dynamic predictor. In this strategy, the lower m-bits of instruction address are used to hash to the history counter table as well as to an opcode history table. The opcode history table stores the corresponding opcode of the branch conditional instruction being hashed. During decision making if the opcode of the conditional branch instruction matches the hashed entry in the opcode history table, the history table prediction (taken/not taken) is considered. If the opcode doesn't match, then the decision is based on the static decision of conditional branches obtained from benchmark behavior (Figure 5). On an opcode mismatch, the opcode history table is updated and the history counter is initialized.

To obtain the static opcode decision, all the benchmarks are profiled to determine the branch decision for each conditional branch instruction. A confusion matrix for all the branch instruction opcodes in each benchmark gives some interesting insights of its actual behavior. One such confusion matrix is shown in Table I for opcode *beq* in benchmark *cc1*. From this matrix it is determined that for benchmark *cc1*, the probability of *bne* being *taken* is higher it being *not taken* while more number

of instructions that are actually *taken* are mispredicted than those which are actually *not taken*. Based on these statistics for all the branch instruction opcodes in each benchmark, the maximum vote mechanism determines the static decision of the conditional branch opcode. Table II shows the static decisions for the conditional branch instructions based on the given benchmarks. We found that the opcodes *beq, blez* and *bltz* are most likey *not taken* while the the opcodes *beq, blez* and *bltz* are most likey *taken*. A limitation of this method is that it is sensitive to the benchmarks being used. Larger benchmarks with variety of programs will help determine the accuracy of this method on a wide variety of programs.

		Predicted		
		Taken	Not taken	Total
Actual	Taken	3309696	388672	4698368
	Not taken	222101	4139834	4361935

TABLE I: Confusion matrix for *bne* in benchmark *cc1*. The probability of *bne* being *taken* is higher it being *not taken* while more number of instructions that are actually *taken* are mispredicted than those which are actually *not taken*.

	anagram	**cc1**	**compress95**	**go**	**perl**	**Static Decision**
beq	NT	NT	NT	NT	NT	NT
bne	T	T	T	T	T	T
blez	NT	NT	T	NT	T	NT
bgtz	NT	T	T	T	T	T
bltz	NT	NT	NT	T	NT	NT
bgez	T	T	T	T	T	T
bc1f	T	T	NT	T	T	T
bc1t	T	T	T	T	NT	T

TABLE II: Static decisions for conditional branch instructions based on analysis of given benchmarks. Using maximum vote mechanism determines the static decision of the conditional branch opcode.

B. Combinational Predictor

The new strategy is proposed to overcome the shortcoming of the previous strategy6 and Bimod strategy. We need to combine the advantages of various branch predictors together to yield a better accurate branch predictor. Mapping of the different branch addresses to the same branch history table is called as Aliasing, which can be reduced with the increased table size. But there will be always a tradeoff between the table size and aliasing.

In the combinational branch predictor, 2 branch predictors working independently of each other are used and the selector will choose the best branch predictor. So the idea is that if both the predictors predict the same, then no change is done. While if the direction of prediction is different for both the predictors then only evaluation is done and counter is incremented [9].

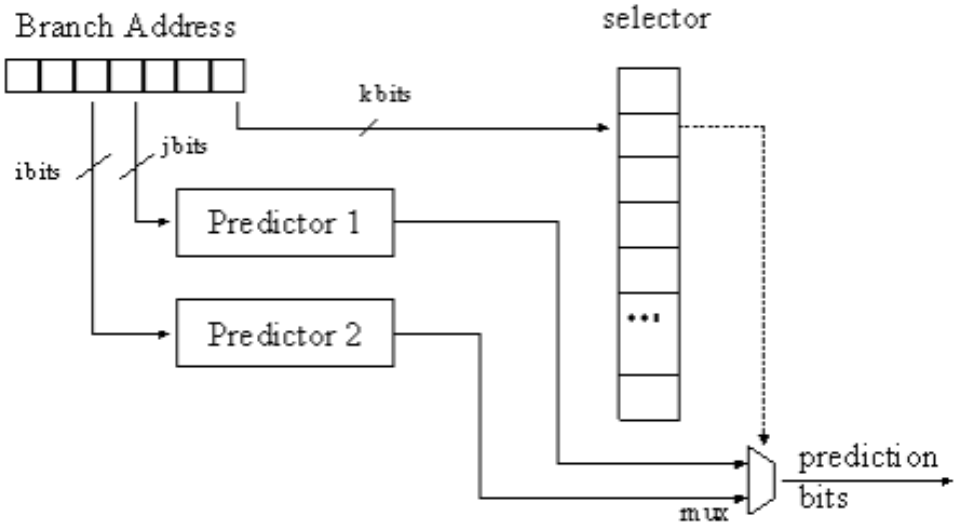

Fig. 6: Combinational Predictor. [6]

Predictor 1:strategy7	Predictor 2:2level	Result
Correct	Correct	No change
Correct	Incorrect	Decrement counter
Incorrect	Correct	Increment counter
Incorrect	Incorrect	No change

Fig. 7: Logic of Combinational Predictor. [9]

Thus in the new strategy a combined predictor containing the counter array will be used, which will select the best branch predictor to use. In this proposed new strategy, strategy7 and 2 level predictor strategy is combined. So p1 and p2 will be the 2 predictors combined. So p1 is the strategy 7 and P2 two level strategy [6]. 2 bit up/down counter is used. Each counter is keeping track of which predictor is more accurate and will decide to use the most accurate predictor.

A 2 level branch predictor uses a 2 dimensional counter called as pattern history table, thus it is based on the concept that the output of the branch will depend upon the other inter correlated branches and the history of the same branch

C. Varying Hash Function for Smith-strategy 7

In Smith-strategy 7, lower m-bits of instruction address are used to index the address to the history table. The addresses with the same lower m-bits hash to the same index

location in the history table and this conflict causes misprediction. If the index function is varied to take into account the higher h-address bits, then the potential conflict with among the instructions away from each other by a significant number of locations can be avoided. If we XOR the higher h-bits with lower m-bits, the higher address bits are taken into consideration for computing the history table index. This hash function is the same as that used by the default Bimod strategy in Simplescalar simulator.

V. Results and Discussion

Misprediction rate and instructions per cycle (IPC) for all the benchmarks are used to compare the existing and proposed branch prediction strategies. Misprediction rates and IPC of all the strategies for history table-size 2048 are compared in Figure 8 and 9. Smith-strategy6 is a poor predictor as compared to Bimodal and other strategies. Since it uses a 1-bit history counter, this result is expected. Coincide strategy gives lower misprediction rate for benchmarks *cc1, go* and *perl* than the existing strategies. In addition, using Bimod hash decreases the misprediction rate only in benchmark *compress95*.

Figures 10, 12 and 11 show the misprediction rate and IPC for varying size of history table for existing and proposed branch predictors. For bimodal, Smith-strategy6 and Smith-strategy7, as the table size increases, the misprediction rate decreases for all the benchmarks (Figure 10(a)-(c)). This is because as the table size increases, lower m-bits used to index the history table increase, reducing the conflicts due to different aliasing. Additionally, IPC increases with increase in table size (Figure 11(a)-(c)). For Smith-strategy7, as the counter-bit length is increased for a fixed table size, the prediction rate first increases from 1-bit to 3-bits and then remains almost constant as the counter bits are increased (Figure 10(d)). These results are in agreement with the results in [1]. As Smith points out in his paper, using 3-bit or higher bits does not improve the results further as too much history does not allow a quick correction of a wrong result. Among the given benchmarks, *anagram* and *compress95* have less misprediction rate (and greater IPC) as compared to benchmarks *perl,cc1* and *go*. We observe similar results for varying table size for the prediction strategies proposed in section IV. As the table size increases, misprediction rate decreases while IPC increases.

Combinational strategy results are shown in Figure 13 and 14. The branch Prediction rate increased in case of benchmarks Go,anagram and cc1.While benchmarks Perl and compress95 showed a little increase in the branch prediction rate.

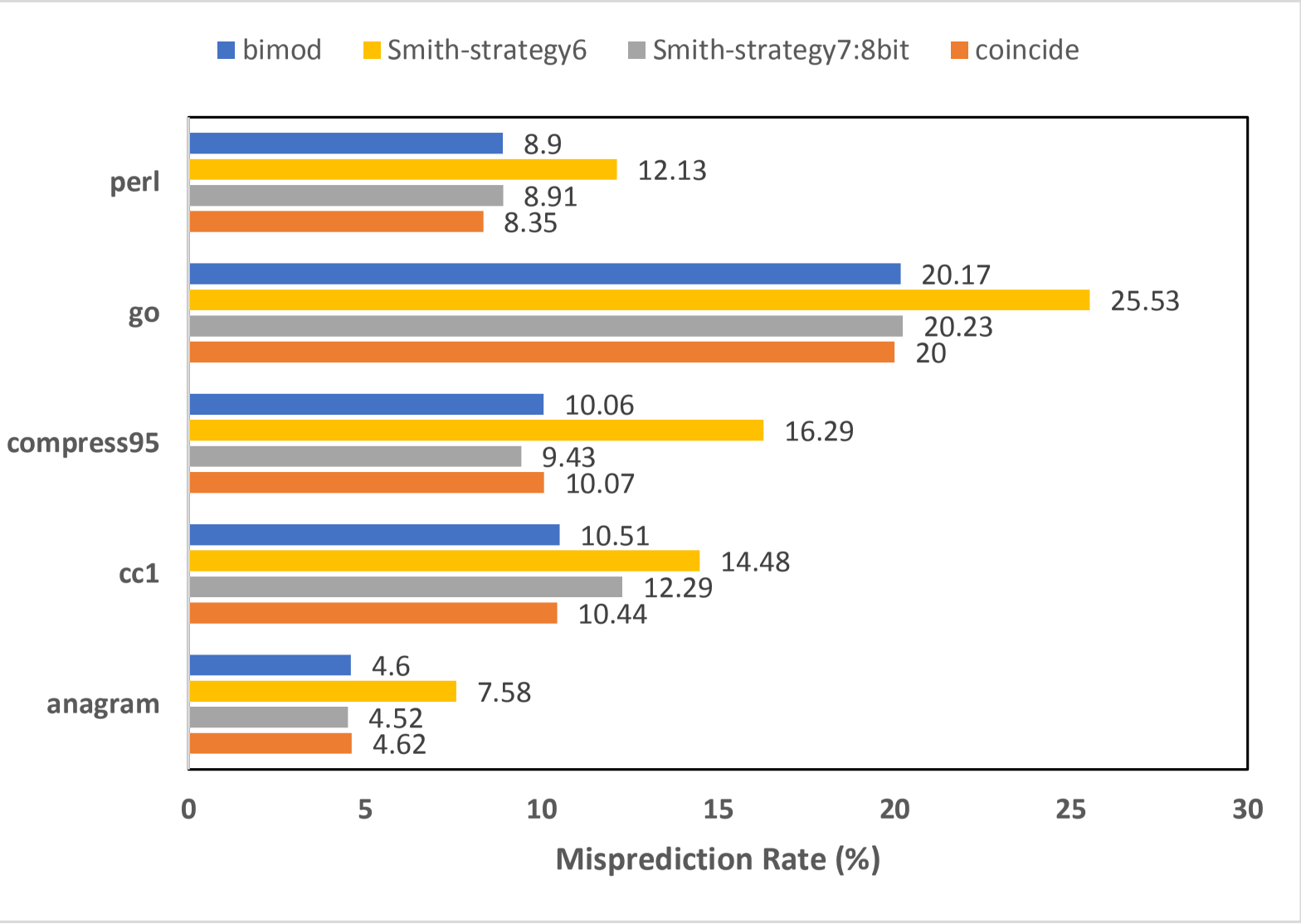

Fig. 8: Smith-strategy6 is a poor predictor as compared to Bimodal and other strategies. Since it uses a 1-bit history counter, this result is expected. Coincide strategy gives lower misprediction rate for benchmarks *cc1, go* and *perl* than the existing strategies.

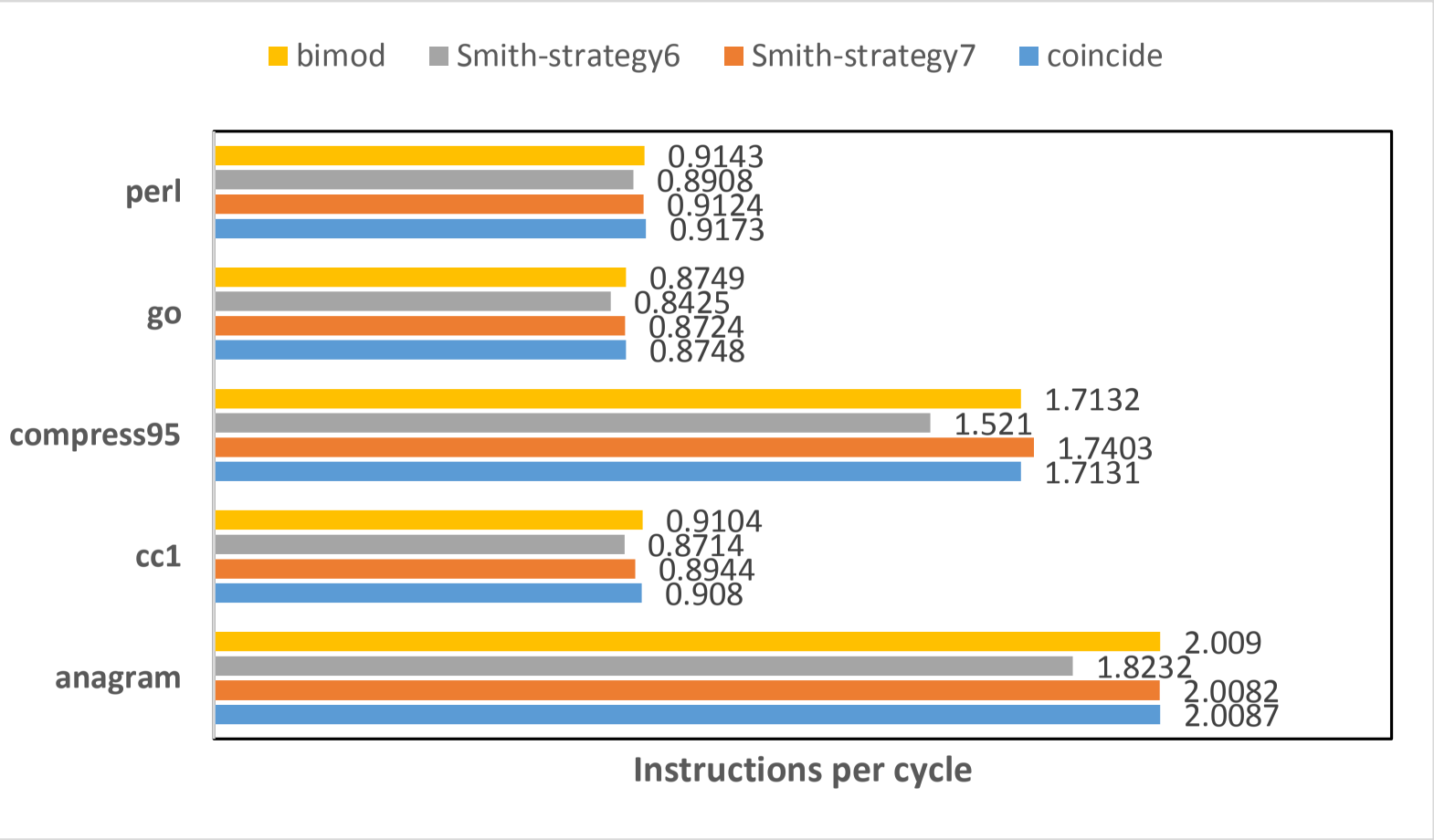

Fig. 9: Comparison of all strategies: Instructions per cycle (IPC).

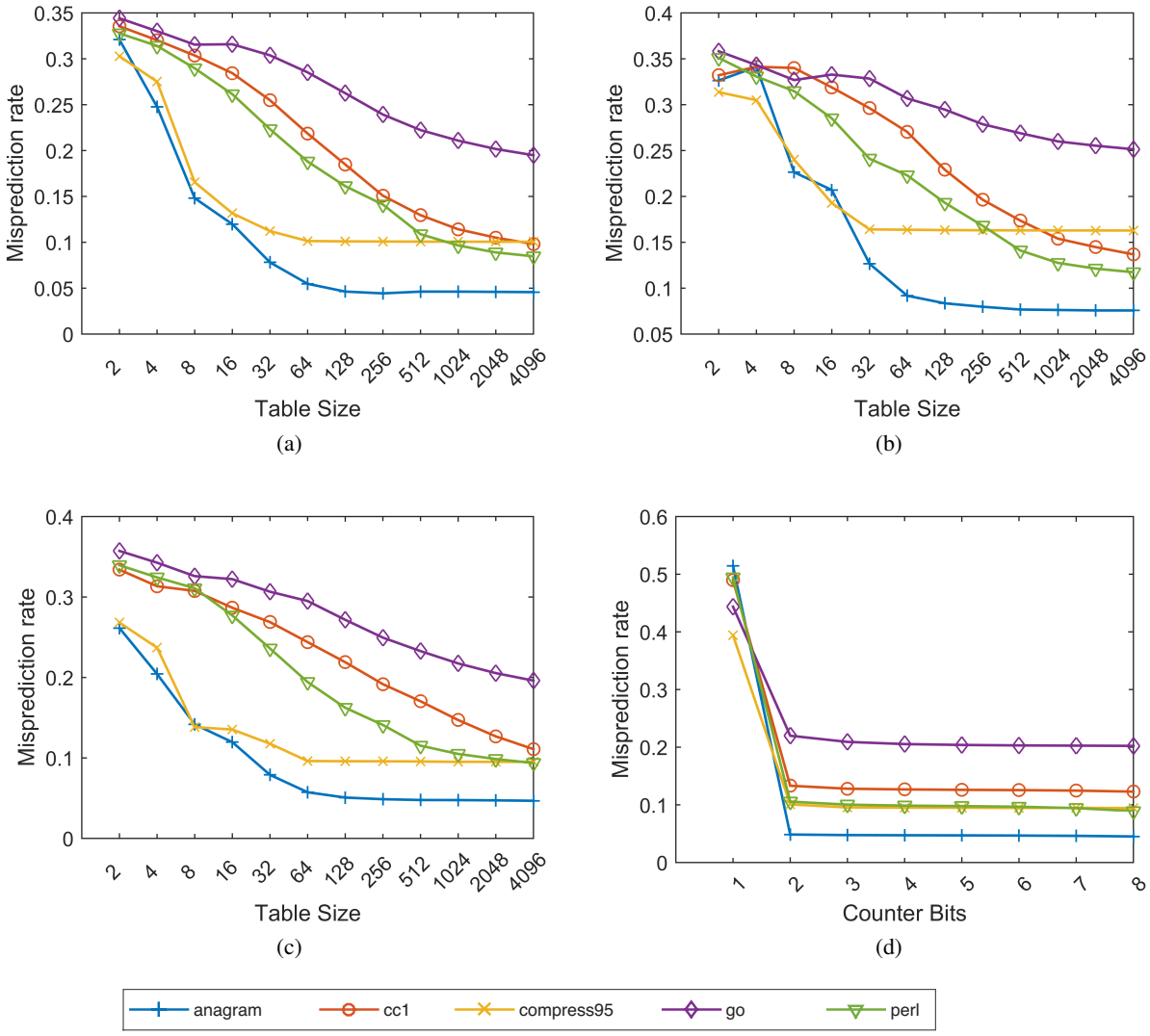

Fig. 10: Misprediction rate (MPR). (a) Bimodal: MPR vs. table size. (b) Smith-strategy6: MPR vs. table size. (c) Smith-strategy7: MPR vs. table size with 2-bit counter. (d)Smith-strategy7: MPR vs. counter bits for 2048 entry table. For bimodal and Smith-strategy6, MPR decreases as the table size increases. For Smith-strategy7, MPR decreases as the table size increases for a given counter bit length. For a fixed table-size, as the counter bits are increased, MPR first decreases from 1-bit to 2-bit and remains almost constant as the counter bits are increased.

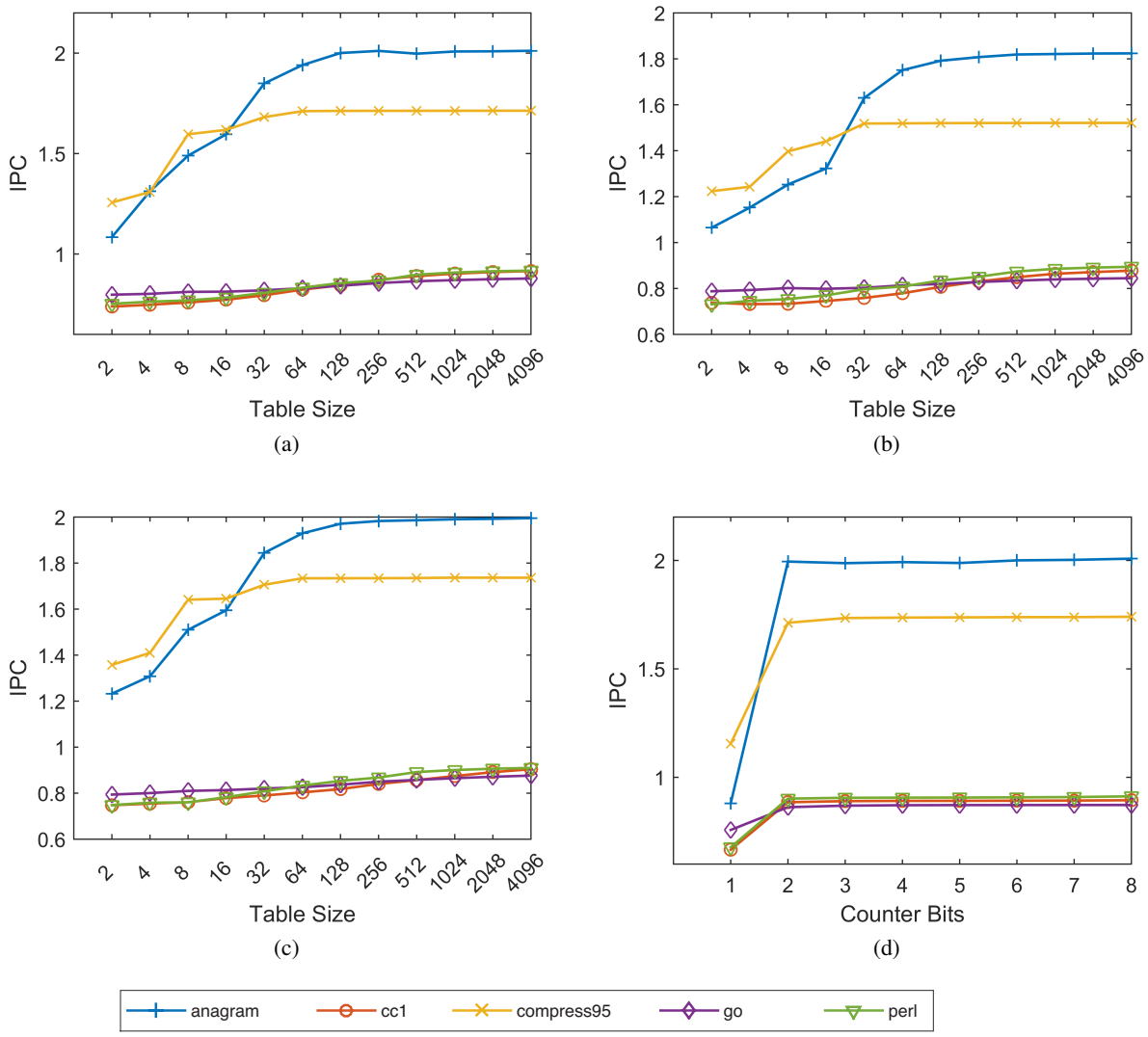

Fig. 11: Instructions per cycle (IPC). (a)Bimodal: IPC vs. table size. (b)Smith-strategy6: IPC vs. table size. (c)Smith-strategy7: IPC vs. table size with 2-bit counter. (d)Smith-strategy7: IPC vs. counter bits for 2048 entry table. For bimodal and Smith-strategy6, IPC increases as the table size increases. For Smith-strategy7, IPC increases as the table size increases for a given counter bit length. For a fixed table-size, as the counter bits are increased, IPC first increases from 1-bit to 2-bit and remains almost constant as the counter bits are increased.

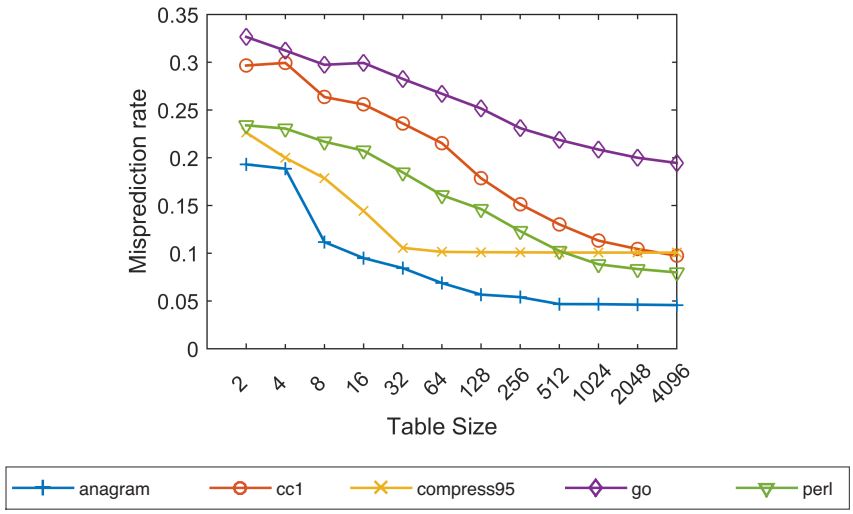

Fig. 12: Coincide predictor: Misprediction rate. Misprediction rate decreases as the table size is increased.

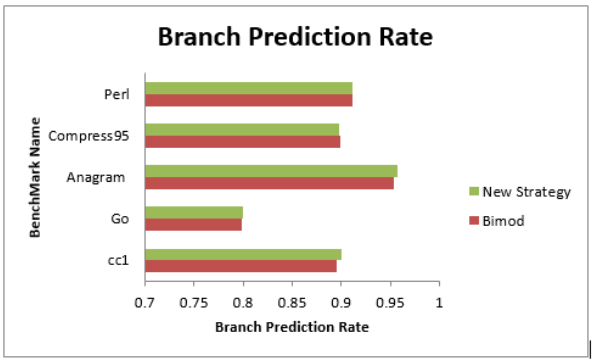

Fig. 13: Combinational predictor: Branch Prediction rate. Comparison with Bimod.

VI. Conclusions

For this project, first two existing branch prediction strategies are implemented (Smith-strategy6 and Smith-strategy7). Next, three ideas are proposed to improve the existing branch predictors: coinciding predictor, combinational predictor and using of a hash function for Smith-strategy7. Finally all the strategies are compared to the default Bimodal strategy. While Smith-strategy6 is a poor predictor, Smith-strategy7 with 3-bit(or more) counter has better performance on some of the benchmarks. Coincide predictor is proposed to overcome the mispredictions due to conflict of two adresses and utilizes static opcode decisions derived from given benchmarks. Combinational predictor combines two existing predictors to improve the overall result. The proposed

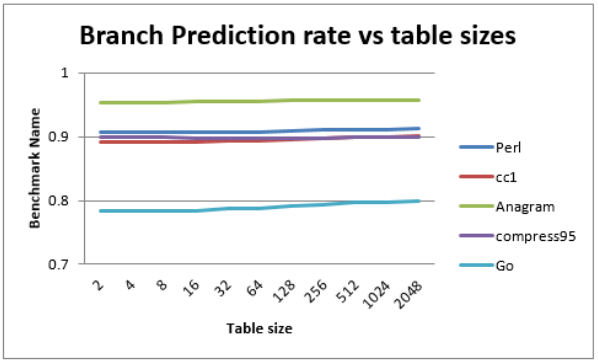

Fig. 14: Combinational predictor: Branch prediction rate vs table size. The branch Prediction rate increased in case of benchmarks Go,anagram and cc1. While benchmarks Perl and compress95 showed a little increase in the branch prediction rate.

predictors are marginally better for some of the benchmarks as they decrease the misprediction rate and increase the IPC.

REFERENCES

[1] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th Annual Symposium on Computer Architecture*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1981, pp. 135–148. [Online]. Available: http://dl.acm.org/citation.cfm?id=800052.801871

[2] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, Feb. 2002. [Online]. Available: http://dx.doi.org/10.1109/2.982917

[3] J. Lee and A. Smith, "Branch prediction strategies and branch target buffer design," *Computer*, vol. 17, no. 1, pp. 6–22, Jan 1984.

[4] D. August, D. Connors, J. Gyllenhaal, and W.-M. Hwu, "Architectural support for compiler-synthesized dynamic branch prediction strategies: Rationale and initial results," in *High-Performance Computer Architecture, 1997., Third International Symposium on*, Feb 1997, pp. 84–93.

[5] E. Sprangle, R. S. Chappell, M. Alsup, and Y. N. Patt, "The agree predictor: A mechanism for reducing negative branch history interference," in *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ser. ISCA '97. New York, NY, USA: ACM, 1997, pp. 284–291. [Online]. Available: http://doi.acm.org/10.1145/264107.264210

[6] S. McFarling, "Combining branch predictors," 1993.

[7] J. Stark, M. Evers, and Y. N. Patt, "Variable length path branch prediction," in *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS VIII. New York, NY, USA: ACM, 1998, pp. 170–179. [Online]. Available: http://doi.acm.org/10.1145/291069.291042

[8] P. yung Chang, E. Hao, and Y. N. Patt, "Alternative implementations of hybrid branch predictors," in *In Proceedings of the 28th Annual International Symposium on Microarchitecture*, 1995, pp. 252–257.

[9] T.-Y. Yeh and Y. N. Patt, "Two-level adaptive training branch prediction," in *Proceedings of the 24th Annual International Symposium on Microarchitecture*, ser. MICRO 24. New York, NY, USA: ACM, 1991, pp. 51–61. [Online]. Available: http://doi.acm.org/10.1145/123465.123475